University of Twente

MSc. Business Information Technology

Master Thesis

Balancing software maintenance effort as a technical debt management strategy

the design and implementation of a novel algorithm

Author:
N. Gijsen

Supervisor:
Dr. A.I. Aldea

Second supervisor:
Dr. M. Daneva

September 28, 2020

Contents

List of Figures

List of Tables

Abstract

With the average lifetime of software systems increasing rapidly, the process of software maintenance becomes a ever more important part of the product life-cycle. Something that is inherent to software maintenance is technical debt, many find this an intangible concept which is difficult to manage. Technical debt can accumulate quickly when neglected, which has a deterrent effect on productivity, making it even harder to reduce the debt in the first place. In this research we propose a method which enables managers to make strategic resource allocation decisions, to keep software at an optimum debt level.

We started by conducting a systematic literature review into the concepts of software maintenance, technical debt and productivity. We found theoretical evidence that TD can be manipulated by adjusting the software maintenance effort allocation. Literature suggested that by reducing technical debt, the productivity of developers could improve.

Based on this literature review, we constructed a process-model which incorporates all the components necessary to manage technical debt by allocating resources. We defined measures for each component that are not software specific so the method can be easily implemented in multiple different projects and organizations. Thereafter we created an artifact which is build on this process-model with the goal of constructing a tool which can be used in practice. We thoroughly documented the design part of this artifact, explaining the design choices made by the researcher.

In the final stage of this research, we build the actual algorithm and implemented it in a medium size IT company. While only being a proof of concept version, the preliminary results are very promising. The evidence suggest that technical debt management strategies can have a large influence on average productivity when considering longer horizons. This means our method can save costs, time and improve productivity with the same amount of effort.

Preface

This thesis marks the end of my personal journey at the University of Twente. Where I started out, 8 years ago, as a bachelors student in business administration, with no clue of what I liked. After four years I decided to follow my passion and choose to enroll in the business information technology master. Overall my time at the University was an incredible experience where I met so many amazing people. I spent almost 12 months to complete this thesis, with many people supporting me along the way. I want to thank everyone, and a few individuals in particular.

First of all, my fist supervisor Adina Aldea. Who was always ready to help or open for discussion. Secondly I like to thank Maya Daneva, who took the time and effort to give me feedback when facing a stressful period.

I also want to thank Topicus for giving me the opportunity to conduct this research and giving me the freedom to take my own route. With Stefan Hessels in particular, which not only helped me in my research but also made my time at Topicus a very enjoyable one, one I am not likely to forget.

Lastly I want to thank my parents, for supporting me in every possible way throughout my studies.

Chapter 1

Introduction

In this chapter we first introduce the core concepts before we define the research problem. Followed by section 1.2 in which we define the main research question, supported by multiple supporting research questions.

1.1 Problem statement

Maintenance

Developing a software system is only a single component of software engineering, after a system went into production, the systems has to be maintained. All these activities that keep an existing piece of software into production are considered software maintenance. The field of software maintenance research already exists since the 1970's. H. Mills [36] was one of the first scholars to mention the need for software maintenance and the challenges involved. While technology has radically changed since the introduction, the necessity of software maintenance remains. The goal is to sustain the product during its life cycle and continue to satisfy the requirements of the users [34]. This is achieved by modifying existing software, while maintaining the integrity of the product. Modifications can be made for a number of reasons, e.g. to patch a vulnerability or to implement a new feature. Software maintenance has a lot in common with software development, but the biggest difference is their place in the product's life cycle. Software development is the primary activity before a product is introduced, while after introduction, most activities are considered software maintenance, including the development of new features. Software engineering literature puts a lot of emphasis on the development process, while maintenance consumes a majority of time and financial resources during the product's life cycle. This makes software maintenance an interesting field to research, as there is still enough room for improvements. Maintenance activities create value both on the short and long term for different stakeholders. It is a common misconception that maintenance primarily consists of corrective work such as fixing

bugs. Studies have shown that over 50% of software maintenance is spend on non-corrective tasks [37].

While initial product development is often project based, with a set timeline and budget, software maintenance is continuous and will go on until the product is retired. Longer service life of a product can increase the total revenue. However, by extending the life cycle, the product becomes harder to maintain. Some efforts have been made in creating frameworks to manage software maintenance. Take for example maintenance maturity models [7]. These give a good understanding of the current maintenance process, but their drawback is that they lack clearly defined practices or decision models. Another trend in managing software maintenance is the increased popularity of the Technical Debt metaphor (TD). Recent studies have shown that TD highly impacts the total cost of ownership, as the accumulation this debt makes software harder to maintain [29].

Technical debt

TD is different from regular debt, as there is no financial obligation to an entity. Debt in the context of software engineering was first described by W. Cunningham [17]. He used the term to describe the possible negative effects of immature source code. Delivering unfinished or low quality code is alike to going in debt, it can help speed up development and as long as debt is repaid, there is no immediate problem. Too much debt can be dangerous, as all time spent on "not-quite-right" code count as interest on that debt. The accumulation of TD can seriously impede maintenance work, such as simple code changes that become more time-intensive due to previously poor design choices.

While TD impacts the total cost of ownership, TD is often not tracked or managed properly. In order to manage TD, it is necessary to quantify TD. Multiple tools exist for this purpose and they tend to use source code analysis to detect TD items. However, tools can not detect all potential types of TD such as architectural debt or technological gaps [29]. Multiple quantification methods have been proposed in literature [40] [51] [23]. In general, debt can be divided into two parts: Principal and Interest. Principal is the effort required to bring the system to the desired quality level. Interest is the additional maintenance work required as result of the existing principal. Empirical evidence suggest a correlation between the amount of TD and the number of bug fixes and other corrective tasks needed [6].

Multiple studies have built upon the concept of TD, by creating sub-types of TD for specific types of debt in the field of software engineering, such as test debt, architectural debt, requirement debt and documentation debt[29]. Technical debt can also be introduced on purpose, so called self-admitted technical debt. This can be done for multiple reasons, such as a shorter time to market. A study on self-admitted TD by Potdar & Shihab [44] found that self-admitted TD is not always removed after introduction. Furthermore, they found senior developers to introduce the most self-admitted TD.

A major cause of TD is schedule pressure, as compromises on quality are

made to meet deadlines. Agile development can help to reimburse debt shortly after it is incurred, however, the opposite often occurs [29]. Without proper retrospects, TD items are not added to the backlog and will be quickly forgotten, resulting in a quick build up of TD in a short period [29]. Furthermore, developers reported that they are frequently forced to introduce additional TD due to existing TD [11].

Implications of technical debt in software maintenance

When allocating resources within maintenance teams, managers often prefer the development of visible artifacts over reducing TD [29]. This is explainable, as delivering new features has a visible effect on the product. TD reduction however, is only visible for developers working on the product. This behavior might work in the short term by generating value for the users. Nevertheless it will likely not in the long term. When we consider the expected life time of a product, the reduction of TD can often be seen as a far better investment than the adding of a new feature. Recent studies suggest that by using better resource allocation policies within software maintenance, more business value can be created with the same amount of resources [23] [31]. These studies simulated different TD strategies in a fictional software project. The results are very promising, although they lack empirical evidence.

Taking TD into consideration during resource allocation could have a serious impact on the long term performance of a software product. In practice we see the life cycle of products being extended longer than before [23]. This makes the technical debt management (TDM) arguably even more important than it used to be, as increased life cycles bring new opportunities and challenges to software vendors.

Current TDM strategies focus on prioritizing individual items of TD. This is often based on some form of cost-benefit analysis [48]. TDM decision models exist to help mangers decide to fix an TD item now, or repay it in the future. While this can be highly effective at lowering the total maintenance costs, it can be very time consuming, especially when considering larger software projects. This makes it unsuitable for higher level management. A TD based resource allocation model would help managers make the right decisions in regard to TD.

Problem definition

Technical debt is something that is impossible to avoid and inherent to software maintenance. It is a very intangible concept and difficult to manage. This is a serious problem, as TD can accumulate quickly when neglected, with falling productivity as a result, making it even harder to repay the debt in the fist place. We want to investigate if we can solve this problem by strategically allocating resources in the maintenance process.

1.2 Research questions

The potential impact of software maintenance management and technical debt on the creation of business value, combined with the lack of research available, illustrates the importance of this study. As the concept of business value is quite vague, we choose to investigate the impact of TD on the team's productivity. Measuring productivity has the advantage that it is a relative performance measurement. This makes it easier for practitioners to compare the performance of different teams.

We aim to create a strategic method for managing maintenance projects in an agile environments. It should approximate the "sweet-spot" of balancing effort on different maintenance tasks, to maximize productivity by reducing time wasted due to TD.

This leads us to the following main question:

How to balance the software maintenance effort in order to maximize productivity by managing technical debt?

First step in approaching the central research question, is to investigate relevant literature in order to build a sound theoretical foundation of the related constructs. Subsequently we create a theoretical framework and finally build an algorithm which can be used in practice. This approach translates to the following research questions:

RQ 1: What is the state-of-art literature regarding software maintenance?

In order to balance the software maintenance effort, we first have to understand the concept of software maintenance. The goal of this research question is to find models or methods to classify the maintenance effort. While maintenance consists of a vast variety of activities, a classification would help group similar activities and make comparisons.

RQ 2: What is the state-of-art literature regarding technical debt?

This question serves the purpose of increasing our understanding on managing technical debt. Firstly we want to investigate the concept of TD, by considering what defines TD and how TD emerges/develops during projects. This also includes different types of TD as these differences could be relevant to the main question. The second goal is to find methods of identifying and measures to quantify TD.

RQ 3: What is the state-of-art literature regarding productivity?

Productivity can be considered the dependent variable in the main research question. Therefore we look towards literature to find ways of measuring productivity in the context of software maintenance. It is important to consider the context of productivity in software maintenance, as the classical definition as used in economics is not sufficient. This is because developers do not produce any physical products, but write new code or maintain existing pieces of code.

RQ 4: What is the effect of software maintenance effort on technical debt?

Products which are in the maintenance stage of the software life cycle, are continuously undergoing changes. We can expect these changes to have an impact on the technical debt of the product. One of the goals is to find support for the existence of this relationship. Secondly, we want to relate specific types of changes or maintenance tasks to TD. For instance, would a preventive maintenance task result in a reduction of TD? These findings would enable us to build a theoretical framework of TD in the context of software maintenance.

RQ 5: What is the effect of technical debt on productivity?

In order to balance the software maintenance effort for optimal productivity, it is important to research the effect TD has on productivity. We are interested in both the direct and indirect effects of TD on productivity. Furthermore, we aim to expand our knowledge of the contextual factors regarding this TD and productivity, for instance, short- versus long-term effects.

RQ6: How can we model and measure the effects of software maintenance strategies on technical debt and productivity?

We want to create a theoretical model by combining the knowledge gained from the previous research questions. This model will help us better understand the dynamics of TD by presenting it in a more abstract and simplified way. Measuring the components of the model is necessary for the further development of an algorithm.

RQ7: How to design a algorithm that approximates the optimal software maintenance strategy for a given project?

While a theoretical model is very useful to help understand the concept of TD in the context of software maintenance, it is less suitable to use in the real world. In order to demonstrate the potential value of balancing the software maintenance effort, we build an algorithm that can use real world project data as input. Based on this input the algorithm should be able to calculate an optimal software maintenance strategy.

Chapter 2

Research methodology

In this chapter we discuss how the research is performed and what methods were used. We first discuss the research design that we follow. After which we elaborate on all of the research methods used.

2.1 Research design

2.1.1 Design science

We chose to use the Design Science Research Methodology(DSRM) as guidelines for our research design. DSRM is an appropriate research method in this case, as we want to design an artifact that solves an organizational problem. We use the guidelines presented by Peffers [42]. DSRM offers us a process model to use as a mental model when conducting design science. It is an iterative process where the research goal is to develop an artifact for the aforementioned research problem.

Activity 1: Problem identification and motivation

This includes the specification of the research problem, as this specification will be the basis of the requirement of the artifact. Furthermore, this includes the justification of the solutions value. The results of this activity are reported in the Introduction.

Activity 2: Define the objectives for a solution

Here we define the objectives for the artifact based on the problem definition. These are realistic objectives based on what is possible and feasible. Moreover, this is the theoretic foundation on which we design and develop the artifact in a later stage. Research questions 1-5 are used to structure this activity, which are elaborated in Chapter 3.

Activity 3: Design and development

This includes determining the architecture of the artifact and its functionality, after which we create the actual artifact. Artifacts can range anywhere from being simple and abstract to being complex and highly detailed. We formulated research questions 6 and 7 for this purpose, which are discussed in Chapter 4.

Activity 4: Demonstration

The goal of this activity is to demonstrate that the artifact is able solve one or more instances of the problem. This is often done in a environment that is a subset or a representation of the intended context. Examples of this are case studies or simulations. The artifact of this research is demonstrated in Chapter 5.

Activity 5: Evaluation

This activity aims to observe and measure the artifacts effectiveness for solving the problem. Observed results from activity 4 are compared to the objectives of a solution from activity 2. At the end of the evaluation, researchers can iterate back to activity 3 in order to improve the artifact or continue on to the next activity. Chapter 6 is where we evaluate this artifact.

Activity 6: Communication

In the form of a research paper, with the goal to communicate the problem and its importance. How the artifact solves this problem, the novelty of the artifact, its effectiveness and the rigor of its design.

2.2 Research methods

2.2.1 Literature review

The method used in the review is based on the work of Kitchenham [27]. It is a well established method of conducting systematic literature reviews (SLR) in the field of software engineering. The goal of this review is to create an overview of existing empirical evidence on the specific research questions in an unbiased manner. The SLR consists of three stages: Planning, Conducting and Reporting the review. Figure 2.1 gives an overview of these stages, their components and the corresponding chapters in which they are reported.

Data sources and search strategy

Preliminary searches in multiple databases (IEEE, Science Direct, Scopus, ACM Digital and Springer link) show similar results. Scopus apears to be the most complete database for our research, as it delivers the most search results, including those of the other databases. This is because Scopus indexes many other

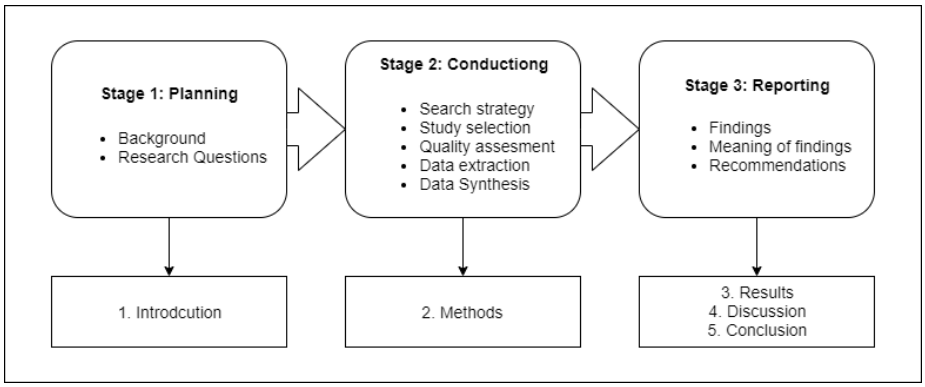

Figure 2.1: SLR components and the corresponding chapters of the report

databases. Together with its user friendliness, Scopus is the search engine of choice for this review.

Scopus allows us to construct sophisticated search strings using Boolean "AND" and "OR" operators. In some cases we use the "*" symbol to cope with potential alternative spellings for the same concepts. For example, sometimes Technical Debt is only used in plural, Technical Debts. Using the search component: "Technical Debt*" includes both the singular and plural form of TD in the search results. In the construction of the search strings, we also consider synonyms of certain constructs. We create a separate search string for each research question. We do this in order to keep the results more manageable.

