An Empirical Study of the Extent and Causes of Technical Debt in Public Organizations Software Systems

Leonard Peter Binamungu[a,*], Danford Ephrahim Phiri[a], Fatuma Simba[a]

[a]*Department of Computer Science and Engineering, College of Information and Communication Technologies, University of Dar es Salaam, Dar es Salaam, Tanzania*

Abstract

Because of short time to market for software systems, suboptimal design and implementation choices are sometimes made by software teams. Technical Debt (TD) is the term used to describe software quality flaws that are caused by time-saving shortcuts. Previous research has focused on the identification, causes, and management of TD. However, most of previous studies on TD have been informed by private and open source software projects, lacking insights from the public sector. Researchers have argued that software systems in the public sector are different from software systems in the private sector and open source projects, because organizations in the public sector are subjected to specific restrictions and requirements. Thus, TD causes, challenges, and the strategies to manage TD might differ between the public sector and the private sector and open source projects, which calls for the need to study TD in the public sector.

To make a start in filling this gap, we studied 102 software systems and surveyed 73 practitioners from 13 public organizations in Tanzania, to understand the extent and causes of TD in the Tanzanian public sector. A substantial amount of TD was found in the studied systems. Architectural TD was the most accumulated type of TD, followed by Design TD and Code TD. Moreover,

*Corresponding author

Email addresses: lepebina@udsm.ac.tz (Leonard Peter Binamungu), danford.phiri1983@gmail.com (Danford Ephrahim Phiri), fatmasimba@udsm.ac.tz (Fatuma Simba)

Electronic copy available at: https://ssrn.com/abstract=4040641

modularity violations and project management issues were found to be the leading causes of TD in software systems of the studied public organizations. We, thus, recommend the investigation of TD management strategies that are suitable for software systems in the public sector. From a public sector viewpoint, we contribute empirical insights to the literature on the extent, causes, and management of TD in software systems.

Keywords: `Technical Debt`, Technical Debt Extent, Technical Debt Causes, Technical Debt Management, Public Organizations, Empirical Study

1. Introduction

In the modern world, organizations involved in the development of software systems are forced to support fast and continuous delivery of customer value in order to stay alive and competitive. Martini and Bosch [1] notes that, in such a situation, suboptimal decisions are sometimes necessary in order to minimize the time between identification of customer needs and delivery of a solution. However, despite its short-term benefits, this approach has negative effects on the quality of software systems, which consequently hinders future maintenance and evolution. Fernandez-Sanchez *et al.* [2] revealed that, over the past few decades, the total cost of maintaining software systems has raised from 35% to nearly 90% of the overall software project cost. They further found that, inability of software systems to accommodate evolving requirements was the main reason for the cost increase. Because of high maintenance cost, many software systems in organizations stop to be used shortly after deployment. Codabux *et al.* [3] assert that the main reason for the inability of software systems to accommodate evolving requirements is the accumulation of flaws introduced during software development and maintenance.

In 1992, Cunningham [4] coined the term Technical Debt (TD) to describe software quality flaws that are caused by time-saving shortcuts [4]. Nowadays, the term is widely used by software practitioners and researchers to communicate to non-technical people the need for refactoring. Similar to the financial regime,

Electronic copy available at: https://ssrn.com/abstract=4040641

TD has both principal and interest. Principal is the cost required to convert an actual system to an ideal one, while interest is the cost of maintaining the software for as long the TD remain unpaid. For a particular software system, TD is the sum of principal and interest. It is advisable to pay the TD in a software system before it gets out of hand, particularly before the interest becomes equal to or greater than the initial software development cost.

Many studies have been conducted to identify, quantify, examine the causes of and manage TD in software systems. However, most of the past studies on TD have focused on private and open source systems [5]. According to Nielsen *et al.* [5], software systems in the public sector are different from software systems in the private sector and open source projects, because organizations in the public sector are subjected to specific restrictions and requirements. As such, challenges posed by TD and the strategies to manage TD might differ between the public sector and the private sector and open source projects. Thus, Nielsen *et al.* identified the importance of studying TD in the public sector, to uncover the best ways to deal with TD in software systems owned by public organizations. Among other things, this will enable public organizations to make optimal use of resources and thus fully reap the benefits of digitization.

To fill the gap of scarcity of TD lessons from the public sector, we set out to study the extent and causes of TD in the public sector in Tanzania. Specifically, we analyzed 102 software systems from 13 public organizations in Tanzania, to understand the extent and technical causes of TD in public sector software systems. We also surveyed 73 professionals involved in the development, maintenance, and management of software systems in the Tanzanian public sector, to understand the social-technical causes of TD in the public sector. In particular, we sought answers for the following research questions (RQs):

- **RQ1:** To what extent is technical debt present in public sector software systems?
- **RQ2:** What are the causes of technical debt in public sector software systems?

Electronic copy available at: https://ssrn.com/abstract=4040641

Additionally, because organizations in the public sector can have differences in the maturity of software processes, affecting the quality of the software systems they develop, we wanted to understand if the Capability Maturity Level of a public organization has an influence on the accumulation of TD. We, thus, added the following research question:

- **RQ3:** Is there a relationship between the Capability Maturity Levels of public organizations and the accumulation of technical debt?

We found a substantial amount of TD in the studied software systems. Architectural Technical Debt (ATD) was the most accumulated type of TD, followed by Design Technical Debt (DTD) and Code Technical Debt (CTD). Moreover, modularity violations and project management issues were found to be the leading causes of TD in software systems of the studied public organizations. Based on the study results, we recommend the investigation of TD management strategies that are suitable for software systems in the public sector. Thus, from a public sector viewpoint, we contribute empirical insights to the literature on the extent, causes, and management of Technical Debt in software systems.

The rest of this paper is structured as follows. Section 2 presents the TD background and the related work. Section 3 presents the approach we used to select the studied systems, collect data, and analyze data. Section 4 presents the results and discussion. Section 5 presents the threats to the validity of the results in this study, and the strategies we used to mitigate the effects of such threats. Section 6 concludes the paper, highlighting opportunities for future work.

2. Background and related work

2.1. Background

TD is a result of quick and poor implementations of software systems artefacts, focusing on short-term benefits, but it makes software systems rigid to accommodate evolving requirements [6]. With respect to software maintenance,

Electronic copy available at: https://ssrn.com/abstract=4040641

Technical Debt is a gap between making a change perfectly and making a change quickly. In other words, TD is a distance between an actual design and an optimal design. Similar to the financial regime, TD is constituted of both principal and interest. TD principal is the cost of fixing technical quality issues of an existing software system to produce an ideal system, while TD interest is a cost spent on maintenance of an existing system due to technical quality issues [7]. The amount of TD in a particular system is a sum of principal and interest. TD interest is associated with interest probability, which determines whether the interest will grow over time as the software system expands.

If the TD present in a software system is left unpaid, it will grow to a Breaking Point (BP), a point in time when TD interest accumulated in a software system becomes equal to the initial software system development cost [8]. Avgeriou *et al.* [9, 6] noted that, TD causes organizations to spend large parts of their budgets and time on fixing bugs to avoid reaching a breaking point, instead of developing new features. This, in turn, negatively impacts digital transformation, modernization, and business velocity. In addition, as TD accumulates, the ability of organizations to respond to changes decrease. This makes it harder to accommodate additional functionalities requested by customers.