The search strings are:

- RQ1:("software maintenance" OR "Software evolution") AND ("maintenance activit*" OR "maintenance task*" OR "maintenance typ*" OR "maintenance classification*")
- RQ2: "technical debt*" AND "software" AND ("maintenance" OR "evolution")
- RQ3: "productivity" AND ("metric*" OR "measur*) AND "software" AND ("maintainability" OR "evolution")
- RQ4: ("software maintenance" OR "software evolution") AND "technical debt*"
- RQ5: "technical debt*" AND "productivity"

In some cases the search strings and selection criteria can result in some important studies missing from the selection. These are often older pieces of original research which are highly relevant to the field. If we think this is the case, we use the reference list from the selected studies to manually add these original works to the selection. This specific method is also known as backwards snowballing [60].

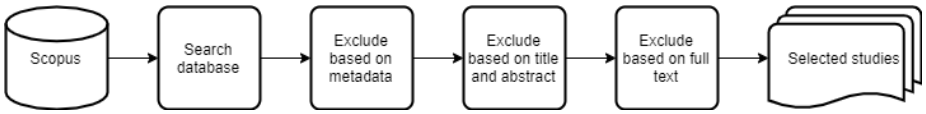

Figure 2.2: Selection process

Study selection

The study selection stage is a multistage process, a schematic overview can be seen in figure 2.2.

Before starting the selection process, it is necessary to define study selection criteria. These criteria have the goal to identify only those studies which provide evidence about the research question. By defining these criteria beforehand, the likelihood of bias is minimal. These selection criteria are referred to as inclusion and exclusion criteria [27].

The first round of exclusions is done by judging the reference properties of the found studies. One of these properties is the publishing year. For two reasons we exclude all studies published before 2010 from the review: Firstly, we are mostly interested in the state-of-art literature of the three main constructs of this study. Contributions made before 2010 are likely to be outdated and therefore not relevant. One drawback of this decision is the possibility of excluding studies which are older but still considered highly relevant. We mitigate this by using the snowballing technique as discussed in the previous section. Secondly, this study is subjected to certain time limitations and including all publishing years would consume considerable more time. Apart from excluding all studies published before 2010, we also exclude studies in other languages than English.

The goal is to include only those studies which are relevant to the research questions. A study is considered relevant when it provides evidence to (partially) answer the research question. During the next stage of selection, studies are included or excluded based on their title and abstract. As it is difficult to determine if a study contains enough useful evidence based on only the abstract and title, relevance is interpreted quite liberally during this stage.

In the final stage of the study selection processes, the remaining studies are read in full. Once again the papers are included or excluded based on their relevance to the research question. After this, the selected studies move on to quality assessment.

Study quality assessment

In addition to applying the inclusion and exclusion criteria, it is also important to assess the quality of individual studies [27]. While the quality of a study can be used as a more detailed inclusion/exclusion criteria itself, we use it primarily as a guide to interpret the results. This is especially useful if we encounter mixed findings, as those could possibly be explained by the quality difference of the studies in question. Furthermore, the quality can be used as a means

Level	Description
0	Evidence obtained from a systematic review
1	Evidence obtained from at least one properly-designed randomised controlled trials
2	Evidence obtained from well-designed pseudo-randomised controlled trials (i.e. non- random allocation to treatment)
3-1	Evidence obtained from comparative studies with concurrent controls and allocation not randomised, cohort studies, case-control studies or interrupted time series with a control group.
3-2	Evidence obtained from comparative studies with historical control, two or more single arm studies, or interrupted time series without a parallel control group
4-1	Evidence obtained from a randomised experiment performed in an artificial setting
4-2	Evidence obtained from case series, either post-test or pre-test/post-test
4-3	Evidence obtained from a quasi-random experiment performed in an artificial setting
5	Evidence obtained from expert opinion based on theory or consensus

Table 2.1: Study design hierarchy for Software Engineering by Kitchenham [27]

of weighting the importance of the selected studies. Individual study quality can be hard to measure, as there is no agreed definition of study "quality" [27]. We use the study design hierarchy for Software Engineering proposed by Kitchenham [27], for assessing the level of evidence (Table 2.2.1). This hierarchy ranks studies based on their research design. Systematic literature reviews are considered the highest level of evidence.

Data extraction and synthesis

In some cases, the contents of studies are almost identical, for instance when conference proceedings are later published as journal articles. As multiple studies based on the same data would bias the results, if possible, we include journal articles over conference proceedings. If not, we include the study with the most recent publishing date. The finalized list of selected papers is then exported to a reference manager. We look at the results as a whole, with the goal to find trends within the specific research community. Data used for this analysis is exported from the digital database and includes: title, publication date, publication source, article type and key words. These findings are reported in Chapter 3.

Data synthesis is the process of collating and summarizing the results of the included studies [27]. As the total number of selected studies per research question is relatively small, synthesis is done descriptively (non-quantitative). The researcher analyses the selected studies and compares the findings among each research question related group of individual studies.

2.2.2 Action research

As the demonstration and evaluation activities in the DSRM by Peffers [42] are quite vague, we want to perform these activities with an additional research method. For selecting a suitable one in conjunction with DSRM we used the framework by Venable [56]. In his work, he compares multiple research methods in the context of DSR. The framework aids in selecting a suitable method for a given research project.

Venable [56] distinguishes 4 types of validation methods, based on a 2x2 matrix, as seen in figure 2.3. One dimension represents the environment in which the method is introduced. This can be done in a naturalistic or an artificial setting. The second dimension is ex ante and ex post evaluation. Ex post refers to evaluation of an instantiated artifact, while ex ante refers to an un-instantiated artifact, such as a new design.

The framework guides us in selecting a particular DSR evaluation strategy, based on the project context, goals and limitations. This is done in 4 steps:

1. Analyze the requirements for the evaluation.
2. Map requirements to the evaluation matrix of Venable [56].
3. Select an appropriate evaluation method.
4. Design the Evaluation in more detail.

The primary goal of this evaluation is to determine the efficacy and quality of the algorithm. Although the artifact is technical in its core, we still need to evaluate it as a socio-technical one. This is because the artifact interacts with organizational factors. As the artifact is still a prototype, speed and low risk are key requirements for the method. By mapping these requirements on the matrix, we can already exclude ex post research methods. We choose a naturalistic environment over an artificial one, so that we can include the social elements of the artifact and evaluate the effectiveness better.

Based on the framework, action research is one of the recommended research methods. We prefer this over focus groups, as it enables us to introduce a treatment in a single case under conditions of practice.

For this method we follow the methodology proposed by Wieringa [58], as it is designed to validate information systems in design science. We follow his process model from his book [59] on design science as a guide. The model can be found in figure 2.4. While we discuss them separately, activities from the empirical cycle and client engineering cycle can be preformed concurrently.

Step 1: Research problem analysis

During this step, the researcher determines what conceptual framework is to be validated, what validation questions are suitable and how to define the population of the TAR. The conceptual framework to be validated is, in our case, the artifact developed in Chapter 5, with some alterations based to fit the client's

DSR Evaluation Method Selection Framework	Ex Ante	Ex Post
Naturalistic	•Action Research •Focus Group	•Action Research •Case Study •Focus Group •Participant Observation •Ethnography •Phenomenology •Survey (qualitative or quantitative)
Artificial	•Mathematical or Logical Proof •Criteria-Based Evaluation •Lab Experiment •Computer Simulation	•Mathematical or Logical Proof •Lab Experiment •Role Playing Simulation •Computer Simulation •Field Experiment

Figure 2.3: DSR Evaluation Framework by Venable [56]

problem context. The client can be seen as the population in this case. We use the following validation questions: What are the effects by the interaction between artifact and context? Does the presented artifact satisfy the requirements?

Step 2: Research & inference design

The measurements are defined during this step. It is important to chose a limited set of relevant measurements because there are an infinite number of aspects that could be measured using TAR [58]. The inference design is used to improve validity, by defining the way of reasoning beforehand. We use a combination of both descriptive and explanatory inference design. Descriptive inference is used to demonstrate the effects of the artifact in the context, while explanatory inference is used to report on unexpected outcomes.

Step 3: Problem investigation

The problem investigation is part of the client helper cycle, this step has the goal of defining the problem of the client. Here we identify the organization's goals, and what problematic phenomena is occur which hinders these aforementioned goals. Furthermore, we identify all relevant stakeholders who are considered part of the problem context.

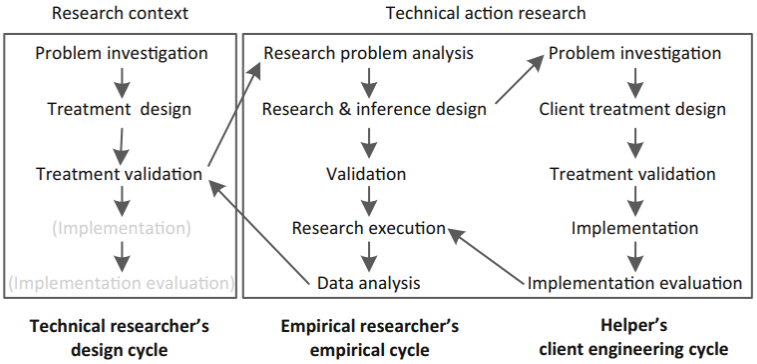

Figure 2.4: Structure of TAR by Wieringa[59]

Step 4: Client treatment design

Together with the client the researcher agrees on a treatment plan based on specific requirements. It is important that this satisfies both the business goals of the client, as well as the research goals of the researcher.

Step 5: Treatment validation

The treatment validation ensures that the actual treatment, together with the research design, allows the validation questions can be answered.

Step 6: Implementation

This is where we actually implement the artifact in the client's organization. First, the artifact is adjusted to work in conjunction with the clients context. After which it can be executed to observe the actual effects.

Step 7: Implementation evaluation

As a last step of the client cycle, we evaluate the outcome with the client. These are initial outcomes based on the specific implementation at the client. These could actually differ from the outcomes of the final report, because the client and researcher can have different goals.

Step 8: Research execution

We switch back to the researchers' perspective for this step. The researcher reports on the client implementation, which will be analyzed in the final step of TAR.

Step 9: Data analysis

In the final step we analyze the data by applying the inferences we designed in step 2. First, we provide descriptive data about the implementation, followed by explanations about the observations made. Using this knowledge we answer the validity questions, completing the evaluation activity of DSRM.

Chapter 3

Literature review

The search strategy as discussed in Chapter 2 resulted in a total of 1219 studies. The multi staged selection process resulted in a final selection of 32 studies, a more detailed overview of these results is presented in table 3.1.

The following observations relate to table 3.1. The first research question has the highest number of raw results. This can be easily explained, as the first search session is about software maintenance in general, while all other research questions are about a more specific sub space of the field. Exclusion by year and language show also an interesting pattern. Only a few papers were excluded based on language (less than 1% of the total amount). This means that most papers excluded in this stage are excluded by publishing year. The first and third question are quite similar in terms of exclusions, with 41% and 47% of papers being excluded by year respectively. This indicates that these research topics are quite mature as a large portion of the work is older than 10 years old.

More interesting are the results of the research questions regarding technical debt (question 2, 4 and 5). Only 7 studies were excluded from the second question because of being published before 2010. This highlights the newness of the concept of TD in the context of software maintenance. Furthermore, not a single papers was excluded by publishing year in the results of research question 4 and 5. All studies were namely published after 2010.

Selection stage	Research question					Total
	1	2	3	4	5	
Unfiltered results	835	128	130	95	31	1219
After exclusion by language & year	344	121	62	95	31	653
After exclusion by title & abstract	40	19	11	19	12	101
After full review	6	9	5	5	5	30
Snowballing	2	-	-	-	-	2
Final selection	8	9	5	5	5	32

Table 3.1: Search results

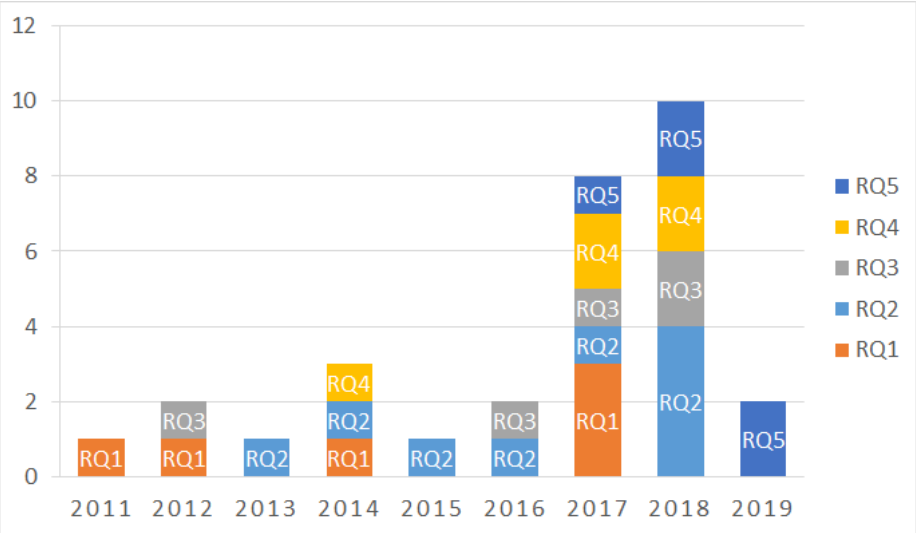

Figure 3.1: Finalized selection by year

If we consider the finalized selection (Figure 3.1), it is even more apparent that the field is getting more attention recently. A big increase in the total amount of research is visible in the years 2017 and 2018. The search was done during the second quarter of 2019, this can possibly explain why 2019 does not appear to continue the trend of the previous years. Figure 3.1 also shows the amount of papers selected per year on the research question level. It is notable that the oldest selected papers for RQ4 and RQ5 were published in 2014 and 2017 respectively. This indicates that TD in relation to software maintenance and productivity has not been investigated until recently.

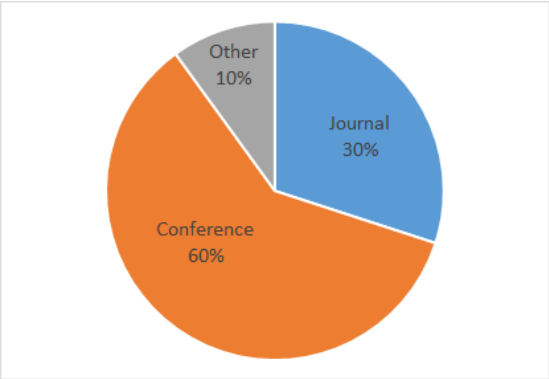

Figure 3.2: Publication types of finalized selection

Figure 3.2 gives an overview of the publication types in the final set of studies. Most of the selected studies are conference proceedings (18 or 60%), followed by journal articles (9 or 30 %) and lastly other types of publications (3 or 10%). The publication types of the selected studies can be used as an indicator for the maturity of the research field. The large portion of conference proceedings in the selected studies, implies a lot of new contributions are made. This is in line with our other observations regarding the selected papers, suggesting a recent influx of academic interest in the field.

3.1 Software maintenance

As already briefly discussed in the introduction, the goal of software maintenance is to sustain the product during its life cycle. This is done by continuing to satisfy the requirements of the users. User requirements change over time, so in order to extend the life time of a product, new features have to be added. However, when products grow in size, so does their complexity. More complex products are harder to maintain and require effort to keep performing. Software maintenance consist of multiple activities which all affect the product. This is often an balancing act for the manager, as they have to put emphasis on which activity is the most important one at a certain point in time. In this section we discuss the current thinking on software maintenance from literature.

The study by Lientz and Swanson [33] was added as a result from backwards snowballing, 5 out of 6 selected papers reference their classification. Also a study by Chapin [16] was added by the researcher as its literature review was deemed highly relevant to the research question and of good quality.

Most papers [1] [37] [24] [20] [39] used the work of Lientz and Swanson [33] as a basis to classify software maintenance activities. Lientz and Swanson discussed three maintenance activities: perfective, adaptive and corrective maintenance. The authors of [20] used this exact same classification, [1] [37] [24]

made the addition of preventive maintenance, this addition is also supported in the ISO/IEC 14764 standard and the SWEBOK [34]. Furthermore, [39] used a classification based on the work of Chapin [16].

Adaptive maintenance aims at changes to the software to cope with changing environments or new technologies. Corrective maintenance is concerned with activities correcting known problems. Perfective maintenance are improvements made to the software to satisfy new user requirements [33]. Preventive maintenance is "modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults." [34].

Chapin [16] did a comprehensive literature study into software maintenance activities, with the goal to create a more fine-grained classification. The activities are classified by the actual work done, unlike the previous classifications, where activities are classified by the intended purpose of the work [33]. He also discusses software evolution activities which do not contain source code changes. Therefore, we only consider those activities which are considered software maintenance from his classification: adaptive, corrective, preventive, enhancive, reductive, performance and groomative.

The authors of [37] investigated the relationship between issue resolution time and the maintenance type. Using the data of 34 open source projects containing over 14000 issue reports they found a significant correlation between the issue resolution time and the maintenance type. Time spent on corrective and perfective maintenance per issue is less than than the time spent on adaptive and perfective maintenance activities [37]. Using these findings, they were able to estimate the effort required for issue reports based on historical data. This estimation can be useful for managers who need to balance the maintenance effort. One limitation is that most of the issues were bug related. This limitation was also present in the study done by [24] and can bias the estimation towards corrective maintenance.

The software maintenance literature has arguably matured considerably. The classifications in the field remained largely the same over the last 10 years. While Chapin [16] proposed the most complete overview of possible activities, it did not gain enough track in the community. More recent studies still use the original classification by Lientz [33] with the addition of preventive maintenance. We also prefer this classification while its more simplistic yet complete enough.

3.2 Technical debt

3.2.1 Technical Debt as a concept

Lavazza [32] argues that TD can be seen as an external software attribute. External attributes require information about the environment of the system in order to be measured. This is also true for TD, as it cannot be measured by software attributes alone. TD also depends on external attributes such as

Type	Lientz 1978[33]	Chapin 2003[16]	Nguyen 2011[39]	Edberg 2012[20]	Murgia 2014[37]	Wu 2017[61]	Abdullah 2017[1]	Grover 2017[24]
Corrective	x	x	x	x	x	x	x	x
Perfective or enhancive	x	x	x	x	x	x	x	x
Adaptive	x	x	x	x	x		x	x
Preventive		x	x		x		x	x
Reductive		x	x					
Performance		x	x					
Groomative		x	x					
General						x		

Table 3.2: Overview of different maintenance classifications

technology. In that way, a piece of code cannot be seen as TD, merely as an attribute of the code. The code can be seen as an entity related to TD. We call this entity a "Technical Debt Item" (TDI).

The cost related to a TDI has multiple components: the principal, the amount of interest and the interest probability. The principal is the cost or the amount of effort required to eliminate the TDI according to the desired quality level. Unlike a financial debt, where the principal is fixed from the beginning, the principal of TD can change over time. For example, the release of a new tool reduces the effort required by the developer to fix the TDI. In this case the principal of the TD is reduced while the TDI remained exactly the same. The amount of interest can be defined as: *"the potential penalty in terms of increased effort and decreased productivity that will have to be paid in the future as a result of not completing these tasks in the present"* [48]. The interest is also associated with a probability: the chance that unpaid debt will result in additional interest. As TD depends on multiple outside factors, from a modeling perspective, these factors contain some level of randomness. Therefore, TD is an estimation of the true costs when the debt has not been paid in full [32]. However, sometimes TD is never repaid, for instance when an application is retired at the end of the life cycle. Reducing or paying off TD can be seen as an investment, as resources are spent in order to avoid spending more resources in the future. The return on this investment has a degree of uncertainty.

3.2.2 Technical Debt types

Since the introduction of the Technical Debt metaphor by Cunningham [17], many sub types emerged in literature to describe a specific kinds of debt. Both the authors of [5] and [48] attempted to map these types by conducting a systematic mapping study. The most recent being the one by Rios in 2018 [48]. Both these studies have similar results, see the types of TD presented in Table 3.3. During the four years between these individual studies, only two new types of debt have been proposed. These are usability and versioning debt. Versioning debt refers to unneccessary code forks, which could be considered as build debt in the classification of Alves [5]. Usability debt refers to inappropriate usability decisions that have to be altered later. The additions by Rios [48] on top of the classification by Alves [5], are very specific and could arguably also be considered other types of debt. Therefore, we prefer the classification by Alves

TD Type	Alves 2014 [5]	Rios 2018 [48]
Architecture Debt	x	x
Build Debt	x	x
Code Debt	x	x
Defect Debt	x	x
Design Debt	x	x
Documentation Debt	x	x
Infastructure Debt	x	x
People Debt	x	x
Process Debt	x	x
Requirements Debt	x	x
Service Debt	x	x
Test Automation Debt	x	x
Test Debt	x	x
Usability Debt		x
Versioning Debt		x

Table 3.3: Types of Technical Debt

[5].

3.2.3 Identification of Technical Debt

Multiple tools exist to identify TDI at the source code level of the artifact. It is important to note that not all Technical Debt Items of the artifact are present in the source code. Therefore, the team can not solely rely on tools for the identification of a TDI and has to consider other strategies as well [48]. Possible strategies include manual tracking of TD. While manually tracking TD can be more complete, because not all TD can be found by analyzing the source code. One large disadvantage is that this approach is more time consuming. A mixed method approach would be ideal.

The authors of [21] proposed a method to quantify the interest associated with a TDI. The interest is calculated based on historical quality rule violation in project. The interest is quantified as the amount of extra defects that occurred in the past or will occur in the future [21]

While TD originated as a metaphor to explain the risks of low quality code, it matured into a concept that is widely applied in the field of software engineering. Many different types of TD have been discussed in literature. Multiple systematic reviews give a complete overview over these sub types. We use the classification of Alvez [5]. Especially the design, code, architecture, test and defect debt are of our interest. As these sub types are directly affected by maintenance activities. Furthermore, we found that in order to identify TD, a mixed strategy would ensure the best estimation.

3.3 Productivity

From an economical perspective, productivity measures output per unit of input. In a organizational context the input translates to effort or resources as input and produced units as output. Measuring the productivity of programmers is a common challenge in software maintenance, as the produced units are not always visible [54]. The authors of [52] also indicate difficulties in measuring the size of the output, as some size measures are not clearly defined and are not repeatable. While there are different methods of measuring productivity, all authors of the selected papers [14], [15], [52], [54], [43] agree that productivity can be defined as a function of the size of the product and the effort required, where effort is dependent on a time measurement.

3.3.1 Size

Size can be measured in multiple ways, usually done by using a software metric or a combination of metrics. Source lines of code (SLOC) are an example of this. While these metrics are often easy to compute, they do not always give a good representation of the actual work done. To cope with this problem, Sneed and Prenter [52] suggest adding a complexity factor to the size measurement of a work item. This measurement is done on a scale from 0.5 to 1.5, where the complexity factor is the degree to which the measured complexity varies from the median complexity. The complexity factor is simply multiplied with the absolute size measurement and results in an adjusted size. This metric gives a better representation of the size of a work item because not all items with the same size are equally difficult to maintain.

3.3.2 Effort

Under effort, we consider the resources an organization needs to invest in order to add value. In the case of software, no raw materials are needed for production. One study [54] does consider computer resources such as CPU time as an input for the productivity calculation. In most cases this is negligible, therefore, we only consider labor in the effort measurement. As stated before, effort is a time measurement, the unit is not relevant. Because in order to see productivity trends, it only requires to be measured in a consistent way [43]. Effort is often measured in (working) days or hours.

It is not hard to compute the size and effort metrics of maintenance teams. By combining these two results, the the productivity of the team can be calculated. Giving an accurate estimation of the productivity is much more difficult, as not a single size or effort metric gives a holistic view of the work completed. Therefore, the most important thing is consistency, measuring in the same way, every time. Comparing individual teams in terms of productivity is almost impossible, however, it is an excellent measure to track team performance over time. Software maintenance is still heavily reliant on people. This can also be

a reason for not measuring productivity. Developers might feel uncomfortable with the pressure that monitoring can introduce.

3.4 Effect of software maintenance effort on Technical Debt

3.4.1 Accumulation of Technical Debt

Not all maintenance efforts accumulate the same amount of TD. While adding a new feature to the system can definitely result in TD accumulation, when good quality practices are in place it does not necessarily occur. The amount of TD that accumulates is highly dependent on the situation. The individual developer plays a part in this [3].

Furthermore, maintenance activities which are under time pressure or build on immature code are the most susceptible to incur TD [46] [35]. The amount of TD accumulated during the products life cycle is highly dependent on the quality of the maintenance effort [46]. While more senior developers have proven to produce less TD overall [3], most TD is self-admitted [55].

3.4.2 Reduction of Technical Debt

Not all reduction of Technical Debt is planned beforehand, as many small TD items are fixed ad hoq, when a developer comes across it in the source code [35], [55]. A study done by Digkas et al. [18] investigated how developers pay back TD. Their work shows that especially these relative small fixes of TD items contribute to the majority of TD reduction during the product's life cycle. The most occurring fixes are: reduction of complex methods, eliminating duplicated code and exception handling problems.

One study shows that self admitted TD is most often fixed by the person who created the TD in the first place [35]. The amount eliminated TD varies per developer, as some developers introduce more TD than they reduce, or the other way around [55].

While the majority of TD items is eliminated by fixes during the life cycle [18], retirement of the product is another way to eliminate debt. Which of these two ways is more economically sound depends on the life time of the product [46]. Around 60% of all TD will be repaid during the life time. Repayment is most often done within a year of the TD introduction. If after this period the TD still exists, it is unlikely that it will ever be repaid [18].

3.4.3 Maintenance activities

It is clear that the continuous process of software maintenance affects the amount of TD. However, the strength of this relationship is highly dependent on multiple factors. Take for example perfective maintenance: if a new piece

Activity	Effect on technical debt
Perfective	Unchanged or increase
Corrective	Reduction, unchanged or increase
Adaptive	Unchanged or increase
Preventive	Reduction

Table 3.4: Effect of maintenance activities on technical debt

of functionality is added to the product, made accordingly to the quality standards, the amount of TD is unchanged. However, if coded poorly (by accident or not), TD will increase. An overview of the possible effect of maintenance activities on TD can be found in table 3.4. Preventive maintenance is key in managing TD, as it directly reduces the principal of TD.

3.5 Effect of Technical Debt on productivity

Unlike the effects of software maintenance on TD, the effects of TD on productivity are far less ambiguous. Nema et al. [38] did a comprehensive literature review on Technical Debt in the context of agile software development. One of the research questions addressed in this review is: "What are the related causes and consequences of incurring Technical Debt in agile software development". Out of 38 primary studies between the years 2002 and 2014, 17 studies reported a loss in productivity due to TD accumulation. Furthermore, 17 studies reported degradation of system quality and 15 reported increased cost of maintenance. Especially when TD is not incurred strategically, it slows teams and lowers productivity due to the extra effort required of fixing bugs and stability issues [38].

Another literature review in effects of TD was done by Spinola et al. [53]. In their work, they present a probabilistic cause and effect diagram for TD. This model can be used to predict the likelihood of a certain TD effect. While their model claims only a direct fall in productivity in 1.7% of all cases, many effects of TD indirectly lower productivity. If we combine these indirect effects, we get a 55.6% chance of TD lowering productivity. These losses in productivity can be significant, as studies have shown: on average 23% of development time is wasted due to TD. Furthermore, developers often incur new TD due to existing TD [11].

We can conclude the effects of TD can seriously impede the productivity of maintenance teams. As labor costs are the highest cost factor in software products, small improvements in productivity can be highly lucrative.

3.6 Summary and conclusions

3.6.1 RQ 1

Software maintenance literature exists for some time now. The core activities of software maintenance have not changed over the years. Also, the fundamental goal of software maintenance remained the same, namely extending the life time of a software product. Over the years multiple classifications have been proposed in literature. The difference between these lie predominantly in the granularity of activity types. This does not necessarily imply that newer classifications are fundamentally better. One change that did happen in the software maintenance landscape is the average lifespan of software. This introduces new challenges and results in software maintenance still being a relevant field for researchers.

3.6.2 RQ 2

Compared to software maintenance, the concept of TD was introduced in scientific literature more recently. In recent years it gained substantially more interest from scholars. There is a consensus in literature that TD can be broken down in two main components, the principal and the interest of the debt. It is only possible to calculate the exact costs when a debt is fully repaid. As both the principal and interest can change over time, current TD is always an estimation of costs. We found many different sub types of TD in literature. These can help differentiate different types of debt. Furthermore, we found that sometimes TD is incurred on purpose, often with the goal of accelerating a feature's the time to market. Multiple tools exist to manage TD, these tools (automatically) track TD of a project over time. Automated TD tools rely on the source code for TD estimations. These estimations are based on code hygiene and best practices. TD types which cannot be derived from the source code can only be tracked by hand, however, this can be very time consuming. Finally we found that TD is most often measured in time, which in turn can be used to calculate the financial metric.

3.6.3 RQ 3

Since the beginning of the software engineering field, researchers have tried to determine the productivity of developers. Similar as in economics, productivity is a function of input and output, over a period of time. For the input we can consider all the resources consumed by the organization during "production". Software engineering is very labor intensive and does not consume raw materials unlike traditional manufacturing processes. Therefore, literature considers labor costs as the only resource of production. We found that the measurement of the output is more often a point of debate among scholars, as it is sometimes already hard to determine what is actually produced, let alone to measure it. The most common method is to use a size measurement for the amount of code that is changed or produced. The advantage of using size metrics is that they

are relatively easy to compute and are quite consistent, making them suitable for tracking trends within single projects. The disadvantage of size metrics is that they are highly dependent on the project type and external attributes. This makes using only size metrics to compare different projects unreliable and therefore undesirable. Both Logical Lines of Code (LLoC) and Functions Points (FP) are often used as size metrics in literature. Both take the complexity of the written code in consideration, resulting in less variance of the measurement due to the way of working of the individual software engineer.

3.6.4 RQ 4

Inherently, software maintenance has an effect on technical debt, as changes made to the source code also change the amount TD of the project. The effect differs depending on the maintenance activity that is being performed. Both perfective and adaptive maintenance tasks add additional functionality to the project. Depending on the quality of the newly added code, the total amount of TD remains the same or increases. Corrective maintenance can both reduce and increase TD, while preventive maintenance has the specific goal of reducing TD.

3.6.5 RQ 5

We found multiple studies that investigated the effects of TD in the field of software engineering. Most of them reported negative effects on the overall productivity due the consequences of TD, both by direct and indirect causes. In some specific cases TD can improve productivity, however, it is important to note this is only possible when we consider productivity over a short time period. When TD is not incurred strategically, we can assume it lowers long term productivity, by decreasing the overall system quality. This results in higher maintenance costs.

Chapter 4

Design and development

4.1 Introduction

The literature review emphasized the potential impact TDM can have on organizational performance. As we have shown, TD can be manipulated in favor of productivity by an effective allocation of resources. Surprisingly, TDM is still overlooked by many managers in the field of software engineering. This is demonstrated by the lack of current TD management methods which facilitate in long term decision making.

In this chapter we propose a theoretical model which can be seen as the foundation of our artifact. The goal of this model is to simulate the effects that software maintenance has on TD. It should help practitioners reduce wasted development time due to TD and therefore it should increase the overall productivity.

As most managers have to deal with a fixed amount of resources, we want our model to support them in making long term decisions regarding TD, based on their current team capacity. While we focus on manipulating TD by resource allocation strategies, TD is also highly dependent on other variables. Especially the amount of TD that is incurred by performing other activities than preventive maintenance, plays a huge role in this. As the goal of this research is to look at high level strategies, the impact of individual performance does not have to matter when the project is large enough. Therefore, we leave it out of the scope of this model.