There is a common agreement in a TD community that ideal versions of software systems do not exist, and that there exist no methods in literature for approaching them [10]. To be more specific, all deployed software systems have a certain extent of TD accumulated. According to Lilienthal [11], long-lasting software architecture can only be achieved if organizations constantly manage instances of TD present in their software systems. Thus, organizations should inevitably manage their debt in order to achieve their short and long term objectives. To accomplish this, a mechanism is required to support organizations in making strategic decisions on when and to what extent TD items in their software systems should be refactored [12]. Moreover, a strategy can guide organizations to understand how much TD is tolerable, and when repaying of TD present in their software systems should be prioritized over fixing bugs and developing new features. From this point of view, identifying and quantifying

Electronic copy available at: https://ssrn.com/abstract=4040641

the TD present in software systems is essential for organizations' management to make informed decisions.

TD can be incurred at different phases of a software development lifecycle. Each TD type is associated with a specific phase, but one phase can have multiple TD types. Mapping TD types to software system development phases can enable organizations to apply specific knowledge to manage specific TD types. In addition, mapping of TD types to software development phases helps to reveal the effect of each type of TD on specific software artefacts. This, in turn, helps organizations to implement appropriate TD management strategies for different types of TD.

Apart from technical factors such as software design and implementation approaches, social-technical factors such as organizational structure and culture, team experience, code ownership, and emphasis on producing more functionalities faster have a huge impact on the accumulation of TD [13]. To be more specific, accumulation of TD in software systems of a particular organization depends on technical factors e.g., modularity violations as well as social-technical factors e.g., poor project management. This suggests that, to establish a TD management approach that suits a particular context, an investigation of factors influencing the accumulation of debt in that context must be performed.

In addition to technical and social-technical factors, TD is also related to the organization software process maturity level, defined by Capability Maturity Model Integration (CMMI). CMMI is a method for measuring the maturity of software development processes in organizations. It measures the maturity of software development processes on a scale of 1 to 5, 1 being poor adherence to best practices and 5 being optimal. A study by Li *et al.* [14] highlighted strong correlation between TD and other software systems maturity concepts, including CMMI. This implies that an organization can rise its TD maturity level by improving awareness of the TD concept and applying appropriate processes, tools and techniques in TD management. Thus, the influence of organizational CMMI level to the accumulation of TD in public software systems is worth an investigation.

Electronic copy available at: https://ssrn.com/abstract=4040641

2.2. Related Work

Previous studies have investigated the extent, causes and management of TD in software systems. Here, we discuss some, but more studies on these aspects have been well-summarized in [5, 15, 16, 2, 14, 17, 18, 19, 20, 21]. Some researchers have investigated TD in organizations of different sizes. For example, Besker *et al.* [22] studied the causes of TD accumulation in software systems developed by startups, the upsides of introducing TD, and the downsides of TD accumulation. *Inter alia*, the following organizational factors were found to affect the accumulation of TD: developers experience, knowledge of software development by founders of software startups, stability/growth of a software team, clarity of gains, absence of a development process, and freedom of developers.

Martini *et al.* [23] studied the causes of Architectural TD in five large companies, and modelled ATD accumulation and recovery over a period of time. The following factors were found to cause accumulation of ATD: business reasons; lack of documentation of architectural requirements; reuse of code from legacy software, open source, or third party software; development in parallel teams; lack of clarity on the effects of software modifications; incomplete refactoring; change of technology; and people-related reasons.

Other researchers have investigated the impact of software development approaches and processes on the accumulation of TD in the resulting software. For instance, Holvitie *et al.* [24] examined the effect of agile software development practices and processes on the accumulation of TD. Agile practices that are conducted systematically were found to reduce TD. They also found that legacy software systems have a tendency to harbor TD.

However, most of the previous studies on TD have focused on private sector and open source software systems, lacking insights about TD in software systems in the public sector [5]. Nielsen *et al.* [5] asserted that software systems in the public sector are different from software systems in the private sector and open source projects, because organizations in the public sector are subjected to specific restrictions and requirements. Moreover, the complexity of projects in the public sector is different from the complexity of projects in the private sector

Electronic copy available at: https://ssrn.com/abstract=4040641

due to the number of people affected by public projects, novelty, resistance to change, organization size and politics [25]. As such, challenges posed by TD and the strategies to manage TD might differ between the public sector and the private sector and open source projects. In particular, it could be that software development stakeholders in the public sector reason and treat TD differently from stakeholders in the private sector. Thus, Nielsen *et al.* identified the importance of studying TD in the public sector, to understand the scale of TD, the causes of TD and uncover the best ways to deal with TD in software systems owned by public organizations.

3. Methodology

This section details the methodology we followed to study the extent and causes of TD in public sector software systems. In particular, it describes the process we followed to select the systems that we studied, and presents the process we followed to collect and analyze the data about the extent and causes of TD in public sector software systems.

3.1. Selection of systems

Convenience sampling was used to select systems for the study. A request to study software systems was sent to 50 government organizations, 13 of which met the following criteria and were selected for the study. One, willingness to provide access to run automated static code analysis to the codebase in their GitLab repositories. Two, possession of at least one software development team. A total of 102 software systems from 13 organizations that met the selection criteria were selected for the study. The selected software systems were from government organizations that belonged to four different sectors: Telecommunication, Finance, Health, and Education.

Of the 102 studied software systems, 91 were in-house developed and 11 were acquired from vendors. This enabled us to capture TD in systems obtained through both of the two software acquisition practices that are commonly employed by government organizations. Though hard to generalize for the whole

Electronic copy available at: https://ssrn.com/abstract=4040641

public sector in Tanzania, the number of studied software systems and the diversity of sectors from which the studied systems came from are sufficient to give insights on the extent and causes of TD accumulation in software systems owned by the public sector.

3.2. Extent of Technical Debt in public sector software systems

Technical debt can cover different aspects of the system [14, 26]. However, the systems' information we could access enabled us to study only three types of TD: architectural, design, and coding (Table 1). So the analysis we provide hereafter is focused on these three types of TD. We followed two steps to determine the extent of TD in a particular system. First, identification of architectural, design, and coding violations in a system. A total of nine (9) Object-Oriented software quality violations caused by architectural, design and coding flaws (Table 2) were used in this study. Second, the lines of code covered by each identified violation were combined by other factors (described in Section 3.2.2) to determine the quantity of TD associated with specific violations.

3.2.1. TD identification

Three automatic static analysis tools were used to identify the violations of interest in the studied systems: Codacy, SonarQube and Code Inspector. These are the most popular automatic static analysis tools available in the literature [9]. The tools analyze source code against Object-Oriented software quality metrics to find indicators of violation of rules of good Object-Oriented design and coding. The use of three tools enabled us to compare the results and select the most accurate ones.