4.2 Building the model

Before building the model, it is important to understand the context of the problem that we aim to solve. Figure 4.1 gives a visual representation of the constructs discussed in Chapter 3. Performing maintenance tasks has some effect on the amount of TD that is present in the project. In turn, this TD has an

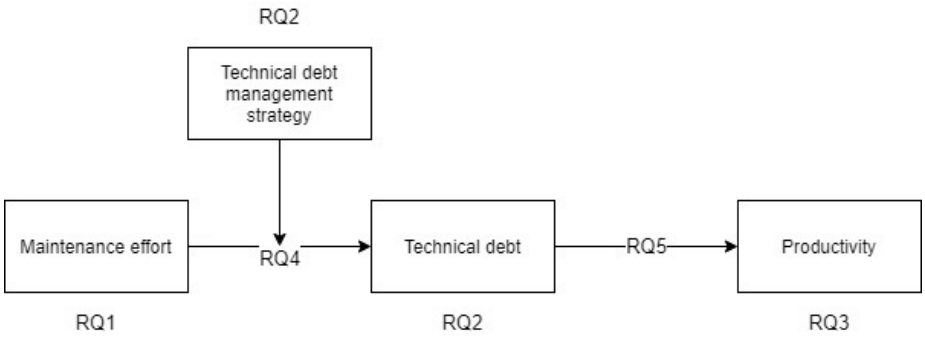

Figure 4.1: High level overview of the problem context and the related research questions

(negative) effect on the overall productivity of this project. Technical debt management mediates the relationship of the maintenance effort and technical debt, as we have seen that TDM can strengthen or weaken the effect maintenance effort has on TD. The process presented in the figure should be considered as a cyclical model, as software maintenance is a continuous process. Depending on the level of analysis, one cycle can be an arbitrary amount of time, e.g. weeks or months. As our research is placed in the context of agile software engineering, we will consider one cycle as a single sprint/iteration form here on.

The TDM strategy will play a central role in the model, because it is a more easy and controllable way to manipulate TD in a real world setting. This is due to the total amount of available maintenance effort is often restricted by the team size. We assume a TDM strategy is established before each sprint cycle and is not altered during the sprint. A TDM strategy aims to manipulate the maintenance effort in order to achieve a desired beneficial outcome. The desired outcome can be expressed in internal and external software attributes. In the case of our model, these are: functional size, technical debt and thus indirectly also productivity.

The desired outcome is heavily dependent on the project and associated goals management tries to achieve. For instance: when maintaining software which is going to be replaced in one year, TD is far less important than when the software is going to be maintained for 10 years. This means that an optimal strategy for one scenario does not necessarily mean it is also optimal in some other scenario.

We distinguish three types of TDM strategies, these are: fixed, variable and no strategy at all. With no strategy, we mean that no effort is spent on preventive maintenance intentionally. A fixed approach entails that a fixed percentage of the resources available, is dedicated to preventive maintenance each sprint. When using a variable strategy, the percentage devoted to preventive maintenance depends on both the current state of the project and organizational factors.

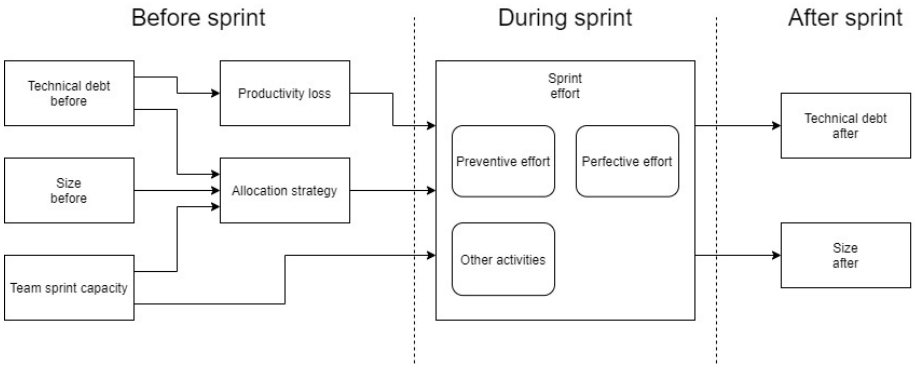

Figure 4.2: Allocation process model

Combining the components results in the process model presented in figure 4.2.

4.3 Components and measures

4.3.1 Before sprint

Technical debt

This describes the total amount of TD (principal) present in the current state of the software project. Preferably measured in units of time (hours/days/weeks), this amount represents an estimation of the total time required to repay all the TD.

Size

We use size as a way to represent the total functionality of the software. Size can be measured in multiple ways, preferably using a methods which take complexity in to consideration. As long as it is measured in a consistent way, the measure itself does not really matter. This is because the goal is compare different iterations of the same project, not different projects. Function points (FP) would be the most valid way of measuring functionality. A less valid measure would be lines of code (LOC), but is far more easy to compute and is less biased compared to FP, and thus more reliable. The trade-off between validity and reliability is up to the practitioner to make. We prefer logical lines of code (LLOC) as a good compromise. It is still reliable because it can be computed by a static code analysis tool. The advantage over LOC is that only logical statements are counted towards the total. It excludes comments and accounts for differences in code density. This means that coding style does not interfere with the metric and is independent from the language.

Team sprint capacity

This can be seen as the production capacity of the team. We define the capacity as the nominal amount of man hours available for the next sprint. In the real world this amount would differ from actual sprint effort, as a team is never 100% productive.

Productivity loss

Productivity loss can be seen as the penalty of having TD, or as interest payments towards the debt. Therefore, the productivity loss directly depends on the total amount of TD, regardless of the total size. Similar as the principal, it is best measured in units of time. The amount of time is the expected time wasted on tasks as a result of having TD, or the time spent on corrective tasks for the coming sprint.

Allocation strategy

The allocation strategy determines how much effort should ideally be spend on what type of maintenance activity. This depends on the available resources (team sprint capacity) and how badly the product is affected by TD. As the total amount of TD in itself does not give a good representation of the current quality level of the project, we factor in the size of the project as well. The function of TD and size is referred to as the relative technical debt. Combining this with the team capacity, we can calculate a resource allocation for the coming sprint. This calculation results in the percentage of available effort that should be spent on preventive maintenance tasks, ranging from 0–100%.

4.3.2 During sprint

In this stage, the actual maintenance tasks are preformed. Preferably the actual distribution of maintenance tasks should be as close to the resource allocation strategy as possible. However, in a real world scenario, there are always unforeseen events that alter this distribution.

Preventive effort

The preventive effort is the percentage of the actual effort that is spend on preventive maintenance tasks, i.e. directly geared towards reducing TD.

Perfective effort

The perfective effort is the percentage of actual effort that is spend on creating new functionality, depending on the quality of this newly added functionality, some amount of TD is incurred when performing this task.

Other activities

These activities are regarded as unexpected effort which neither is spend on preventive or perfective tasks. This may include things such as time spent on side projects or overhead.

4.3.3 After sprint

As both preventive and perfective work affect the product's code size and technical debt, these have to be re-evaluated after each sprint.

By definition, preventive work has a negative effect on TD. As TD is measured in time, the hours spent on preventive maintenance should perfectly correlate with the actual reduction in TD. Reducing TD should change very little to the functionality of the product, so the effects on the code size are negligible. The addition of new functionalities increases the code size. It is highly likely that some of pieces of added code contain some TD and therefore also increase the total amount of TD of the product.

4.4 Requirements

In order to better structure the design the artifact, the author made a series of requirements, which have to be met in order to solve the research problem. These requirements are based on the authors opinion regarding the subject. We specify requirements for each of the four categories of Lauesen [30]: goal-, domain-, product- and design-level requirements. This is the authors preferred way of creating high level requirements.

4.4.1 Goal-level requirements

Based on external software attributes and historical data, managers should be able to determine the optimal resource allocation strategy for a selected future point in time. This does not incorporate the cost-benefit analysis of individual TD items, as we have seen there are already multiple solutions available for that specific task.

4.4.2 Domain-level requirements

- The artifact can estimate the effects of allocation strategies on TD and size.
- The artifact should work for multiple different types of software projects.

4.4.3 Product-level requirements

- The artifact should support simple allocation strategies, with a fixed percentage of preventive maintenance.

- The artifact should support variable allocation strategies, the percentage of preventive maintenance is relative to the relative technical debt.
- The artifact should be able to account for the penalty of having TD, for each simulated sprint.
- The artifact should be able to account for newly introduced debt, as a result of the newly created functionality.
- The artifact calculates the increase in functionality for the next sprint, based on the allocation strategy.
- The artifact calculates the increase of TD for the next sprint, based on the allocation strategy.

4.4.4 Design-level requirement

- The artifact shall visualize multiple strategies in a graph by plotting size and time.
- Input variables can be given by the user before running the code.

4.5 Design of the artifact

The first version of the algorithm is built by the author in the form of a python script. Based on several input variables, which are explained in the next section, the algorithm will compute the effects of different allocation strategies. Calculations are done per iteration, with the results as input for the next iterations, this is repeated for a number of n times.

The results of each strategy are visualized in a graph, so the practitioner can easily determine which strategy is optimal for the time-frame he wishes to investigate.

4.6 Input variables

The artifact requires several inputs from the user in order to function correctly. The first version is based on the following inputs: Size, Available effort, Technical debt, Interest rate, Quality standard and Productivity

4.6.1 Internal and external software attributes

The software attributes are: size, technical debt and interest rate. We highly recommend the use of a static code analysis tooling for the calculation of these attributes. The algorithm needs to have a TD measurement as input variable, so using a static code analysis tool with support of TD measurement is vital for the correct working of the artifact. As the study by Rios [48] highlighted, many

tools exist for the identification of TD. It is up to the end-user to select such a tool and to configure it.

The interest rate is introduced to estimate the effects of the current TD on productivity. The interest rate is the expected percentage of the total TD that has to be spend each iteration. For example, if at any point in time the interest rate is 6% and the total TD is 1000 hours, we expect to waste 60 hours of our available effort due to TD.

Our algorithm is designed around the use of a TD measurement in hours. So the cost or principal of every TD item should be measured in hours or converted to hours. If the user chooses a tool which does not support this feature, an average fixing time should be determined for each TD type.

Furthermore, most static code analysis tools can easily calculate the size of a code base. As mentioned in Chapter 4, we prefer the use of logical lines of code, however, any numeric size measurement can be used.

4.6.2 Organizational attributes

The organizational attributes are: available effort, quality standard and productivity. The available effort are the man-hours that can be spend on the project for each iteration. Based on these hours, the algorithms will make a resource allocation strategy, taking the time wasted as a result of TD into account.

The average quality standard of the developers is also important, as newly created code is seldom without flaws, thus containing TD. We express this attribute as the average amount of hours TD that is introduced for every unit of size. E.g. a value of 0.05 would mean in our case, that every new LLOC added to the source-code contains on average 3 minutes worth of TD.

The productivity of the employees working on the project is also required as an input for the algorithm. It refers to the average amount of size units that are produced every hour that is spend on perfective tasks.

4.7 Allocation strategies

4.7.1 Fixed strategy

A fixed allocation strategy is where the amount of resources devoted to TD prevention are the same for each iteration, regardless of the current level of TD. The amount is expressed as a fixed percentage of the total available effort, visualized in figure 4.3.

The advantage of this strategy is the ease of use, as little calculations are needed to determine the distribution of available resources. The strategy can be used in practice as a quick and easy improvement of the current quality level. As in most cases, some preventive effort is always better than none.

This strategy also has disadvantages, first of all, there is no perfect ratio which works all the time. The optimal ratio is dependent on many different attributes and therefore hard to get right. The result of this is, is that the fixed

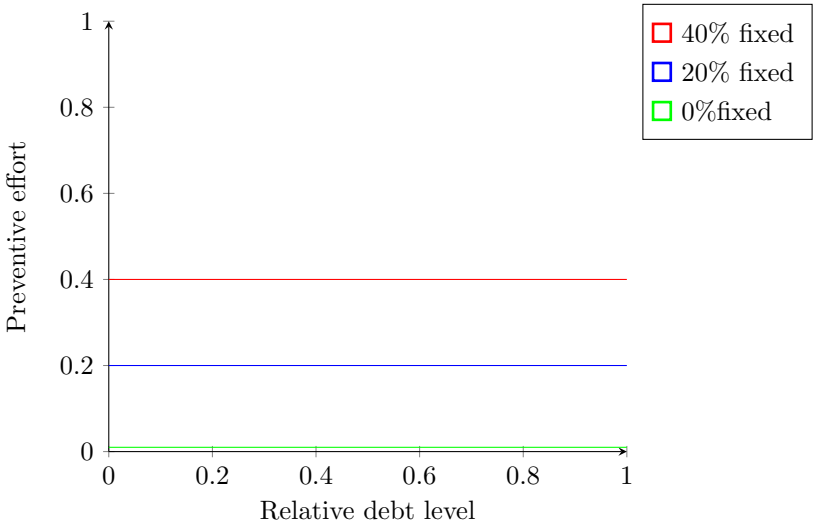

Figure 4.3: Preventive effort based on the relative debt level

ratio, as determined by the practitioner, is either too high or too low. When the ratio of preventive work is too low, the danger arises that the amount of TD will become too large, hindering productivity due to the extra effort required as a result of that debt. On the other hand, if the ratio of preventive work is too high, not enough time is spent on creating new functionality. This results in a less desirable product for the end-user.

4.7.2 Variable strategy

We consider an allocation strategy variable when the effort is dependent on the current TD level. The relative preventive allocation effort can be expressed as a function of the relative debt level. The function defines the aggressiveness in which TD is removed. Figure 4.7.2 visualizes multiple variable allocation strategies as exponential formulas, with different levels of aggressiveness. As an addition to the strategies as modeled in the figure, one could also add certain thresholds. For instance, when the relative debt level is below a certain number, no preventive maintenance will be conducted.

The biggest advantage of this type of strategy is its flexibility towards unexpected changes in effort. For example, due to some unforeseen events, not enough effort was spent on preventive maintenance during the previous sprint cycle. In a fixed strategy scenario the amount of preventive maintenance would just stay the same for the next sprint. However, with a variable strategy, extra resources can be allocated when the TD level rises. So a variable strategy makes it easier to keep the amount of TD at the desired level.

One disadvantage is that this strategy might be harder to implement. Since the preventive effort would change every iteration, it requires more work than

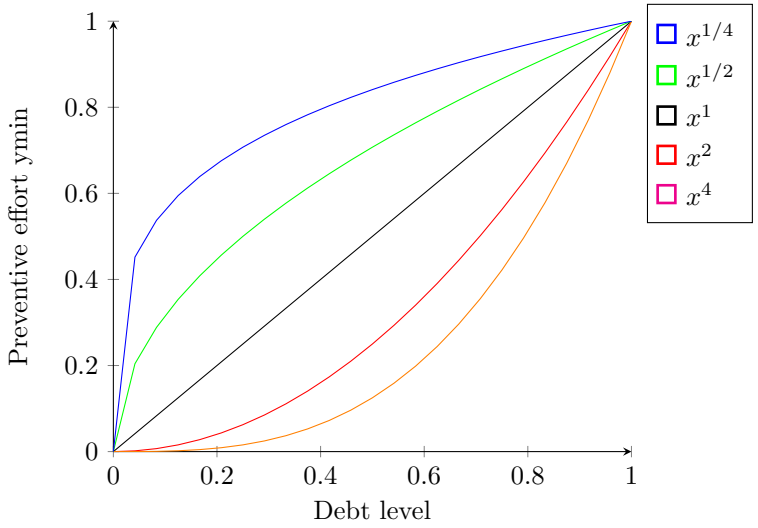

Figure 4.4: Preventive effort based on debt level

a fixed strategy to calculate.

4.8 Logic

For the first version of the artifact we only distinguish two types of maintenance activities for the allocation of resources: preventive maintenance and perfective maintenance.

Now that we have defined all input variables and the strategies, we can build the underlying logic of the algorithm. Before we introduce the logic of any specific allocation strategies, we first have a define generic function which estimates the TD and size after one sprint cycle. So for any allocation distribution, taking into account the current TD, size, available effort and interest rate, this function outputs the new size and TD.

This calculation is done by first subtracting the expected lost time, due to the current debt (interest), from the available effort. This results in the estimated real available effort. If we multiply this with the ratio of preventive maintenance, we get the reduction of TD. One thing to note is that this is not the final reduction of TD for this sprint cycle, as the perfective effort can also introduce new TD. The ratio of perfective effort multiplied by the estimated real available effort results in the time that will be spent on perfective tasks. If we multiply this time with the average LLOC a programmer produces, we get the increase in size. The quality standard determines how much TD this newly added functionality contains.

In order to determine the resource allocation that will be used in the next sprint cycle, we need some functions which calculate this for various strategies.