3.2.2. TD quantification

We devised a mechanism to quantify TD in the studied systems. In particular, the quantity of TD in a system was computed as a function of three variables: number of lines of code (LOC) in must-fix violations; the average effort required to fix violations to an ideal level, measured in *hours per line of code*; and the average cost of fixing violations to an ideal level, measured in *cost*

Electronic copy available at: https://ssrn.com/abstract=4040641

Table 1: Types of studied TD and their respective indicators (Source: Zazworka *et al.* [27])

Technical Debt Type	**Indicator**
Architectural TD	Dependency violations, modularity violation, compliance violations, system-level structure quality issues, lack of handling interdependent resources, and non-uniform usage of architectural policies and patterns
Design TD	Violations of the principles of good object-oriented design, code smell (Brain Class, Brain Method, Data class, Disperse Coupling, Feature Envy, Tradition Breaker, Shortgun Surgery, Intensive Coupling, God Class), Long functions/methods, Complex functions/methods, Deep, Nested Complexity, Violated Afferent/Efferent Coupling (AC/EC), Excess Number of Function Arguments, and Presence of Grime
Code TD	Missing comments, Code duplication, Slow algorithm, and Non-uniform naming or inappropriate naming

Electronic copy available at: https://ssrn.com/abstract=4040641

Table 2: Studied Object-Oriented software quality metrics (Source: Lanza [28])

Parameter	**Description**
Lines of Code (LOC)	Measures the size of a software system in terms of number of lines of code in a function, class or module. Smaller number of lines of code implies lower level of complexity.
Cyclomatic Complexity (CC)	Measures the number of independent execution paths in a software system source code. Lower number of execution paths implies higher quality.
Coupling Between Objects (CBO)	Measures the level of coupling between functions, classes or modules. Less coupling implies lower level of complexity.
Lack of Cohesion of Methods (LCOM)	Measures the level of encapsulation in a function, class or module. Higher cohesion implies low level of complexity.
Weighted Methods per Class (WMC)	Measures the sum of complexities in a software system function, class or module. Lower weight implies lower level complexity.
Response For a Class (RFC)	Measures the level of communication between functions, classes or modules. Fewer responses implies lower level of complexity.
Maintainability Index (MI)	Measures the ease of software system maintenance. Higher index implies higher maintainability and quality.
Number of Children (NOC)	Measures the total number of children nodes belongs to a given class. More children nodes imply higher level of complexity.
Depth of Inheritance Tree (DIT)	Measures the depth of a software system class inheritance hierarchy. Greater depth of inheritance implies higher level of complexity

Electronic copy available at: https://ssrn.com/abstract=4040641

of labor per hour. Thus, the quantity of TD present in the studied software systems was calculated by multiplying the values of the three variables: total number of lines of code in must-fix violations, average effort, and average cost.

To obtain the average effort in hours per LOC for a single developer, one software project from each of the studied organizations was randomly selected and used for effort estimation. Specifically, for a particular organization, information about the number of lines of code in the selected system, the number of developers who contributed to the system, and the duration of the project (in hours) were used to compute the effort of a single developer (in hours per line of code). Then, the average of the efforts of a single developer in the 13 studied organizations was used to obtain a value of the average effort required to fix violations to an ideal level. The average effort was found to be 0.04 hours/LOC. Moreover, we used data on the average salary of a software developer (obtained from HR departments of the studied organizations) to calculate average cost of labor per hour for a software developer in the studied organizations. The average cost of labor for the 13 studied organizations was found to be 4.2 USD/hour.

The following is the summary of values of independent variables used for TD quantification in the context of the studied public organizations.

- Average development cost per hour for a single developer (cost of labor per hour): 4.2 USD.
- Average development effort per hour for a single developer (hours per line of code): 0.04.
- Cost of developing a new line of code (USD/line): 4.2 *0.04 = 0.168.

3.3. Causes of Technical Debt in public sector software systems

Causes of TD can broadly be categorized into technical and social-technical causes [29]. This study investigated both technical and social-technical causes of TD in public sector software systems.

Electronic copy available at: https://ssrn.com/abstract=4040641

3.3.1. Technical causes of TD

To establish the most influential technical causes of TD in the software systems of the studied public organizations and the degree of influence of each technical cause to the accumulation of TD, we analyzed the extent of TD contributed by each indicator of violation of good software development practices (Table 1). The degree of influence of each technical cause to the accumulation of TD was determined by its contribution to the overall TD found in the studied software systems.

As well, to establish the strength of association between TD and technical causes of TD accumulation, correlation analysis was performed for each of the identified technical causes. The degree of association was determined by a correlation coefficient, r, which was used to rank the association of TD and technical causes of TD accumulation in the studied systems.

Graylin *et al.* [30] observed that Lines of Code (LOC), Cyclomatic Complexity (CC), and software system age are three internal software attributes that are most influential to the quality of software systems. So we wanted to understand whether TD accumulation in the studied software systems is affected by the quality of software systems. We, thus, performed correlation analysis between each of the three internal software attributes and the accumulation of TD in the studied systems. This enabled us to determine the influence of each internal software attribute to the accumulation of TD in public software systems. The degree of association was determined by correlation coefficient, r.

3.3.2. Social-technical causes of TD

To identify social-technical causes of TD accumulation, we surveyed 73 software development professionals from the 13 studied organizations. These were professionals with sufficient knowledge on TD. Table 3 shows the distribution of survey respondents. Two types of questions were on the questionnaire: questions that aimed to assess participants' knowledge on the TD concept, and questions that aimed to understand the influence of various social-technical factors on the accumulation of TD in the software systems of the studied organizations. The

Electronic copy available at: https://ssrn.com/abstract=4040641

influence of each social-technical cause of TD was presented on a scale of 0 (no influence) to 10 (maximum influence to the accumulation of TD). The questionnaire was sent to 156 software development professionals, 82 out of whom responded (52.6% response rate). Of the 82 respondents, 9 did not demonstrate understanding of TD, and so their responses were excluded from further analysis. We, therefore, remained with 73 responses, which informed the results we report. The scores for each social-technical cause of TD accumulation were aggregated to obtain a total score. Then, the top ten social-technical causes of TD accumulation in government organizations were identified.

Table 3: Distribution of survey respondents by role

Role	#	%
Developer	31	42%
System analyst	14	19%
Database administrator	13	18%
Business analyst	10	14%
Project manager	5	7%

3.4. Capability Maturity Level and TD accumulation

To assess the influence of organization software maturity level to the accumulation of TD, the capability maturity level questionnaires were distributed to ICT managers in the 13 studied organizations. The Software Engineering Institute (SEI) maturity questionnaire [31] was used to gather information about organizational adherence to best practices defined by SEI. To determine the effect of software process maturity level of an organization on the accumulation of TD, the ratio of TD to lines of code (LOC) was computed for each studied system. Dividing TD by LOC helped us to take care of variations in the sizes of the studied systems. We then constructed a linear regression model to help us understand if the Capability Maturity Level of a public organization has an influence on the accumulation of TD.

Electronic copy available at: https://ssrn.com/abstract=4040641

3.5. Ethical considerations

Since this study collected primary data, the following research ethics were observed. Participation in any aspect of this research was voluntary–express consent was obtained and documented from participants before they were involved in the study. To ensure confidentiality, participants' identities were anonymized in the publication of results. In connection to this, names of software systems and organizations involved in this study were coded. Also, research participants were not harmed in any way, i.e., safety and dignity of participants was prioritized.

4. Results and Discussion

We now present and discuss the results.

4.1. Extent of TD present in public sector software systems

Of the 102 software systems we studied, eight (8) were found to have reached a critical point, a point at which the extent of TD present in a software system is higher than the cost used to develop the system. Six (6) of the eight (8) software systems at critical point were acquired from vendors, and two (2) were developed internally. The fact that many of the systems at critical point were acquired from vendors could be explained by the fact that, software systems developed externally are often hard to maintain due to the difficulty in understanding the source code and other artifacts.

Moreover, an amount of TD equivalent to USD 2,441,353 was found in the 102 studied public organizations' software systems. This value was the TD principal at the time of investigation. Figure 1 shows how the different TD types were distributed in the studied public software systems. ATD was the most incurred, with 56.4% of the total debt found, followed by DTD (29.1%) and CTD (14.5%). Whether the debt is incurred accidentally or is because of deliberate trade-off decisions, ATD can be most expensive to recover [32]. Architectural decisions are usually made at early stages of system development. Correcting,

Electronic copy available at: https://ssrn.com/abstract=4040641

at a later point, early software deployment decisions is usually associated with high interest payment. So it could be that ATD is higher than DTD and CTD in the studied software systems because software teams are scared of the high costs that could be associated with attempts to correct mistakes in the software architectures.

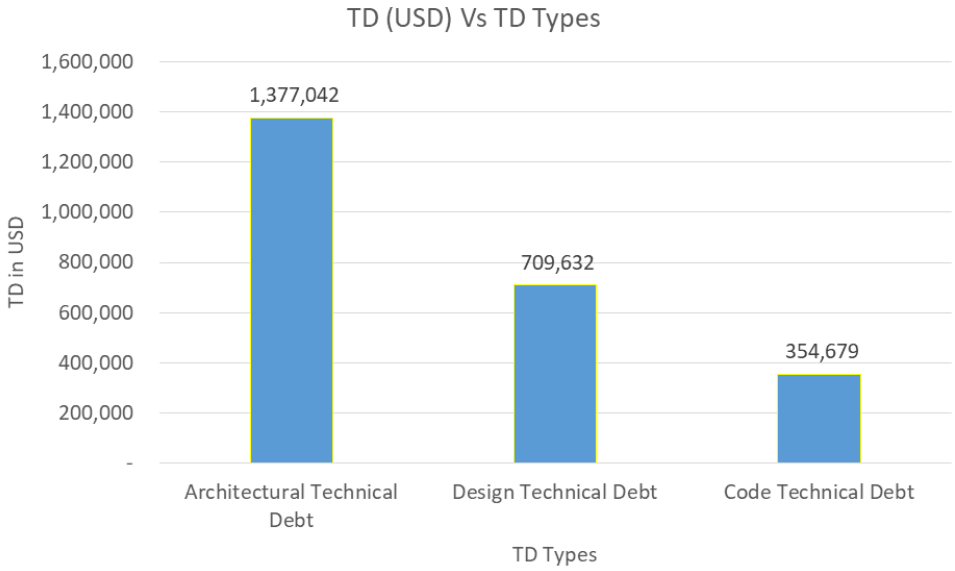


Figure 1: Distribution of TD types in the studied government software systems

These results are similar to what was reported by Lenarduzzi *et al.* [9], who also noted that, in general, ATD contribute more to the total debt present in software systems. However, this study has only examined the extent of TD present in public software systems at the time of our research; it does not show the evolution of each type of TD as the software system expands. Future studies should investigate how individual TD types evolve with software age. Importantly, these results suggest that either there is a lack of software systems architects in the studied public organizations or the available software systems architects are not utilized appropriately. We, thus, recommend that public organizations should perform skills gap analysis and provide appropriate software development trainings, in addition to utilizing the available professionals appro-

Electronic copy available at: https://ssrn.com/abstract=4040641

priately. This should reduce reckless TD from software systems.

Further, the study found 3,341,683 TD items caused by architectural, design, and coding flaws in the studied public sector software systems. The TD items identified spanned 14,253,122 LOC. Also, we found no significant variation of the extent of TD present in software systems across business domains. However, the TD present in software systems developed by vendors was found to be higher compared to TD in in-house developed software systems. We also observed variations in the cost of developing a new LOC across different in-house developed software. This could be caused by variations of experiences of software development teams in the studied public organizations: more experienced software development teams are expected to develop more LOC compared to less experienced teams in a given time, implying that the cost of developing a new LOC is low for more experienced teams compared to less experienced teams.

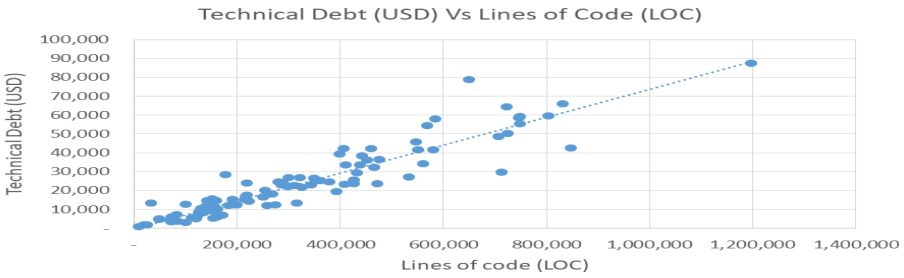
In addition, this study found that, in general, for each LOC in Tanzania public organizations' software systems, there is an average of USD 0.073 of TD. This means that the cost of refactoring a single LOC to an ideal state is nearly half of the cost of developing a new LOC in the studied public organizations. Although the average TD per LOC in Tanzania is smaller compared to TD per LOC in countries such as US, which is USD 3.61 according to 2017 CAST Report on Application Software Health (CRASH) [33], this comparison is not perfect because the two variables used to quantify TD, cost of labor per hour and effort measured in hours per LOC, vary in different contexts. However, it helps to give an overview of the magnitude of the TD problem. Public organizations must make sure that the cost of refactoring a single LOC does not exceed the cost of developing a new LOC.

4.2. Relationship between TD and internal software attributes

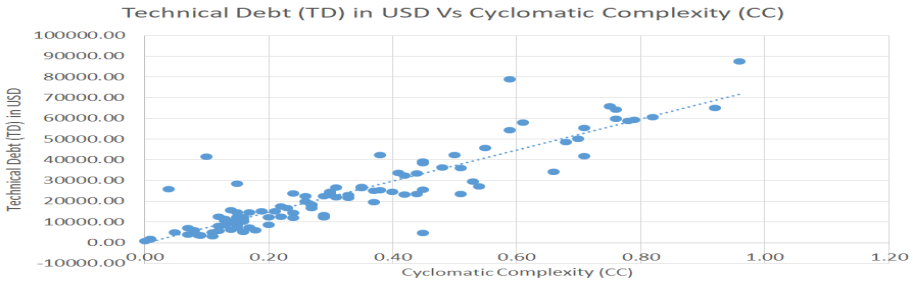
Figure 2(a) and Figure 2(b) respectively show the correlations between TD and LOC and TD and CC in Pearson Product Moment Correlation (PPMC). PPMC is a measure of the strength and direction of association that exists

Electronic copy available at: https://ssrn.com/abstract=4040641

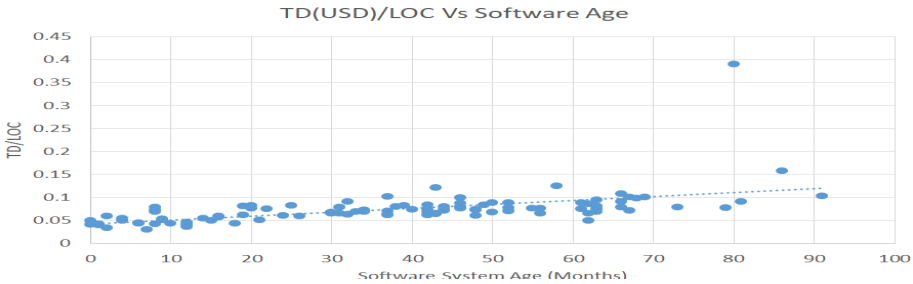
between two variables. The correlation between the ratio of TD to LOC and software system's age is presented in Figure 2(c).