Determining this for a fixed strategy is very easy. As the fixed percentage is effectively the allocation ratio to preventive tasks. The remainder of the available effort is the ratio spent on perfective tasks.

Relative strategies are a bit more complex but still easy to calculate. As the prefentive effort is a function of relative debt, we just need to know the debt and exact function. The relative debt is defined as total size, times the average time to produce one LLOC, divided by the total amount of TD. This gives us the following formula to calculate the preventive effort:

$$E_{preventive} = Debt^x$$

Where x is the level of aggressiveness, i.e. how strong we act on TD. If the strategy is to allocate the preventive effort 1:1 according to the debt then $x = 1$, results in a linear pattern. For $0 < x < 1$ the strategy becomes more aggressive in reducing TD as x moves towards 0. For $x > 1$ the strategy puts lesser emphasis on reducing TD at low debt levels as x becomes larger.

Having both the functions for determining the resource allocation and estimating the effects, we can now compare the performance of different strategies. For each strategy, we calculate the allocation based on the initial input variables, then perform the simulation of one sprint cycle, those results are then saved into a dataframe. After saving they serve as the input for the next allocation calculation and sprint cycle and so fort. By doing this for each strategy a number of n times, we can determine the optimal strategy.

Chapter 5

Demonstration

As mentioned in Chapter 2, we will use technical action research as our validation method. To align this method with the structure of DSRM, we consider step 1-6 of TAR as the demonstration part and 7-9 as evaluation. Step 7-9 are discussed in Chapter 6. The goal of this chapter is to investigate if the artifact has the expected outcome when it is introduced in the intended context.

5.1 Research problem analysis

The demonstrated artifact is the algorithm which is build based on the TDM process model. The intended context of the artifact is existing software maintenance projects which are in need for high level TDM.

The client participating in this research is Topicus, a large IT company from the Netherlands. The company has over 1000 employees and is active in multiple sectors. They specialize in enterprise information systems.

We demonstrate the algorithm on one specific product within their financial division. We investigate the effects of the interaction between artifact and context. Furthermore, the artifact has to satisfy the requirements mentioned in Chapter 4.

5.2 Research and inference design

At the implementation of the artifact, we make use of the same measurements as defined during the design and develop activity, as can be found in Chpater 4. This makes it easier to demonstrate, as the artifact requires little customization. Furthermore, it helps the validation of the original algorithm. As it is heavily dependent on measuring the external environment, we need these measures to be representative.

The primary data sources are the multiple snapshots of the source-code. Also data from the time registration software is used to determine effort metrics.

All measurements gathered during the demonstration is stored in a database. The data does not contain any sensitive information and can therefore be made available for other researchers on request.

5.3 Problem investigation

The specific software product of Topicus, which we consider the context of this demonstration, is a large enterprise level information system for large financial institutions. At the time of writing, this product is almost 10 years old and still under continuous maintenance. Around 100 employees work on the project, of which a large portion are software engineers.

As we have seen, the concept of TD has attracted lots of interest from scholars in the last couple of years. The same applies practitioners. TDM has been on the radar of the client's management team for some time now. However, only small efforts have been made in managing it thus far.

At the time of the initial development of the product, a micro-service architecture was relatively uncommon. As with many other products from the time, it was built a monolithic piece of software, many dependencies make it time intensive to maintain.

While efforts have been made to reduce architectural debt by refactoring the source-code to a more service oriented architecture, it still contains considerable TD. With the amount of wasted time spent on fixing bugs rising, managers started looking into TD. Since some time now, the teams that work on the project implement a fixed TDM allocation strategy. Apart from this, little TDM practices are in place.

Talks with the client indicate that TD is not actively monitored and thus the fixed strategy is not validated. While some employees indicate that some TDM is probably beneficial for the performance in the long run, nobody seems to know how much time should be spent on reducing TD. Even though the fixed strategy is in place, some managers still prioritize the perfective activities in order to make deadlines and therefore deviate from the strategy.

We established the following client goals: 1. Validate the benefits of better technical debt management. 2. Improve long term productivity by implementing a more beneficial TDM strategy.

We aim to contribute towards the goal by demonstrating our algorithm in this specific case.

5.4 Client treatment design and validation

We plan the treatment in two individual parts. First we gathered and prepared the required data. In the second part we apply the algorithm on this data-set. We do this so we can fist check the data reliability before actually running the algorithm, reducing the chance of analyzing results which are based on unreliable data.

Timestamp	Size	Debt	Annual interest	Cumulative effort
1-1-2020	500000	1100 days	360 days	0
1-14-2020	501000	1102 days	361 days	250 hours
...	...	...	...	...

Table 5.1: Example of data collection table

The data gathering part is the majority of the work, because we have to collect data from multiple sources, prepare the data and validate it. However, from both the clients' and researchers' perspective, the second part of the treatment is more crucial. As the algorithm aims to give new insights to the client in TDM, while the researcher can observe the artifact in its intended context.

5.5 Implementation

5.5.1 Data gathering

Data gathering is an important step before we can implement the actual algorithm. As unreliable data will result in an unreliable outcome. The goal is to get an idea on how the software and organizational attributes change over time and end up with a reliable set of input variables for our algorithm.

The first source of data is a set of snapshots of the source-code. Based on code analysis we want to extract key variables such as the amount of TD. We require multiple historic snapshots of the same code base as things like quality standard cannot be derived from a single snapshot. The second data source is the internal billing system, here all employees register the amount of hours they worked on a project. Combining these sources gives us the ability to compare the state of the software in relation to the effort spent on it.

Unfortunately there were no historic snapshots stored of the whole code base, due to size limitations. Therefore we first conducted the data gathering process for a single component of the software. We combine and store the gathered data in a dataframe, an example of the schema used can be found in table 5.1. This allowed us to prove that the data gathering techniques were reliable, before applying it to the most recent version of the total source code.

Data gathering of software attributes

As the product in our case is built using the .NET programming language. We decided on NDepend as our analysis tool. NDepend is a static code analysis tool specifically build for .NET applications, whilst it is also build with TD identification in mind. The tool gives us general code metrics, such as the total number of LLOC, dependencies and complexity. Furthermore, the TD identification component measures both the principal and annual interest of TD.

For the TD calculation, NDepend analyses the source code based on a set of predefined rules. The used set of rules is based on what is considered good coding practices. This makes the tool suitable to measure most of the test, code and design types of TD. Some rules can also detect architectural debt. In order to get an accurate estimation of the amount of TD, we used a total of 121 rules, each rule having a different debt estimation and interest estimate. An overview of the rules can be found in appendix B. The debt estimation is left on default, as it is almost impossible to determine the actual amount of debt without having accurate historical debt data.

As the analysis is done on the source code level, TD can be discovered at a very low level. This enables us to aggregate TD by methods, classes or components. The principal of TD items are expressed as the estimated time required to eliminate the TD item. The interest of TD items are also expressed in time, namely the estimated annual additional time spent on maintenance due to the TD item.

We also use NDepend to calculate the functional size of the product for each snapshot. This is done using the logical lines of code metric. A more detailed explanation on how this this metric works can be found in Chapter 4.

Data gathering of organizational attributes

To get a good estimation for the effort that has been spent on the product, we can make use of the hour registration software which is already present at the company. Each employee is deemed to register the amount of hours worked on a single project. Unfortunately not all data is of the same level of detail.

For instance, sometimes (especially when invoiced to a third party) the registered hours are traceable to a specific bug, by linking the registration entry to a user story in Jira. The opposite also exists: large amounts of untraceable work is registered on generic project entries. This makes it hard for us to distinguish historical resource allocation based on the registered hours.

We therefore chose to aggregate all hours worked on the whole project, including overhead. Because of the large amount of employees, the average should give us a good indication of how much time can be allocated towards future sprint cycles. As effort can change overtime, we track the cumulative effort since the first time stamp.

We excluded all other employees than developers from the data-set, because those roles do not directly contribute to the functional size of the source code. This is done to simplify the measurements, otherwise a very large portion of all hours worked is devoted to overhead tasks. Example of these roles are: managers, testers and analysts.

Historical data

To confirm the reliability of our data gathering technique we created a data set of historical data from a single module of the software product. This module, called STO, is a relatively small in comparison with the whole project. We chose

Date	Cumulative size (LOC)	Total debt (d)	Cumulative effort (h)
1/8/2019	0	12.61	0
1/29/2019	784	14.34	102.5
2/26/2019	784	14.4	107
3/28/2019	2058	17.26	206.5
4/10/2019	2116	17.46	323
5/20/2019	2116	17.46	408
6/20/2019	2154	17.46	408
7/31/2019	2159	18.67	513
8/30/2019	2158	17.46	769.5
9/30/2019	2398	18.17	1049.5
10/31/2019	2398	18.88	1539.5
1/16/2020	2943	18.29	2055.5
1/17/2020	3136	18.99	2058
1/22/2020	3136	18.99	2098
2/7/2020	3135	18.99	2397.5
2/13/2020	3809	19.02	2413.5

Table 5.2: Historical data of the STO module

STO as the size makes for an easier analysis. Furthermore, the module saw continuous maintenance over the past year, this allows us to establish multiple data points for each of our metrics.

We have collected data from in time-period ranging from the 1st of January 2019 until the 13th of February 2020. We analyzed a total of 16 source code snapshots from this time-period, an overview can be seen in Table 5.5.1. Developers working on this project logged their working hours on a daily basis. The interval between different data points varies, because we are limited to the available snapshots. We base our effort calculation on these timestamps. We calculated the total amount of hours spend by developers until the snapshot.

We also took the cumulative value of the code size, as we want to use the first data point as zero measurement. The initial size of STO on 1/8/2019 was 6432 lines of code. If we look at the data, we see the size increasing quite consistently. The same goes for TD, this almost directly related on to the size. This can be explained, as there were no TDM activities during this period.

In figure 5.1 we have plotted the cumulative size against the cumulative effort. As expected the size increases as more effort is put into development work. An important thing to note is that in reality the cumulative size lags behind the total effort. As effort is tracked on a daily bases, the source-code analysis is done on the current production version. This means that there is some time between the hours are registered and the code is analyzed after deployment. The second finding is the convergence of the two graphs. The amount of hours rise faster than the size, this can be explained by a drop in

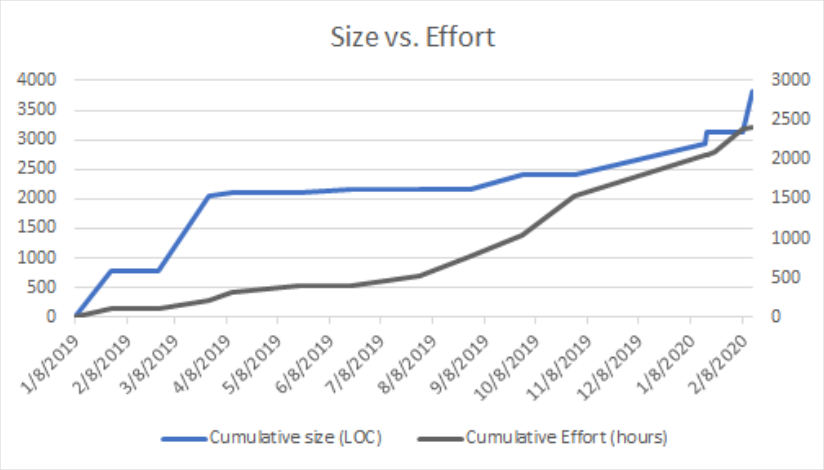

Figure 5.1: Comparison of the cumulative size versus cumulative effort

productivity. This happened in the third quarter of 2019. We consulted the Topicus about this phenomenon. Topicus hired a external party to increase the capacity of the team. However, as the new employees have no experience with the system, a loss in productivity is to be expected.

5.5.2 Finalizing the input variables

Before we can run the actual algorithm on the whole software product, we have to decide on the input variables. Namely: size, effort, debt, interest, productivity and quality. The variables are initialized based on the historical data of the previous section and a NDepent analysis on the whole code base.

Size

We ran a NDepent analysis with the rules of Appendix B on the 10th of March 2020. We take this date as the starting point of our estimation algorithm. The analysis showed us that the total size of the source code was 491366 LOC. We take this exact number as the initial size value for our algorithm.

Effort

Effort is a bit harder to determine, because the scale of this product, means a lot of developers work on the project. As we have seen in the previous section, even if we consider a single team, there is a high variance in the effort throughout the year. Based on talks with Topicus, we have initialized the available effort at 50 FTE. They are convinced this is a representative number for the average amount of developers. We assume that each moth a full time employee would

work 17 days, for 8 hours a day. This results in the nominal available effort of 6800 hours a month.

Debt and interest

For determining the TD and interest, we used the same analysis as for the size. Aggregating all individual TD items resulted in a total amount of 1109 days. Associated with this TD is an estimated monthly interest of 67,475 days. This translates to a monthly rate of 6,1%, i.e. 6,1% of the total TD is spent each month due to the existence of that debt.

Productivity and quality

The nominal productivity is based on the average productivity of the STO team. After more than a year of development, the average LOC per registered hour was 1,57. However, we also have to adjust for the preventive maintenance that has been conducted over this time period. We assume that the company stuck to their strategy of devoting 10% of the available effort to preventive maintenance. In that case the productivity averages at 1,75 LOC per hour. We use this as the base productivity level in our algorithm.

For the quality standard we also had to adjust for preventive effort. Doing this resulted in an estimated debt increase for STO of 393,51 hours. That is around 0,15 hours of TD for every line of newly introduced code.

5.5.3 Running the algorithm

Now we have determined all input variables, we can run the algorithm to look at the effects of the different allocation strategies. Each unit of time represents one month of maintenance work, the strategies are evaluated for a 12 and 60 months time-frame. We do this to highlight the short versus long term effects of individual strategies.

In order to keep things comprehensible, we demonstrate only 6 individual strategies in this section. This should give a good indication what type of strategy would be the best. Adding more strategies is relatively easy, but out of the scope of this proof of concept. Because our primary objective is to highlight the effects of preventive effort prioritization. The number of presented strategies have different levels of debt reducing aggressiveness and should suffice for reaching our goal. An overview of the selected strategies is presented in Table 5.2.

Short-term

Figure 5.3 shows the results of the 6 individual strategies plotted over a 12 month time period. We see a clear pattern, stategies which put a higher emphasis on TD reduction, perform considerabily worse in terms of total functional size. This is to be expected, as we already found in our literature review that TD can improve the short term productivity.

Strategy	Parameter	Remarks
Fixed	0	Base level with no preventive effort
Fixed	0.1	Current strategy used by the client
Fixed	0.5	50:50 preventive/perfective distribution
Relative	1	1:1 relative effort distribution
Relative	1/8	Strategy with aggressive TD prevention
Relative	2	Strategy with low priority on TD prevention

Figure 5.2: Demonstrated strategies

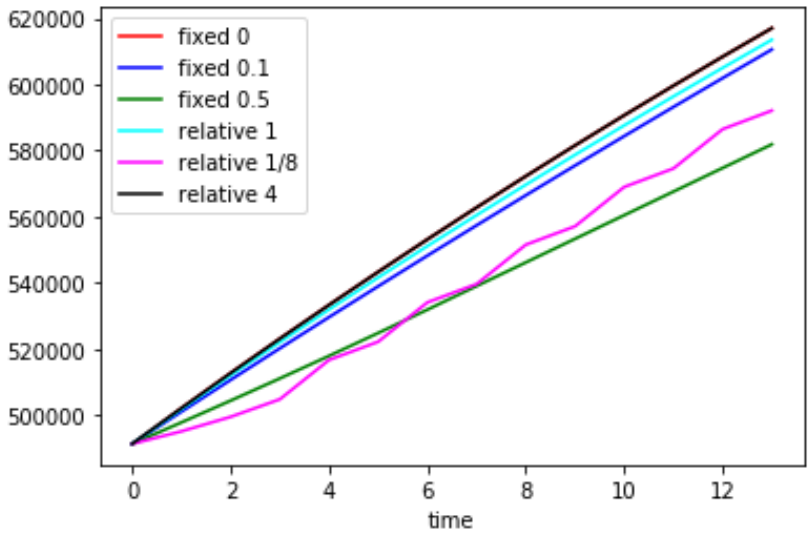

Figure 5.3: 12 month demonstration period

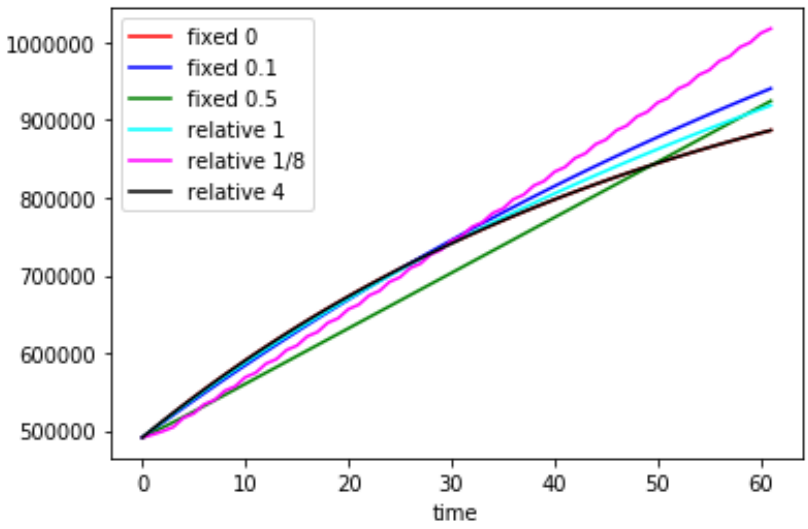

Figure 5.4: 60 month demonstration period

Another thing to note are the small differences between strategies: fixed 0, fixed 0.1, relative 1 and relative 4. This can be explained by current debt level of the product. The total amount of TD is relatively small compared to the total size of the application. Because of this, only little time is wasted due to TD, making these strategies perform quite similar as the devote little resources to preventive tasks.