(a) Correlation between TD and LOC



(b) Correlation between TD and CC



(c) Correlation between TD/LOC and software age

Figure 2: Plots of how TD varies with lines of code, cyclomatic complexity, and software age

A strong positive linear correlation between TD and LOC agrees with an intuition that, as LOC increase, more TD items are introduced and existing items accumulate TD through interest. According to Scott and Johnson [34], software systems designs decay as systems evolve. This explains the observed linear

Electronic copy available at: https://ssrn.com/abstract=4040641

correlation between TD and LOC. The sharp decrease of TD after a certain increase in LOC is also not counter-intuitive. Sharp decrease in TD indicates refactoring. Absence of sharp decrease of TD at a certain interval suggests the absence of refactoring in software systems. This raises a question as to why public organizations are not refactoring their software systems despite the large amount of TD present. A possible explanation for this is lack of visibility of TD items present in their software systems, making software practitioners in public organizations to be unaware of the extent of TD present in their software systems. We recommend that public organizations should use TD visualization tools, to enable visibility of TD items present in their software systems.

In addition, a strong positive linear correlation between TD and LOC implies that TD is spread all over the codebase. This means that TD items accumulated in government organizations software systems are mostly incurred unintentionally rather than being deliberate through trade-off decisions. Unintentional or accidental TD items are mostly incurred in development teams where there is a lack of visibility of software system's health as development progresses. Because of inability to visualize software system's health during development or maintenance, software teams become unable to prioritize source code that need refactoring. The strength of this correlation confirms that software developers in public organizations rarely work to improve the quality of existing codebase. In other words, public organizations have limited or no refactoring through strategies such as TD prioritization to manage TD present in software systems. This causes flaws to continue spreading from most expensive code (code with high interest rate) to less expensive code (code with low interest rate). This result also suggests lack of knowledge and awareness of TD phenomenon in the studied public organizations.

Furthermore, a positive linear correlation between TD and Cyclomatic Complexity is not counter-intuitive. As a software system expands, its internal components increase and interactions among the components increase as well. This makes a system harder to understand and maintain. According to Nguyen-Duc [35], software complexity refers to the extent to which a software system

Electronic copy available at: https://ssrn.com/abstract=4040641

is hard to comprehend, test, and modify. Once the software system becomes harder to understand, introduction of new features become very tedious and time-consuming. This is when quick and dirty implementation choices are made deliberately or accidentally in order to save time. As a result, system internal structure deteriorates. This could explain why the finding showed strong positive correlation between TD and CC. The result is supported by an empirical study by Jay *et al.* [30], which found a practical linear relationship between LOC and CC that holds across programming languages and paradigms. It is hereby recommended that public organizations should manage complexity in order to control TD present in their software systems.

As well, a linear correlation between TD/LOC and software age implies that there is no refactoring (debt repayment) strategy in place. In an environment where software refactoring is performed, TD present in software system does not rise linearly with software age. Data points would have scattered on either side of the correlation line to indicate whether the refactoring was done correctly or introduced more bugs. This correlation confirms the notion that absence of refactoring strategy causes rapid deterioration to breaking point for software systems in the studied public organizations. However, still the same question remains: why public organizations are not refactoring their software systems? Apart from the previous explanation, another reason could be lack of knowledge on TD concept among software practitioners in public organizations. Thus, it is hereby recommended that public organizations should train their software development teams and create awareness of the TD concept.

More specifically, the deviation of data points more on either side of correlation lines as software systems expand, as presented in Figure 2, can be explained as follows. Software systems expand for various reasons such as fixing bugs, adding new features or refactoring. These activities can increase or reduce code disorganization (entropy) and complexity, depending on the knowledge and experience of the developers performing the tasks. This explains why data points deviate either side of correlation lines as software systems evolve. The lower side deviation indicates refactoring in which adding new features or

Electronic copy available at: https://ssrn.com/abstract=4040641

fixing bugs was done correctly and helped to reduce TD. The upper side deviation indicates refactoring in which adding new features or fixing bugs was done incorrectly and caused introduction of more TD. The phenomenon matches with the assertion by Lehman and Belady [36]: software disorganization (entropy) and complexity increases as the software evolves, unless a specific work is done to maintain or reduce it. To control rapid software entropy rise, software practitioners in public organizations must prevent unnecessary increase in LOC and code duplication when adding new features or removing bugs. Object-oriented techniques, particularly volatility-based decomposition and code reuse, can be applied to overcome this challenge.

Deviation of data points from correlation lines could also be explained from a social dimension of software development. Large systems are usually developed by distributed teams. In such teams, complete understanding of the system is distributed across developers participating in the project, in such a way that no one has a holistic understanding of the software system under development. The rationale of the decisions made on developed artifacts are communicated. If communication between teams is not handled correctly, it introduces TD. Large software systems tend to become more complex as a social dimension of software development is added [29].

4.2.1. Coefficients of correlation between TD and internal software attributes

Based on what is presented in Figure 2, the correlation coefficients, r, of the relationships between TD and internal software attributes (LOC, CC, and software system age) were 0.93, 0.90, and 0.53 respectively. The correlation coefficient, r, indicates the strength and direction of a linear relationship. The higher the absolute value of r indicates the strength of the relationship, while a negative or positive value indicates the direction of the relationship.

The r value of 0.93 indicates that public organizations software systems incur TD at a very high rate on introduction of new lines of code. This could be caused by lack of proper communication between software development teams in public organizations, causing software practitioners to become unaware of

Electronic copy available at: https://ssrn.com/abstract=4040641

the rationale behind design decisions of certain artifacts. If design decisions are not communicated properly across development teams, new TD items could be introduced during addition of new features or refactoring. Treude and Storey [37] described the importance of knowledge sharing across teams during software development projects. Public organizations should enhance communication between their software development teams in order to avoid introduction of new TD items during software modification. This could also help avoid the effects of Cyclomatic Complexity and software aging.

4.2.2. Modelling the relationship between TD and internal software attributes

Table 4 shows the results of the multiple linear regression analysis model that relates TD with LOC, CC, and software age. The p-value for the overall F-test (Significance F) is less than 0.05, implying that the model is statistically significant and can be used to understand how TD is related to LOC, CC, and software age. R-squared is equal to 0.91, implying that the three independent variables, LOC, CC, and software age, can explain about 91% of the variation in the dependent variable (TD). The p-values of all the three variables, LOC, CC, and software age, are statistically significant at 99% confidence level.

4.3. Causes of TD in public sector software systems

We now present technical and social-technical causes of TD in the software systems of the studied public organizations.

4.3.1. Technical causes of TD accumulation in public sector software systems

Figure 3 shows the percentage contribution of each technical cause to the accumulation of TD in the studied software systems. The technical causes are presented from most to least influential.