Long-term

In order to get a better understanding of the long term effects on productivity, we have plotted the functional size for each strategy over a 5 year period. In Figure 5.4 we can see that over time, the differences between the individual strategies increase. Furthermore, we can see multiple intersections of the graphs, indicating the point in time when a given strategy becomes better performing over the other one.

As expected, strategies which put more emphasis on TD reduction, outperform when the time span increases. If we compare the best and the worst performing strategy, we see a large difference in the total functionality considering both strategies. The relative strategy with parameter 4 has an estimated total size of 883318 LOC after 60 months, where using parameter 1/8 results in 1017770 LOC. This amount translates to a difference in average productivity of about 15%. Moreover, we should also consider the total amount of TD present in the system at the end of the simulation. The relative 1/8 strategy is very agressive in reducing TD, this is portrait in the estimated debt at the end of period 60, which is 0. Whereas the relative 4 strategy has a final debt amount

of 8,521.12 days.

5.6 Summary

We first identified the problems Topicus is facing in relation to TDM. During the investigation of these problems we identified two goals: the need for validation of TDM strategies and improving long term productivity. We collected data on all the software and organizational attributes required for our algorithm. Using the historic data of the STO module, we validated the data gathering part of the treatment. After establishing all the input variables, we ran the algorithm in both a short and long term scenario, using multiple strategies. We found that the optimal strategy is heavily depended on the studied time-frame, proving that the right TDM strategy can have a substantial impact on the productivity.

Chapter 6

Evaluation

6.1 Implementation evaluation

In this section we discuss findings specific to the case study. So what do the results mean to Topicus, and how can future implementations be improved. We will first discuss the actual implementation process at Topicus. Then we will share some observations on the implementation outcome. And finally we will reflect on the requirements we set in chapter 4.

6.1.1 Implementation process

The first part of the implementation process was the most difficult and time consuming. This was the data gathering. While there is a lot of available data, it is hard to make sure it is reliable. Companies are complex structures and often do not match up to simulations. Employees are not a constant, nor are they homogeneous in terms of output. When gathering data, we have to take all these inconsistencies into account.

We chose the input variables based on historic data of a single team. This made it a lot easier to make consistent measurements, as we have a better overview of contextual factors which can affect the data. A disadvantage of this approach is a chance that the chosen team does not represent the rest of the organization.

The use of NDepend was straight forward, it allowed us to analyze multiple release versions of the same software component. Linking this to the hour registration software was very important in getting a better understanding of the maintenance process. It allowed us to see trends over time and get an insight on how the component developed in relation to the effort spend on it.

Once we have got all the input variables, the execution of the algorithm was very straightforward.

6.1.2 Observations

The algorithm gave some interesting insights for Topicus on how TD influences the maintenance process. First of all, the results highlight the actual need for preventive maintenance. Because productivity can deteriorate over time when a sub-optimal strategy is in place.

Secondly, we see that the actual strategy has a large influence on the performance of the team. In this case, aggressive TD reduction seems to be beneficial when considering longer time periods. It was also interesting that the current implemented strategy at Topicus was one of the better performing ones. This was the 10% fixed strategy, while Topicus did not have a good explanation why they chose 10% in particular, it does seem to be reasonable estimate for this specific project.

In terms of implementing a more mature version of the artifact in the company, multiple improvements can be made. First of all, it is important to determine the expected life time of the product. As we have seen there is a large variance in the optimal strategy depending on the examined time period. Secondly more effort should be spend on making sure the input variables are reliable and the estimations should match the actual values. By continuously monitoring these variables, the algorithm can be adjusted accordingly, to improve the reliability.

If the input variables are reliable and the estimations match up to the real world, than more strategies should be tested. And then a optimal strategy should be chosen.

6.1.3 Requirements

Together with Topicus, we evaluated the requirements that were made before the implementation.

Goal-level requirements

Our goal-level requirement stated that we want to enable managers to determine the optimal resource allocation strategy for a selected future point in time. While being simple and bare-bone, we consider our algorithm succeeded in delivering upon this promise. Multiple strategies can be easily added to the artifact and the examination periods can be changed. And the graphical output makes it possible to quickly compare different strategies.

Domain-level requirements

We formulated two domain-level requirements at the start of design phase. One required the algorithm to estimate the effects of allocation strategies on TD an size. The other stated the artifact should work for multiple different types of software projects.

The first requirement was definitely accomplished. As our artifact calculates the size and TD of multiple strategies. The second one is a bit harder to confirm.

As we only demonstrated the artifact using one project. However, we made sure the input variables are generalizable, making it considerably more easy to adjust the artifact for other projects.

6.1.4 Product-level requirements

Most of the product-level requirements were established to specify the mechanics of software maintenance and TD. We accomplished to simulate the change in TD, the productivity penalty of TD, newly introduced TD and the change in functionality.

The remaining product-level requirements referred to different types of strategies that should be implemented in the artifact. We successfully implemented both types of strategies, namely: fixed and relative allocation strategies.

6.1.5 Design-level requirement

The design-level requirements were constructed to determine how the user should interact with the artifact. The first requirement of this section denoted that the user should see a visualization of multiple strategies. Using the mathplotlib library of python, we successfully implemented this requirement.

The second requirement was the ability for users to give the input variables before running the algorithm. This requirement has only be partially implemented. As the user is only able to set the input variables by changing the source-code. However, this takes very little effort to update for future versions of the artifact.

6.2 Research execution

After the implementation we organized a workshop at the client in order to evaluate the method. During this workshop the researcher presented the preliminary results of the study and the outcomes of the implementation itself. After which there was an open discussion about the implementation and the method in general. For this workshop we invited a diverse group of employees, which were all considered stakeholders in the project. After the workshop, the participants were asked to fill out a questionnaire. This questionnaire was based on unified theory of acceptance and use of technology (UTAUT) by Venkatesh et al. [57]. We chose this model as it is widely used method to determine the acceptance of new technology.

The UTAUT model is a combination of multiple acceptance models. The model itself contains 4 key variables that affect the behavioral intention and the use behavior of a system. These 4 variables are: Performance Expectancy, Effort Expectancy, Social Influences, Facilitating Conditions. The questions in the questionnaire are also categorized based on these 4 variables. We used all the relevant questions of the original work with some minor changes to better

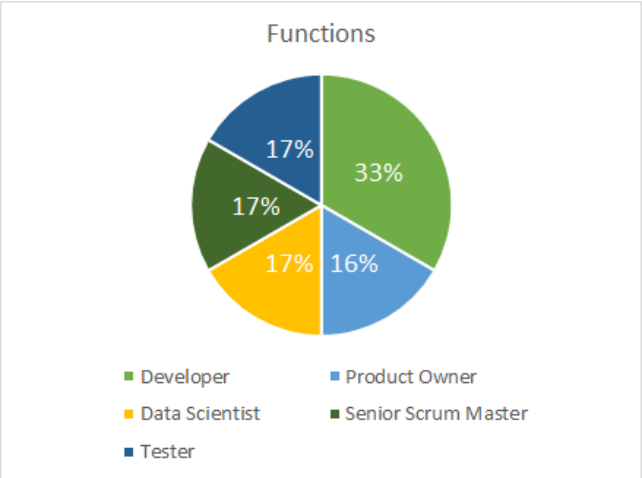

Figure 6.1: Different functions of questionnaire participants

suit this research project. All the questions can be found in Appendix C. We will discuss the results of the questionnaire in the next section.

6.3 Data analysis

6.3.1 Questionnaire

The workshop was attended by 7 employees of Topicus, of which 6 filled in the questionnaire. While the number is quite low, we still think this group of employees gives an accurate representation of the company. The participates had a wide variety of different functions and were spread across different teams within the same division, as can be seen in Figure 6.1. The average age of the participants is quite young, with all of them being under 40 years of age, seen in Figure 6.2. This should not be a problem, as this reflects the average age of the company. However, this can influence the results, because age is seen as a mediating variable for all key constructs. So the lower age can positively affect both the behavioral intention and use behavior.

Performance expectancy

The questions in this part relate to the expected performance gains users will have when using the system, results can be found in figure 6.3. We removed the question: "If I use the system, I will increase my chances of getting a raise." from the original questionnaire, as chances of getting a raise by using the system is irrelevant in this specific company. The original questionnaire spoke only of personal productivity, we added an extra question to include team productivity as well. Q4: "Using the method increases the productivity of my team." We

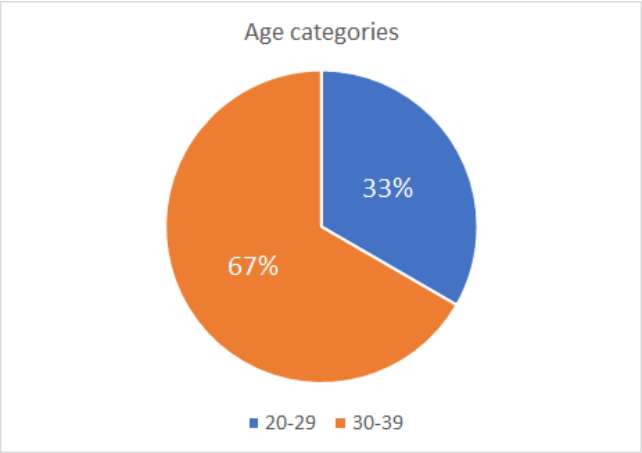

Figure 6.2: Age categories of questionnaire participants

did this because improving team productivity is one of the main goals of the system and therefore part of performance expectancy. The majority of the participants agreed with Q1 and Q4, with nobody disagreeing. Indicating the participants found the method to be useful and expect that it can help boost the teams' productivity. The other two questions were related to completing tasks more quickly and personal productivity, which have a mixed result. This is also evident in figure 6.7 were we can observe a relative high standard deviation for Q3 in particular. This might be explained by the difference in functions of the participants. As the method would likely mean that a employee in a management position has to do additional work, while developers might get a boost in productivity by the more efficient resource allocation the method provides.

Effort expectancy

When a new system of method requires a lot of effort to learn or operate, it might withhold users from using the system, results can be found in figure 6.4. In that regard, the results from this section are very positive, 83.33% of the participants agreed or strongly agreed with question 6 and 7. Furthermore, all participants agreed or strongly agreed with question 8. Only question 5 has some mixed results. This question check whether the interaction with the method is clear and understandable. As the presented method is very basic with no user interface, we do not think this would be an issue with the final product. One thing to note is that the experience of the users moderates the effect effort expectancy has on the behavioral intention of the system, all of the participants have a technical background. However, we do not think this is a problem, as the method primarily meant for use in a technical context.

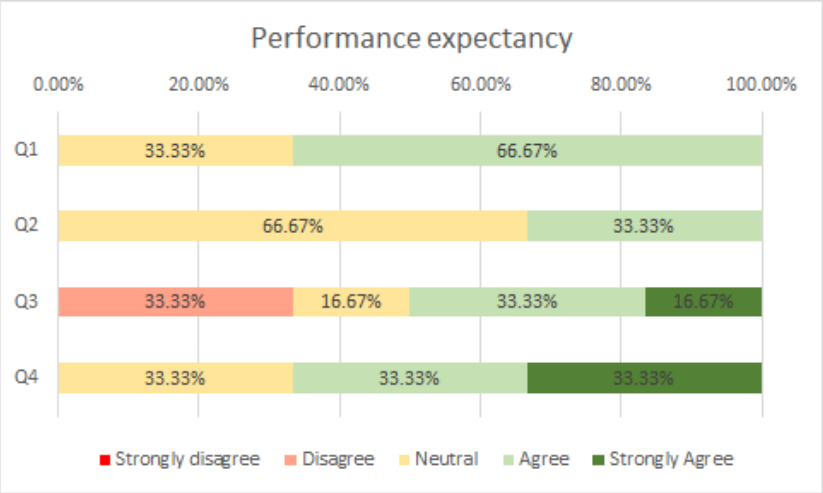

Figure 6.3: Results on performance expectancy

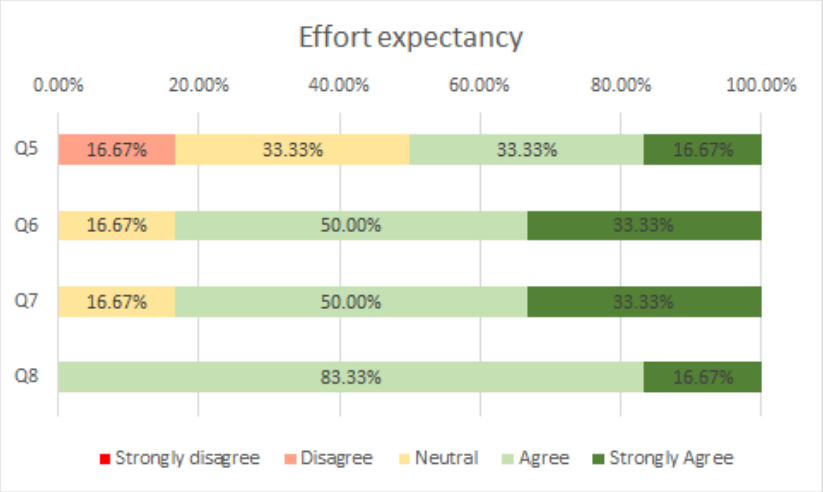

Figure 6.4: Results on effort expectancy

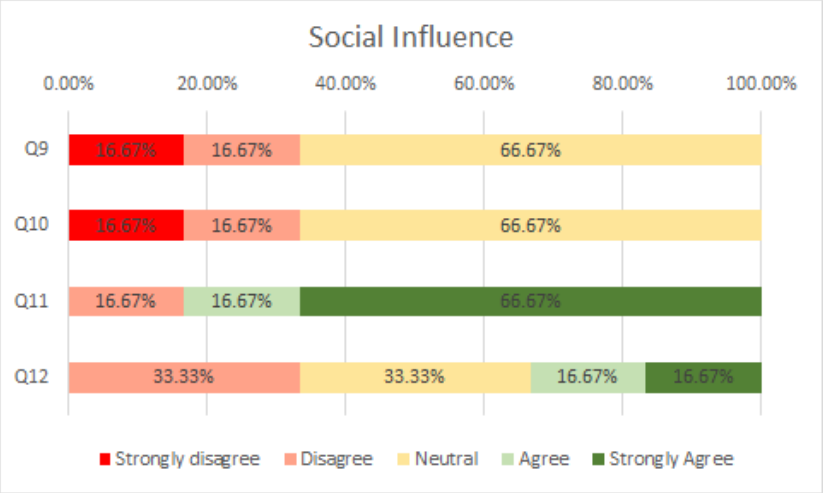

Figure 6.5: Results on social influence

Social influence

In this part we look at the social influences from inside and outside of the organization, which affect the behavioral intention, results can be found in figure 6.5. Only question 11 has a positive result. For question 9 and 10 we see that the majority of participants have a neutral opinion and the others disagree. Question 12 has some mixed results. This indicates that the participants do not think they are really influenced by other to use the method. However, they do think the senior management is helpful in regards to using the method.

Facilitating conditions

Overall the participants tend to agree with the facilitating conditions, results can be found in figure 6.6. It is important to note that disagreement with question 15 is actually a positive outcome. Only question 13 seems to have mixed results, indicating that the available resources for the method might become a problem. Compatibility with other systems also does not seem to be an issue, this is likely due to the design choice of making the method independent form specific programming languages.

6.4 Summary and conclusions

After implementation we can conclude that the most critical success factor is the ability to gather reliable data, as the results of the method are solely based on the input variables. This is not an easy process as some input variables are hard to meassure, tools such as NDepend can help in this process. While there

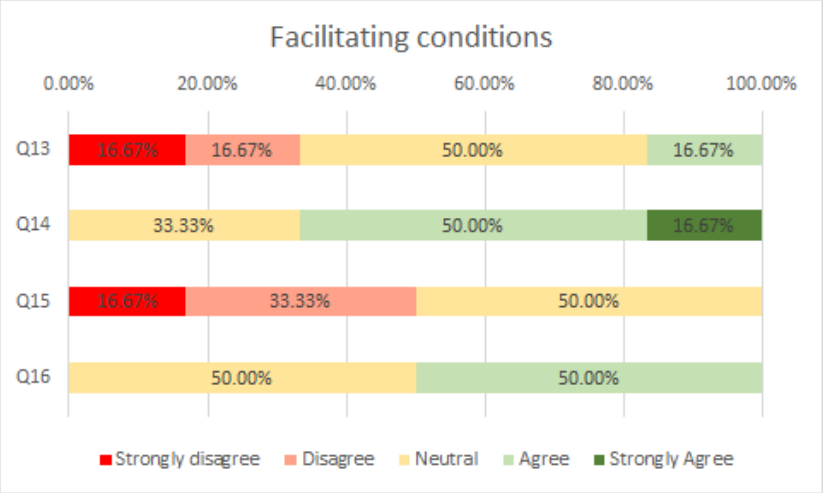

Figure 6.6: Results on facilitating conditions

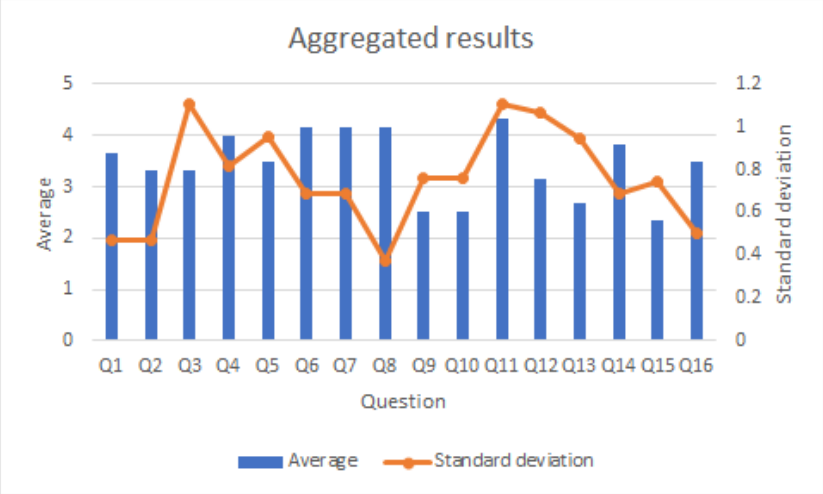

Figure 6.7: Results overview including standard deviation

is still a lot of room for improving the method, based on the requirements we made beforehand, our method has the intended behavior.

The outcome of the method is not the only part in successfully implementing a new technology. Therefore, we also investigated the user acceptance of this method during a workshop. Overall we think the results of the workshop and questionnaire are very promising. Especially the positive scores on performance expectancy and effort expectancy indicate that the method would be valuable and requires little effort to learn. We can still improve on social influence, but since the method is still far from an end product, we do not think this would become a problem in the future, as long as all stakeholders will be involved in the process. As for the facilitating conditions, more resources would benefit the future adoption of the method. This should arguably not that difficult to improve, as successful implementation of the method should actually save resources in the long run.

Chapter 7

Conclusions

7.1 Conclusion

In this chapter we first conclude the results per research question This is followed by the contributions we made; both to practitioners as literature. We finalize with the limitations of this research and suggestions for future work.

RQ 1: What is the state-of-art literature regarding software maintenance?

Software maintenance literature exists for some time now. The core activities of software maintenance have not changed over the years. Also, the fundamental goal of software maintenance remained the same, namely extending the life time of a software product. A general trend in the field is the increase in the average lifespan of software. This introduces new challenges and results in software maintenance becoming an ever more important component in the software lifecycle. One goal of this research question was to find models for classifying software maintenance. Multiple classifications have been proposed in literature. The difference between these lie predominantly in the granularity of activity types.

RQ 2: What is the state-of-art literature regarding technical debt?

Compared to software maintenance, the concept of TD was introduced in scientific literature more recently. In recent years it gained substantially more interest from scholars. There is a consensus in literature that TD can be broken down in two main components: the principal and the interest of the debt. It is only possible to calculate the exact costs when a debt is fully repaid. As both the principal and interest can change over time, current TD is always an estimation of costs. We found many different subtypes of TD in literature. These can help differentiate different types of debt. Furthermore, we found that sometimes TD is incurred on purpose, often with the goal of accelerating a feature's

the time to market. Multiple tools exist to manage TD, these tools (automatically) track TD of a project over time. Automated TD tools rely on the source code for TD estimations. These estimations are based on code hygiene and best practices. TD types which cannot be derived from the source code can only be tracked by hand, however, this can be very time consuming. Finally we found that TD is most often measured in time, which in turn can be used to calculate the financial metric.

RQ 3: What is the state-of-art literature regarding productivity?

Since the beginning of the software engineering field, researchers have tried to determine the productivity of developers. Similar as in economics, productivity is a function of input and output, over a period of time. For the input we can consider all the resources consumed by the organization during "production". Software engineering is very labor intensive and does not consume raw materials unlike traditional manufacturing processes. Therefore, literature considers labor costs as the only resource of production. We found that the measurement of the output is more often a point of debate among scholars, as it is sometimes already hard to determine what is actually produced, let alone to measure it. The most common method is to use a size measurement for the amount of code that is changed or produced. The advantage of using size metrics is that they are relatively easy to compute and are quite consistent, making them suitable for tracking trends within single projects. The disadvantage of size metrics is that they are highly dependent on the project type and external attributes. This makes using only size metrics to compare different projects unreliable and therefore undesirable. Both Logical Lines of Code (LLoC) and Functions Points (FP) are often used as size metrics in literature. Both take the complexity of the written code in consideration, resulting in less variance of the measurement due to the way of working of the individual software engineer.

RQ 4: What is the effect of software maintenance effort on technical debt?

Inherently, software maintenance has an effect on technical debt, as changes made to the source code also change the amount TD of the project. The effect differs depending on the maintenance activity that is being performed. Both perfective and adaptive maintenance tasks add additional functionality to the project. Depending on the quality of the newly added code, the total amount of TD remains the same or increases. Corrective maintenance can both reduce and increase TD, while preventive maintenance has the specific goal of reducing TD.

RQ 5: What is the effect of technical debt on productivity?

We found multiple studies that investigated the effects of TD in the field of software engineering. Most of them reported negative effects on the overall

productivity due the consequences of TD, both by direct and indirect causes. In some specific cases TD can improve productivity, however, it is important to note this is only possible when we consider productivity over a short time period. When TD is not incurred strategically, we can assume it lowers long term productivity by decreasing the overall system quality. This results in higher maintenance costs.

RQ 6: How can we model and measure the effects of software maintenance strategies on technical debt and productivity?

We constructed a model to better understand how TDM strategies influence TD and productivity. Maintenance effort has a certain effect on the amount of TD that is in the system. TDM strategies determine how the maintenance effort is allocated, i.e. how much effort is spent on each type of maintenance activity. We used this as a basis for our allocation process model. The model is split in three parts: before, during and after each sprint. The before part refers to the software and contextual attributes that can be seen as the input for what is going to happen during the sprint. These attributes are: TD, size, team capacity, productivity loss and allocation strategy. During the sprint effort is spent on different types of maintenance activities, these activities are classified as: preventive, perfective and other. After the sprint we measure the TD and Size again. We can then compare this to the values from the start of the sprint to estimate the effects of the strategy.

RQ 7: How to design an algorithm that approximates the optimal software maintenance strategy for a given project?

Together with Topicus we established a set of requirements for a proof of concept. We designed an algorithm based on the resource allocation process model that would fulfill these requirements. The algorithm estimates the amount of TD and the size after one sprint based on all the input variables. This result is then used as the input for the next sprint cycle. This is done for multiple strategies, which we have also defined. The end results are plotted in a graph, allowing the user to easily compare the performance of the strategies based on a timeframe of choosing.

7.2 Contributions

In this research we made contributions to both practitioners as to literature. Findings of the implementation can help practitioners make better business decisions while our contributions to literature can help improve the general knowledge of TD.

7.2.1 Contributions to practitioners

During our literature study we found that many researchers concluded that TD has a deterrent effect on productivity. Our research confirms this, but more importantly, it makes the productivity loss more tangible by visualizing the estimated growth of the application for multiple strategies. This allows practitioners to align their TDM strategy with the long term business goals. We created the algorithm to be as generic as possible, as this allows practitioners to implement it more easily regardless of the measurements used. This makes it also a relatively inexpensive method to implement, as there is no need for proprietary software. A business analyst would have to spend some time gathering data, after which the results of the method can be used by the project manager for decision making.

7.2.2 Contributions to literature

When we started this research, no studies in which allocation strategies regarding TD were investigated in real world cases were known to the researchers. Some studies tried simulating the effect of allocation strategies based on hypothetical data. This study improves on that by conducting a comparable simulation in a real word case. Our results are very promising, implying that high level strategy can make a substantial difference in the quality and effort required for a piece of software.

7.3 Limitations and Future work

First we discuss the limitations regarding the design and implementation of the algorithm Subsequently, we discuss the limitations of the evaluation. Where applicable, we give our thought on how to mitigate these limitations in future work.

7.3.1 Design limitations

In the current version of the algorithm, the estimated interest of all technical debt items is oversimplified. All TD items are aggregated and used to calculate a single interest percentage, while in reality each item can have a different amount of estimated interest. Given our limited resources, we based this single percentage on historical data. One way to mitigate this in the future would be to have a separate interest percentage per debt category or ideally per item. However, this would also seriously make the algorithm more complex, as the reduction in debt by preventive effort would also take these different percentages in mind.

A second limitation is the reliance on automated debt analysis. While tools such as NDepend can give a good estimation with little effort, not all debt can be found this way. It is hard to estimate these other types of debt. It would be

interesting to see if this would have an impact on the most optimal strategy, if these other types would be included.

7.3.2 Evaluation limitations

We first evaluated the artifact based on the initial requirements. Because this was done by the researcher and the company, there is an obvious threat to result in a biased view. But we are confident that the artifact would also function in different organizations. The artifact itself is easy to implement, the challenge lies in gathering reliable data to supply the algorithm with. Having an organization where this data is not easily accessible would make a successful implementation a lot more challenging.

The results for the workshop and questionnaire were predominantly positive. However, this form of evaluation has some limitations. First of all the participants of the workshop were given a demonstration. Therefor they were unable to actually use the artifact themselves. This meant that the acceptance is only based on perceived use of the algorithm. Furthermore, the amount of employees which participated was quite low, which is an obvious validity threat. In future work, the group of participants should be larger.

While we evaluated the artifact based on requirements and acceptance, we did not validate the actual estimations. This is a limitation we had to make due to limited resources. In future work this should also be evaluated, but this can be very time consuming as the evaluation period of the algorithm is at least a few months.

Bibliography

[1] Zuriani Hayati Abdullah, Jamaiah H. Yahaya, Zulkefli Mansor, and Aziz Deraman. Software Ageing Prevention from Software Maintenance Perspective – A Review. *Journal of Telecommunication, Electronic and Computer Engineering*, 9(3-4 Special Issue):93–96, 2017.

[2] Muhammad Ovais Ahmad, Pasi Kuvaja, Markku Oivo, and Jouni Markkula. Transition of software maintenance teams from scrum to KanbanAhmad, M. O., Kuvaja, P., Oivo, M., & Markkula, J. (2016). Transition of software maintenance teams from scrum to Kanban. Proceedings of the Annual Hawaii International Conference on System Scie. *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2016-March(October 2017):5427–5436, 2016.

[3] Reem Alfayez and Barry Boehm. An Exploratory Study on the Influence of Developers in. pages 1–10, 2018.

[4] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. De Mendonça, Rodrigo O. Spinola, Forrest Shull, and Carolyn Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[5] Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, and Rodrigo O. Spínola. Towards an ontology of terms on technical debt. *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*, pages 1–7, 2014.

[6] T Amanatidis, A Chatzigeorgiou, and A Ampatzoglou. The relation between technical debt and corrective maintenance in PHP web applications. *Information and Software Technology*, 90:70–74, 2017.

[7] Alain April, Jane Huffman Hayes, Alain Abran, and Reiner Dumke. Software Maintenance Maturity Model: the software maintenance process model: Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(3):197–223, 2005.

[8] Gabriele Bavota and Barbara Russo. A Large-Scale Empirical Study on Self-Admitted Technical Debt. (May 2016), 2017.

[9] Terese Besker and Jan Bosch. Technical Debt Cripples Software Developer Productivity - A longitudinal study on developers' daily software development work Terese. 2018.

[10] Terese Besker, Antonio Martini, and Jan Bosch. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, 135:1–16, 2018.

[11] Terese Besker, Antonio Martini, and Jan Bosch. Technical debt cripples software developer productivity. (May):105–114, 2018.

[12] Terese Besker, Antonio Martini, and Jan Bosch. The Journal of Systems and Software Software developer productivity loss due to technical debt — A replication and extension study examining developers ' development work. 156:41–61, 2019.

[13] Cor-Paul Bezemer and Andy Zaidman. Multi-Tenant SaaS Applications : Maintenance Dream or Nightmare ? *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92, 2010.

[14] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. *Proceedings - International Conference on Software Engineering*, pages 419–429, 2012.

[15] Stamatia Bibi, Apostolos Ampatzoglou, and Ioannis Stamelos. A Bayesian Belief Network for Modeling Open Source Software Maintenance Productivity. In Kevin Crowston, Imed Hammouda, Björn Lundell, Gregorio Robles, Jonas Gamalielsson, and Juho Lindman, editors, *IFIP Advances in Information and Communication Technology*, volume 472 of *IFIP Advances in Information and Communication Technology*, pages 32–44. Springer International Publishing, Cham, 2016.

[16] Ned Chapin, Joanne E Hale amd Khaled Md. Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal on Software Maintenance and Evolution: Research Practice*, 13(July 2000):3–30, 2003.

[17] Ward Cunningham. The WyCash portfolio management system. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) - OOPSLA '92*, number October, pages 29–30, New York, New York, USA, 1992. ACM Press.

[18] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. How Do Developers Fix Issues and Pay Back Technical Debt in the Apache Ecosystem ? pages 153–163, 2018.

[19] Georgios Digkas, Mircea Lungu, Alexander Chatzigeorgiou, and Paris Avgeriou. The evolution of technical debt in the apache ecosystem. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10475 LNCS:51–66, 2017.

[20] Dana Edberg, Polina Ivanova, and William Kuechler. Methodology mashups: An exploration of processes used to maintain software. *Journal of Management Information Systems*, 28(4):271–304, 2012.