The high contribution of modularity violation to the overall TD found (56.40%) suggests the presence of high module coupling with very low cohesion in the studied public organizations' software systems. This level of module coupling implies that software systems in public organizations are not well-designed, which

Electronic copy available at: https://ssrn.com/abstract=4040641

Table 4: Regression analysis of the relationship between TD and internal software attributes

Regression Statistics					
Multiple R	0.955170187				
R-squared	0.912350085				
Adjusted R-squared	0.909666925				
Standard Error	13095923.91				
Observations	102				
ANOVA					
	df	*SS*	*MS*	*F*	*Significance F*
Regression	3	1.75E+17	5.83E+16	340.0281	1.19E-51
Residual	98	1.68E+16	1.72E+14		
Total	101	1.92E+17			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*
Intercept	-13173794.59	2937261.489	-4.485060197	1.98E-05	-19002694.42
Age (months)	409716.6002	58054.07889	7.05749894	2.44E-10	294510.1701
CC	-48155593.6	18783231.15	-2.563754511	0.011874	-85430304.51
LOC	216.7166863	18.38391643	11.78838509	1.69E-20	180.2344024

Electronic copy available at: https://ssrn.com/abstract=4040641

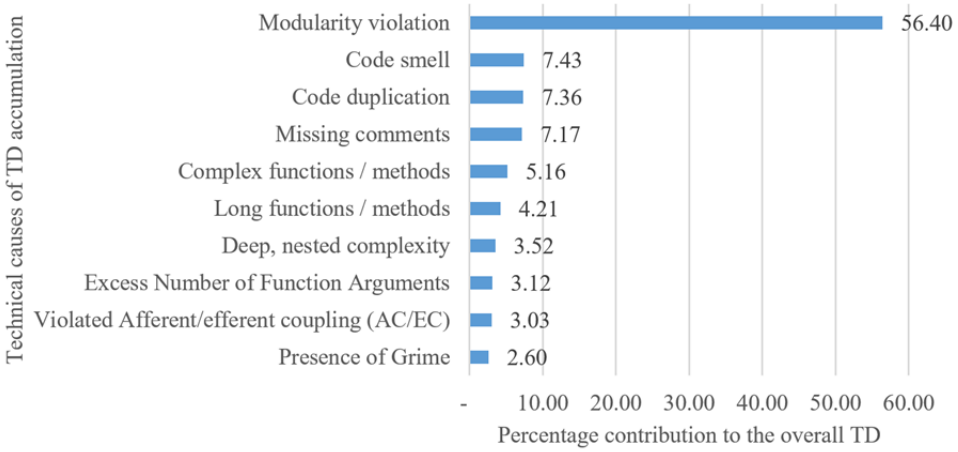


Figure 3: Top ten technical causes of TD accumulation

is likely to be caused by absence of specific expertise, particularly software architects and designers, in public organizations' software development teams. Modularity is a key principle in software design. For a software system to be easily maintained and evolved, its modules must be loosely coupled and highly cohesive. Tight coupling in software systems, with no clear decoupling strategy, implies the absence of visibility of structure and properties of software components. To address this problem, public organizations must put more emphasis on techniques and tools for producing modular software architectures and designs that can be easily maintained and evolved. The degree of other technical causes of TD suggests the need for continuous quality management of software systems in public organizations, using appropriate standards, guidelines and tools.

4.3.2. Social-technical causes of TD accumulation in public software systems

Figure 4 shows top ten social-technical causes of TD accumulation in software systems of the surveyed public organizations. Among these social-technical causes, four (4) are related to project management issues, two (2) are related to knowledge issues, three (3) are related to software development issues, and one (1) is related to organizational issues. The results indicate that, project

Electronic copy available at: https://ssrn.com/abstract=4040641

management issues and software development practices are two main sources of TD accumulation in the software systems of the studied public organizations.

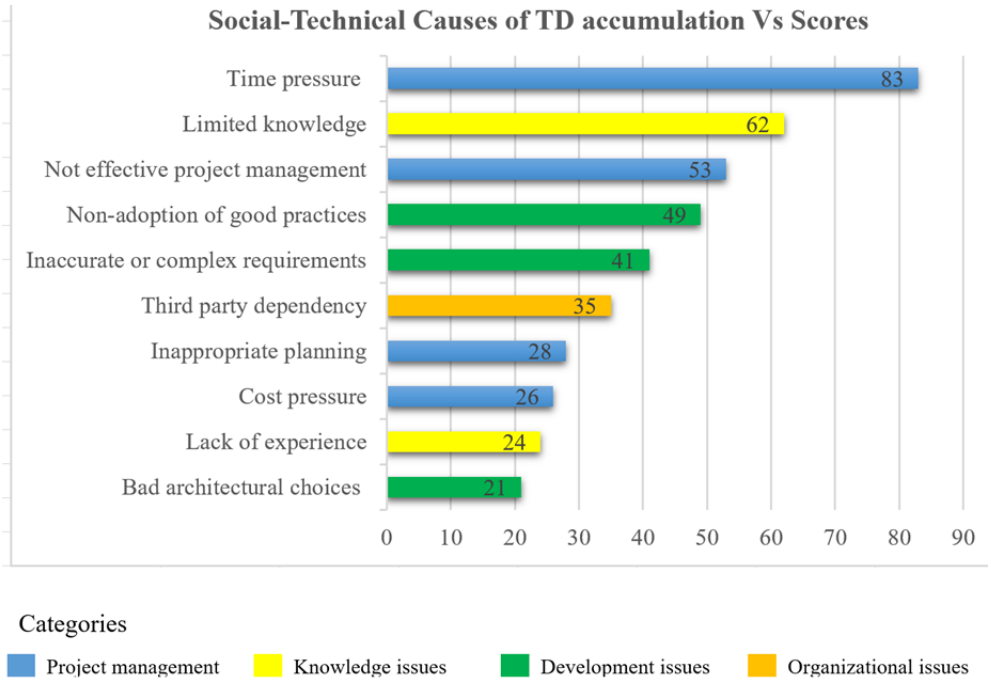


Figure 4: Social-technical causes of TD accumulation

Almost a half (45%) of the top ten most influential social-technical causes of TD accumulation in the software systems of the studied public organizations are related to management issues. Studies have reported the effects of project management on the quality of software systems. Software projects quality management is a key success factor for customer satisfaction [38]. Some specific issues related to project management include lack of qualified software systems project managers, and poor application of project management tools, techniques, and best practices. Poor project management is responsible for failure of almost 70% of technology projects worldwide [39]. Wiese *et al.* [40] proposed a framework for integrating project management with TD management, and demonstrated a reduction of TD in organizations by more than 60%. Software project managers

Electronic copy available at: https://ssrn.com/abstract=4040641

in public organizations should have good knowledge of using tools, techniques and best practices to successfully manage software projects.

Issues related to software development practices contribute more than a quarter (26.3%) of the top ten social-technical causes of TD accumulation in the software systems of the studied public organizations. This indicates that, some development teams in public organizations do not adhere to best practices of software development. Also, organizational and knowledge issues together contribute more than a quarter (28.7%) of top ten social-technical causes of TD accumulation in the software systems of the studied public organizations. This represents TD incurred deliberately due to organizational decisions and lack of knowledge. Deliberate TD is incurred when quality is traded for quick delivery, typical of many public organizations, in which, sometimes, servicing specific political interests is highly prioritized. These decisions are made at organizational level.

The fact that time pressure is the most influential social-technical cause of TD accumulation in the studied public organizations could be explained as follows. Most public organizations in developing countries still have manual business processes. The demand to automate their processes is too high. This creates time pressure for software development teams to respond fast to organizational needs by taking shortcuts during software development. By doing so, they violate principles of good software development and incur TD. Other researchers have also reported time pressure to be among the most influential social-technical causes of TD accumulation [41]. Proper project planning is key for public organizations to prevent incurring TD through time pressure.