[21] Davide Falessi and Andreas Reichel. Towards an open-source tool for measuring and visualizing the interest of technical debt. *2015 IEEE 7th International Workshop on Managing Technical Debt, MTD 2015 - Proceedings*, pages 1–8, 2015.

[22] Brian Fitzgerald and Klaas Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123(October 2018):176–189, 2017.

[23] Eduardo Ferreira Franco, Joaquim Rocha, Hamilton Carvalho, Martins Marcelo, and Kechi Hirama. An Analysis of Technical Debt Management Through Resources Allocation Policies in Software Maintenance Process. pages 1–17, 2016.

[24] Nishant Grover, Jyotsna Saxena, and Vikas Sihag. Proceedings of the International Conference on Data Engineering and Communication Technology. 469:603–611, 2017.

[25] Yuepu Guo and Carolyn Seaman. A portfolio approach to technical debt management. page 31, 2011.

[26] Mik Kersten. What Flows through a Software Value Stream ? 2018.

[27] Barbara Kitchenham and S Charters. Procedures for Performing Systematic Literature Reviews in Software Engineering. *Keele University & Durham University, UK*, 2007.

[28] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Information and Software Technology*, 51(1):7–15, 2009.

[29] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[30] Soren Lauesen. *Software requirements: styles and techniques*. Pearson Education, 2002.

[31] Luigi Lavazza, Sandro Morasca, and Davide Tosi. A Method to Optimize Technical Debt Management in Timed-boxed Processes. (c):45–51, 2018.

[32] Luigi Lavazza, Sandro Morasca, and Davide Tosi. Technical debt as an external software attribute. *Proceedings - International Conference on Software Engineering*, pages 21–30, 2018.

[33] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[34] E Ma. *SWEBOK 3.0 The Guide to the Software Engineering Body of Knowledge*. 2013.

[35] Everton S Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An Empirical Study On the Removal of Self-Admitted Technical Debt. 2017.

[36] Harlan D Mills. Software development. *Developments in Mathematics*, 35(Dc):35–91, 2016.

[37] Alessandro Murgia, Giulio Concas, Roberto Tonelli, Marco Ortu, Serge Demeyer, and Michele Marchesi. On the influence of maintenance activity types on the issue resolution time. *ACM International Conference Proceeding Series*, pages 12–21, 2014.

[38] Woubshet Nema, Pilar Rodríguez, and Markku Oivo. Analyzing the concept of technical debt in the context of agile software development : A systematic literature review. 82:139–158, 2017.

[39] Vu Nguyen, Barry Boehm, and Phongphan Danphitsanuphan. A controlled experiment in assessing and estimating software maintenance tasks. *Information and Software Technology*, 53(6):682–691, 2011.

[40] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. An empirical model of technical debt and interest. page 1, 2011.

[41] Pentium Pcs and Windows Nt. Contents of the specification.

[42] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77, 2008.

[43] Dan Port and Bill Taber. An empirical study of process policies and metrics to manage productivity and quality for maintenance of critical software systems at the jet propulsion laboratory. *International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[44] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pages 91–100, 2014.

[45] Narayan Ramasubbu and Chris F. Kemerer. Towards a model for optimizing technical debt in software products. *2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings*, pages 51–54, 2013.

[46] Narayan Ramasubbu and Chris F. Kemerer. Managing technical debt in enterprise software packages. *IEEE Transactions on Software Engineering*, 40(8):758–772, 2014.

[47] Narayan Ramasubbu and Chris F Kemerer. Managing Technical Debt in Enterprise Software Packages. 40(8):758–772, 2014.

[48] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102(May):117–145, 2018.

[49] Nicolli Rios, Rodrigo Oliveira Spínola, Manoel Mendonça, and Carolyn Seaman. The most common causes and effects of technical debt: First results from a global family of industrial surveys. *International Symposium on Empirical Software Engineering and Measurement*, (October):1–10, 2018.

[50] Babak Darvish Rouhani, Mohd Na Z.Ri Mahrin, Fatemeh Nikpay, Rodina Binti Ahmad, and Pourya Nikfard. A systematic literature review on Enterprise Architecture Implementation Methodologies. *Information and Software Technology*, 62(1):1–20, 2015.

[51] Carolyn Seaman and Yuepu Guo. Measuring and Monitoring Technical Debt. *Advances in Computers*, 82:25–46, 2011.

[52] Harry M. Sneed and Wolfgang Prentner. Analyzing data on software evolution processes. *Proceedings - 26th International Workshop on Software Measurement, IWSM 2016 and the 11th International Conference on Software Process and Product Measurement, Mensura 2016*, pages 1–10, 2017.

[53] Rodrigo Oliveira Spínola, Salvador Brazil, Salvador Brazil, Salvador Brazil, and Carolyn Seaman. Supporting Analysis of Technical Debt Causes and Effects with Cross-Company Probabilistic Cause-Effect Diagrams. 2019.

[54] Tejaswini, Srushti Patil, Suman Salimath, Venuprasad Naik, Ritam Nandi, Mahesh S. Patil, Indira Bidari, and Satyadhyan Chickerur. Programmer Productivity Analyzer Tool. In *2017 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC 2017*, pages 1–8. IEEE, dec 2018.

[55] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. Continuous Delivery Practices in a Large Financial Organization. 2016.

[56] John Venable, Jan Pries-Heje, and Richard Baskerville. A Comprehensive Framework for Evaluation in Design Science Research: Advances in Theory and Practice (DESRIST 2012). *Proceedings of the 7th International Conference on Design Science Research in Information Systems*, pages 423–438, 2012.

[57] Viswanath Venkatesh, Michael G Morris, Gordon B Davis, and Fred D Davis. User acceptance of information technology: Toward a unified view. *MIS quarterly*, pages 425–478, 2003.

[58] Roel Wieringa and Ayşe Moralı. Technical Action Research as a Validation Method in Information Systems Design Science\nDesign Science Research in Information Systems. Advances in Theory and Practice. 7286:220–238, 2012.

[59] Roel J. Wieringa. Design science methodology: For information systems and software engineering. *Design Science Methodology: For Information Systems and Software Engineering*, pages 1–332, 2014.

[60] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *ACM International Conference Proceeding Series*, 2014.

[61] Hong Wu, Lin Shi, Celia Chen, Qing Wang, and Barry Boehm. Maintenance effort estimation for open source software: A systematic literature review. *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, pages 32–43, 2017.

Appendix A

Source-code

```
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

size = 491322   # initial size in logical lines of code
available_effort = 6400 # hours of development time per timeframe
technical_debt = 8872 # initial size of the debt in hours
monthly_interest = 0.06 #monthly interest percentage
quality_standard = 0.2 # hours of TD for every new line of code produced
nominal_productivity = 1.5 # average hourly output of developers in LLOC

#Create dataframe with time, debt and size as columns
def new_empty_df():
new_df = pd.DataFrame({'time': 0,
        'debt': technical_debt,
        'size': size}, index=[0] )
return new_df

def new_debt_size(td, size, preventive_factor, perfective_factor):

actual_effort = available_effort - (td * monthly_interest)

if actual_effort >= 0:

size_increase = actual_effort * perfective_factor * nominal_productivity
debt_increase = size_increase * quality_standard
debt_decrease = actual_effort * preventive_factor
new_debt = td + debt_increase - debt_decrease
if new_debt < 0:
new_debt = 0
```

```
new_size = size + size_increase
return (new_debt, new_size)
else:
return(td, size)


def update_df(df, strategy, factor):
old_time = df.iloc[-1,0]
old_debt = df.iloc[-1,1]
old_size = df.iloc[-1,2]

if strategy == 'fixed':
allocation = strategy_fixed(factor)
if strategy == 'linear':
allocation = strategy_smart(available_effort, old_debt, old_size, 1)
if strategy == 'relative':
allocation = strategy_smart(available_effort, old_debt, old_size, factor)

t = new_debt_size(old_debt, old_size, allocation[0], allocation[1])
new_time = old_time + 1
new_debt = t[0]
new_size = t[1]

dictionary = {'time': new_time,'debt': new_debt,'size': new_size}
df = df.append(dictionary, ignore_index=True)
return df


def calc_debt(df, timeframe, strategy, factor):
while df.iloc[-1,0] <= timeframe:
df = update_df(df, strategy, factor)
return df

def strategy_fixed(preventive_factor = 0.1):

preventive_effort = preventive_factor
perfective_effort = 1 - preventive_factor

return(preventive_effort, perfective_effort)

def strategy_linear(available_effort, td, size):
if td != 0:
relative_debt = td / (size / nominal_productivity)
else:
relative_debt = 0
```

```
preventive_effort = relative_debt
perfective_effort = 1 - preventive_effort

return(preventive_effort, perfective_effort)

def strategy_smart(available_effort, td, size, factor = 0.5):
if td!=0:
relative_debt = td / (size / nominal_productivity)
else:
relative_debt = 0
preventive_effort = (relative_debt ** factor)
perfective_effort = 1 - preventive_effort
return(preventive_effort, perfective_effort)
```

Appendix B

NDepend debt rules

```
Code Smells:

Avoid types too big
Avoid types with too many methods
Avoid types with too many fields
Avoid methods too big, too complex
Avoid methods with too many parameters
Avoid methods with too many overloads
Avoid methods potentially poorly commented
Avoid types with poor cohesion
Avoid methods with too many local variables


Object Oriented Design:

Avoid interfaces too big
Base class should not use derivatives
Class shouldn't be too deep in inheritance tree
Class with no descendant should be sealed if possible
Overrides of Method() should call base.Method()
Do not hide base class methods
A stateless class or structure might be turned into a static type
Non-static classes should be instantiated or turned to static
Methods should be declared static if possible
Constructor should not call a virtual method
Avoid the Singleton pattern
Don't assign static fields from instance methods
Avoid empty interfaces
Avoid types initialization cycles

Design:
```

Avoid custom delegates
Types with disposable instance fields must be disposable
Disposable types with unmanaged resources should declare finalizer
Methods that create disposable object(s) and that don't call Dispose()
Classes that are candidate to be turned into structures
Avoid namespaces with few types
Nested types should not be visible
Declare types in namespaces
Empty static constructor can be discarded
Instances size shouldn't be too big
Attribute classes should be sealed
Don't use obsolete types, methods or fields
Do implement methods that throw NotImplementedException
Override equals and operator equals on value types
Boxing/unboxing should be avoided

Architecture:

Avoid namespaces mutually dependent
Avoid namespaces dependency cycles
Avoid partitioning the code base through many small library Assemblies
UI layer shouldn't use directly DB types
UI layer shouldn't use directly DAL layer
Assemblies with poor cohesion (RelationalCohesion)
Namespaces with poor cohesion (RelationalCohesion)
Assemblies that don't satisfy the Abstractness/Instability principle
Higher cohesion - lower coupling
Avoid mutually-dependent types
Example of custom rule to check for dependency

API Breaking Changes:

API Breaking Changes: Types
API Breaking Changes: Methods
API Breaking Changes: Fields
API Breaking Changes: Interfaces and Abstract Classes
Broken serializable types
Avoid changing enumerations Flags status
API: New publicly visible types
API: New publicly visible methods
API: New publicly visible fields

Dead Code:

Potentially Dead Types

Potentially Dead Methods
Potentially Dead Fields
Wrong usage of IsNotDeadCodeAttribute

Security:

Don't use CoSetProxyBlanket and CoInitializeSecurity
Don't use System.Random for security purposes
Don't use DES/3DES weak cipher algorithms
Don't disable certificate validation
Review publicly visible event handlers
Pointers should not be publicly visible
Seal methods that satisfy non-public interfaces
Review commands vulnerable to SQL injection
Review data adapters vulnerable to SQL injection

Visibility:

Methods that could have a lower visibility
Types that could have a lower visibility
Fields that could have a lower visibility
Types that could be declared as private, nested in a parent type
Avoid publicly visible constant fields
Fields should be declared as private
Constructors of abstract classes should be declared as protected or private
Avoid public methods not publicly visible
Event handler methods should be declared as private or protected
Wrong usage of CannotDecreaseVisibilityAttribute
Methods that should be declared as 'public' in C\#, 'Public' in VB.NET

Immutability:

Fields should be marked as ReadOnly when possible
Avoid non-readonly static fields
Avoid static fields with a mutable field type
Structures should be immutable
Property Getters should be immutable
A field must not be assigned from outside its parent hierarchy types
Don't assign a field from many methods
Do not declare read only mutable reference types
Array fields should not be read only
Types tagged with ImmutableAttribute must be immutable
Types immutable should be tagged with ImmutableAttribute
Methods tagged with PureAttribute must be pure
Pure methods should be tagged with PureAttribute

Naming Conventions:

Instance fields naming convention
Static fields naming convention
Interface name should begin with a 'I'
Abstract base class should be suffixed with 'Base'
Exception class name should be suffixed with 'Exception'
Attribute class name should be suffixed with 'Attribute'
Types name should begin with an Upper character
Methods name should begin with an Upper character
Do not name enum values 'Reserved'
Avoid types with name too long
Avoid methods with name too long
Avoid fields with name too long
Avoid having different types with same name
Avoid prefixing type name with parent namespace name
Avoid naming types and namespaces with the same identifier
Don't call your method Dispose
Methods prefixed with 'Try' should return a boolean
Properties and fields that represent a collection of items should be named Items.
DDD ubiquitous language check
Avoid fields with same name in class hierarchy
Avoid various capitalizations for method name

Source Files Organization:

Avoid referencing source file out of Visual Studio project directory
Avoid duplicating a type definition across assemblies
Avoid defining multiple types in a source file
Namespace name should correspond to file location
Types with source files stored in the same directory, should be declared in the same namespa
Types declared in the same namespace, should have their source files stored in the same dire

System specific (.NET FRAMEWORK USAGE):

Mark ISerializable types with SerializableAttribute
Mark assemblies with CLSCompliant (deprecated)
Mark assemblies with ComVisible (deprecated)
Mark attributes with AttributeUsageAttribute
Remove calls to GC.Collect()
Don't call GC.Collect() without calling GC.WaitForPendingFinalizers()
Enum Storage should be Int32
Do not raise too general exception types
Do not raise reserved exception types
Uri fields should be of type System.Uri

```
Types should not extend System.ApplicationException
Don't Implement ICloneable

System.Collections:

Collection properties should be read only
Don't use .NET 1.x HashTable and ArrayList (deprecated)
Caution with List.Contains()
Prefer return collection abstraction instead of implementation

System.Runtime.InteropServices

P/Invokes should be static and not be publicly visible
Move P/Invokes to NativeMethods class
NativeMethods class should be static and internal

System.Threading

Don't create threads explicitly
Don't use dangerous threading methods
Monitor TryEnter/Exit must be both called within the same method
ReaderWriterLock AcquireLock/ReleaseLock must be both called within the same method
Don't tag instance fields with ThreadStaticAttribute
Method non-synchronized that read mutable states

System.Xml:

Method should not return concrete XmlNode
Types should not extend System.Xml.XmlDocument

System.Globalization:

Float and Date Parsing must be culture aware

System.Reflection:

Mark assemblies with assembly version
Assemblies should have the same version

Microsoft.Contracts:

Public methods returning a reference needs a contract to ensure that a non-null reference is
```

Appendix C

Questionnaire

Demographic questions

- D1: Age of the participant
- D2: Function of participant

Performance expectancy

- Q1: I would find the method useful in my job.
- Q2: Using the method enables me to accomplish tasks more quickly.
- Q3: Using the method increases my personal productivity.
- Q4: Using the method increases the productivity of my team.

Effort expectancy

- Q5: My interaction with the method would be clear and understandable.
- Q6: It would be easy for me to become skillful at using the method.
- Q7: I would find the method easy to use.
- Q8: Learning to operate the method is easy for me.

Social influence

- Q9: People who influence my behavior think that I should use the method.
- Q10: People who are important to me think that I should use the method.
- Q11: The senior management of this business has been helpful in the use of the method.
- Q12: In general, the organization has supported the use of the, method.

Facilitating conditions

- Q13: I have the resources necessary to use the method.
- Q14: I have the knowledge necessary to use the method.
- Q15: The method is not compatible with other methods I use.
- Q16: A specific person (or group) is available for assistance with method difficulties.