Other social-technical causes of TD accumulation appearing in the top ten list revealed in the present study do not deviate significantly from previous studies, except third party dependency. Some public organizations still depend on external vendors to develop and maintain their software systems. This practice can be very costly if contractual issues are not handled correctly. And it could explain why most survey respondents rated third party dependency as one of the top ten social-technical causes of TD accumulation in the studied organiza-

Electronic copy available at: https://ssrn.com/abstract=4040641

tions. In general, software development capacity building is essential for public organizations to avoid third party dependency.

4.4. Relationship between Capability Maturity Level and the accumulation of TD

Of the thirteen (13) studied organizations, only two (2) had the characteristics of the third level (Defined) in CMMI, seven (7) matched the CMMI second level (Repeatable), and four (4) matched the CMMI first level (Initial). Table 5 shows the relationship between TD/LOC and Capability Maturity Level.

Table 5: Regression analysis of the relationship between TD and Capability Maturity Level

Response variable	**Explanatory variable**	**Estimate**	**t value**	**p value**	
TD/LOC	(Intercept)	150.96	5.813	7.35E-08	***
	Capability Maturity Level	13.12	1.149	0.253	
Multiple R-squared: 0.01304, Adjusted R-squared: 0.003169					
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1					

Because the p value is greater than 0.05, the results show no relationship between the level of software process maturity in public organizations and the amount of TD present in their software systems. A possible explanation for this phenomenon is that, in Tanzania, general government software systems (software systems that are used by all government organizations) are developed by the e-Government Authority (eGA)[1]. This means that organization-specific software processes maturity level is unlikely to dictate the extent of TD present in general government software systems.

4.4.1. Summary of results and recommendations

In general, Architectural TD was higher than Design TD and Code TD in the studied public sector software systems, and modularity violations and project management issues were found to be the leading causes of TD in software

[1] https://www.ega.go.tz/

Electronic copy available at: https://ssrn.com/abstract=4040641

systems of the studied public organizations. Based on the results of this study, we recommend the investigation of TD management strategies that are suitable for the public sector, taking the dictates of public organizations into account. Meanwhile, public organizations need deliberate efforts to prevent, identify and manage TD, including prioritization of TD refactoring.

5. Threats to Validity and Mitigation Strategies

As in any empirical study, there are a number of threats to the validity of the results of this study.

5.1. Internal Validity

Using specific indicators to identify TD items present in software systems can lead to inaccurate results due to indicator overlap. A single line of code can be counted in one or more TD items. To mitigate the possibility of inaccurate results caused by indicator overlap, each TD indicator was investigated separately. Moreover, TD identification tools have different underlying violation detection mechanisms and defined rules. These rules are dependent on the programming languages being investigated. Zazworka *et al.* [27] conducted a study to compare TD identification approaches. Their study reported only a few result overlaps when different tools were used. Using a specific tool could pose a threat to the validity of our results. To mitigate this threat, the study applied more than one tool to identify violation in the same codebase artifact. In case of deviations, average values were considered.

As well, using LOC to quantify TD present in software systems poses another threat to internal validity. The assumption that each LOC takes the same effort to debug is not ideal. This is because software systems differ in complexity. It is generally considered that, complex software systems are harder to design, develop and maintain compared to simple systems. The time required to develop a single LOC for a complex sorting algorithm can be significantly higher compared to the time required to develop a single LOC for saving student's

Electronic copy available at: https://ssrn.com/abstract=4040641

data in a school management system. Thus, to assume that LOC across public organizations' software systems require the same effort to develop or debug may not be ideal. Considering the fact that public organizations have a wide range of software systems with different inherent complexities, a threat to validity can rise. However, there is no proven method in the literature for estimating effort required to debug software systems. To mitigate the validity threat caused by the variation in complexities of software systems, our sample included software systems with different complexities.

5.2. External Validity

The use of convenience sampling to select the studied software systems could affect the generalizability of our results. Software systems in public organizations have different structures, sizes, complexities, and ages. Thus, using convenience sampling could affect our ability to obtain a representative sample of software systems in the Tanzanian public sector. To limit this threat, software systems were selected from different organizations, with different structures, sizes, complexities, and ages.

5.3. Construct Validity

This study assumed that, every detected violation of software development best practices is associated with TD. This poses a threat to construct validity because some violations may not be associated with TD. Ideally, for a violation to qualify as a TD, it must be associated with interest payment [42]. In other words, for a violation to be a TD, it must have interest probability greater than zero. This is because, in practice, not all flaws that occur during software development or maintenance will impede organizations in the future. To understand which flaws are likely to impede organizations during the lifetime of a software system, the concept of interest probability [43], which is beyond the scope of this study, can be used. Future studies can focus on only TD items with greater-than-zero interest probability.

Electronic copy available at: https://ssrn.com/abstract=4040641

5.4. Conclusion Validity

The validity of the conclusion that TD in government organizations software systems can be predicted from LOC, CC, and software age depends on the size of the dataset used for regression analysis. To mitigate this threat to conclusion validity, the study has investigated 102 government organizations software systems. This number of software systems investigated can be considered big enough to make valid conclusions.

6. Conclusion

Suboptimal design and implementation decisions made to save time during software development are formally known as Technical Debt. Studies have been conducted to establish the extent, causes and management of TD in software systems. However, most studies on TD have focused on private and open source software systems, lacking insights about TD in software systems in the public sector [5]. Software systems in the public sector are different from software systems owned by private organizations and open source systems, because public organizations are subjected to specific requirements and restrictions. This creates a possibility of differences in extent, causes, and management of TD between private organizations and public organizations, calling for the need to study TD identification, causes, and management of TD in public sector software systems.

Using 102 software systems from 13 organizations, this study investigated the extent and causes of TD in software systems in the Tanzanian public sector. Specifically, we set out to understand three things: the extent of TD in public sector software systems, the causes of TD in public sector software systems, and whether there is a relationship between the Capability Maturity Level of a public organization and the amount of TD. We found a substantial amount of TD in the studied software systems. Architectural Technical Debt was the most accumulated type of TD, followed by Design Technical Debt and Code Technical Debt. As well, modularity violations and project management issues

Electronic copy available at: https://ssrn.com/abstract=4040641

were found to be the leading causes of TD in software systems of the studied public organizations. We also found no relationship between organization maturity levels and TD accumulation in the studied public sector software systems. Based on the study results, we recommend the investigation of TD management strategies that are suitable for software systems in the public sector.

Apart from investigating strategies to manage TD in public sector software systems, future studies should attempt to consider the concept of TD interest probability [43] when identifying TD in the public sector, to shed more light on the extent and causes of TD in public sector software systems. Further, future research should consider other types of TD [14, 26], apart from those related to architecture, design, and code, as covered by the present study. Also, because this study did not investigate the evolution of each type of TD as the software system expands, future studies should investigate how individual TD types in public software systems evolve with software age.

Acknowledgement

Authors would like to acknowledge anonymous participants and Tanzanian public institutions that participated in this study.

Data

To support reproducibility studies, data for this study can be accessed using the following link: https://github.com/da196.

References

[1] A. Martini, J. Bosch, The danger of architectural technical debt: Contagious debt and vicious circles, in: 2015 12th Working IEEE/IFIP Conference on Software Architecture, IEEE, 2015, pp. 1–10.

[2] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, A framework to aid in decision making for technical debt management, in: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), IEEE, 2015, pp. 69–76.

Electronic copy available at: https://ssrn.com/abstract=4040641

[3] Z. Codabux, B. J. Williams, N. Niu, A quality assurance approach to technical debt, in: Proceedings of the International Conference on Software Engineering Research and Practice (SERP), The Steering Committee of The World Congress in Computer Science, Computer ..., 2014, p. 1.

[4] W. Cunningham, The wycash portfolio management system, ACM SIGPLAN OOPS Messenger 4 (2) (1992) 29–30.

[5] M. E. Nielsen, C. Ø. Madsen, M. F. Lungu, Technical debt management: A systematic literature review and research agenda for digital government, in: International Conference on Electronic Government, Springer, 2020, pp. 121–137.

[6] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), in: Dagstuhl Reports, Vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[7] A. Nugroho, J. Visser, T. Kuipers, An empirical model of technical debt and interest, in: Proceedings of the 2nd workshop on managing technical debt, 2011, pp. 1–8.

[8] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, T. Amanatidis, Estimating the breaking point for technical debt, in: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), IEEE, 2015, pp. 53–56.

[9] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, et al., An overview and comparison of technical debt measurement tools, IEEE Software.

[10] P. Kouros, T. Chaikalis, E.-M. Arvanitou, A. Chatzigeorgiou, A. Ampatzoglou, T. Amanatidis, Jcaliper: search-based technical debt management, in: Proceedings of the 34th ACM/SIGAPP Symposium on applied computing, 2019, pp. 1721–1730.

Electronic copy available at: https://ssrn.com/abstract=4040641

[11] C. Lilienthal, Sustainable Software Architecture: Analyze and Reduce Technical Debt, dpunkt. verlag, 2019.

[12] P. Ciancarini, D. Russo, The strategic technical debt management model: An empirical proposal, Open Source Systems 582 (2020) 131.

[13] R. Alfayez, P. Behnamghader, K. Srisopha, B. Boehm, An exploratory study on the influence of developers in technical debt, in: Proceedings of the 2018 international conference on technical debt, 2018, pp. 1–10.

[14] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, Journal of Systems and Software 101 (2015) 193–220.

[15] M. BenIdris, Investigate, identify and estimate the technical debt: a systematic mapping study, Available at SSRN 3606172.

[16] T. Besker, A. Martini, J. Bosch, Managing architectural technical debt: A unified model and systematic literature review, Journal of Systems and Software 135 (2018) 1–16.

[17] C. Becker, R. Chitchyan, S. Betz, C. McCord, Trade-off decisions across time in technical debt management: a systematic literature review, in: Proceedings of the 2018 International Conference on Technical Debt, 2018, pp. 85–94.

[18] L. F. Ribeiro, M. A. de Freitas Farias, M. G. Mendonça, R. O. Spínola, Decision criteria for the payment of technical debt in software projects: A systematic mapping study., in: ICEIS (1), 2016, pp. 572–579.

[19] W. N. Behutiye, P. Rodríguez, M. Oivo, A. Tosun, Analyzing the concept of technical debt in the context of agile software development: A systematic literature review, Information and Software Technology 82 (2017) 139–158.

[20] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, The financial aspect of managing technical debt: A systematic literature review, Information and Software Technology 64 (2015) 52–73.

Electronic copy available at: https://ssrn.com/abstract=4040641

[21] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, C. Seaman, Identification and management of technical debt: A systematic mapping study, Information and Software Technology 70 (2016) 100–121.

[22] T. Besker, A. Martini, R. E. Lokuge, K. Blincoe, J. Bosch, Embracing technical debt, from a startup company perspective, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 415–425.

[23] A. Martini, J. Bosch, M. Chaudron, Architecture technical debt: Understanding causes and a qualitative model, in: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE, 2014, pp. 85–92.

[24] J. Holvitie, S. A. Licorish, R. O. Spínola, S. Hyrynsalmi, S. G. MacDonell, T. S. Mendes, J. Buchan, V. Leppänen, Technical debt and agile software development practices and processes: An industry practitioner survey, Information and Software Technology 96 (2018) 141–160.

[25] L. Anthopoulos, C. G. Reddick, I. Giannakidou, N. Mavridis, Why e-government projects fail? an analysis of the healthcare. gov website, Government Information Quarterly 33 (1) (2016) 161–173.

[26] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, F. A. Fontana, Technical debt prioritization: State of the art. a systematic literature review, arXiv preprint arXiv:1904.12538.

[27] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, Comparing four approaches for technical debt identification, Software Quality Journal 22 (3) (2014) 403–426.

[28] M. Lanza, R. Marinescu, Identity Disharmonies, Springer, 2006.

[29] A. Tornhill, Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis, Pragmatic Bookshelf, 2018.

Electronic copy available at: https://ssrn.com/abstract=4040641

[30] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, W. Charles, et al., Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship, Journal of Software Engineering and Applications 2 (03) (2009) 137.

[31] D. Zubrow, W. Hayes, J. Siegel, D. Goldenson, Maturity questionnaire, Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST (1994).

[32] A. Martini, F. A. Fontana, A. Biaggi, R. Roveda, Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company, in: European Conference on Software Architecture, Springer, 2018, pp. 320–335.

[33] Newzoo, Global esports market report: Trends, revenue, and audience toward 2020, http://strivesponsorship.com/wp-content/uploads/2017/02/Newzoo_Free_2017_Global_Esports_Market_Report.pdf (2017).

[34] H. Scott, P. M. Johnson, Generalizing fault contents from a few classes, in: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE, 2007, pp. 205–214.

[35] A. Nguyen-Duc, The impact of software complexity on cost and quality-a comparative analysis between open source and proprietary software, Journal of Software Engineering and Applications 8 (2) (2017) 17–31.

[36] M. M. Lehman, L. A. Belady, Program evolution: processes of software change, Academic Press Professional, Inc., 1985.

[37] C. Treude, M.-A. Storey, Effective communication of software development knowledge through community portals, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 91–101.

Electronic copy available at: https://ssrn.com/abstract=4040641

[38] K. Jamsutkar, V. Patil, P. Chawan, Software project quality management, International Journal of Engineering Research and Applications (IJERA) ISSN (2012) 2248–9622.

[39] S. Aidane, The "chaos report" myth busters, Guerrilla Project Management-Research.

[40] M. Wiese, M. Riebisch, J. Schwarze, Preventing technical debt by technical debt aware project management, in: Proceedings of the 2021 IEEE/ACM International Conference on Technical Debt, 2021, pp. 91–101.

[41] R. Ramač, V. Mandić, N. Taušan, N. Rios, M. G. de Mendonca Neto, C. Seaman, R. O. Spínola, Common causes and effects of technical debt in serbian it: Insightd survey replication, in: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2020, pp. 354–361.

[42] G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, O. Matei, R. Heb, The risk of generating technical debt interest: a case study, SN Computer Science 2 (1) (2021) 1–12.

[43] A. Ampatzoglou, A. Michailidis, C. Sarikyriakidis, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, A framework for managing interest in technical debt: an industrial validation, in: Proceedings of the 2018 International Conference on Technical Debt, 2018, pp. 115–124.

Electronic copy available at: https://ssrn.com/abstract=4040641