Managing technical debt through software metrics, refactoring and traceability
Charalampidou, Sofia

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2019

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.
1 INTRODUCTION

The concept of Technical Debt refers to the cost introduced during the software engineering process, as a result of compromising internal quality to meet urgent business demands and provide short-term solutions. The concept stems from the financial domain: “Quick and dirty” engineering decisions and choices, which are made during different development phases, incur debt analogous to financial debt that must be paid to ensure long-term sustainability. Similarly to financial debt, Technical Debt incurs interest payments in the form of increased future maintenance costs. The debt can be unintentional or intentional (McConnell et al. 2007). Unintentional debt refers to cases of poor decisions due to lack of knowledge (e.g. “what is layering?”1). Intentional debt can be short or long term: it can be incurred reactively for tactical reasons with identifiable shortcuts or incurred proactively for strategic reasons. Technical Debt Management (TDM) is one of the most popular research areas in software engineering at this moment, and a relatively young area (90% of the research volume is published after 2010 (Ampatzoglou et al. 2018).

One of the most prevalent characteristics of technical debt is its interdisciplinary nature, since it combines elements both from financial and software engineering theory. Below we present some basic definitions from (Ampatzoglou et al. 2015b) on the intersection of the two fields, mapping financial concepts to software engineering terminology.

1 https://www.productplan.com/glossary/technical-debt/
• **Principal:** The effort that is required to address the difference between the current and the optimal level of design-time quality, in an immature software artefact or the complete software system.

• **Interest:** The additional effort needed to maintain the software, because of its decayed design-time quality.

• **Repayment:** The amount of effort spent on improving design-time quality. This effort will decrease the effort needed for future maintenance tasks.

• **Return on Investment:** The ratio of (1) the additional amount of money that is earned by bringing the product earlier into the market or (2) the additional amount of money that has been earned from the company by investing the effort of the principal in an activity different than the improvement of design-time quality, over the principal.

• **Bankruptcy:** A software engineering project could be considered bankrupt, in case that it fails to survive/evolve (either being cancelled or forced to be rewritten from scratch) due to large maintenance costs, caused by the accumulation of technical debt.

Principal can be calculated rather straightforwardly. Interest on the other hand is one of the most complex notions studied in Technical Debt Management (TDM) literature, and therefore it has attracted special attention from researchers (for instance see (Ampatzoglou et al. 2015c, Guo et al. 2011, Seaman et al. 2012).

TD can occur throughout the life cycle of a product, and it can be related to different artifacts. Thus, in the literature various types of TD have been identified, depending on the moment it is incurred or the activities and artifacts it is associated with. For this purpose, Alves et al. (2016) have created a taxonomy of 15 TD types. Among them, the most common ones are: the design debt and the architecture debt, which refer to problems in maintenance and evolution caused by issues either at the detailed-design (e.g., violation of object-orientation principles) or the architectural level (e.g., lack of modularity); the documentation debt, referring to problems occurring due to missing, inadequate or incomplete documentation; the test debt, occurring due to low-quality testing activities or low code coverage; and the code debt, which refers to low source code quality that can affect its maintainability. The same five TD types rank also as the top-5 most frequently studied TD types in another secondary study on technical debt by Li et al. (2015), as well as the most interesting ones for practitioners (Ampatzoglou et al. 2016). Even though the concrete ranking among these five types varies, in most cases the difference is not significant.
There have been numerous activities proposed for technical debt management (TDM). According to Li et al. (2015), efficient TDM requires the execution of up to eight activities:

- **TD representation**, which aims at representing TD in a uniform manner addressing the concerns of particular stakeholders
- **TD communication**, which aims at communicating TD by making it visible to stakeholders so that it can be discussed and further managed
- **TD monitoring**, which deals with observing the evolution of the cost and benefit of unresolved TD over time.
- **TD identification**, which aims at detecting artifacts that suffer from TD caused by intentional or unintentional technical decisions in a software system through specific techniques, such as static code analysis
- **TD measurement**, which aims at quantifying the benefit and cost of known TD in a software system through estimation techniques, or estimating the level of the overall TD in a system.
- **TD prioritization**, which aims at ranking identified TD items, according to certain predefined rules to support deciding which TD items should be repaid first and which TD items can be tolerated until later releases.
- **TD prevention**, which aims at preventing potential TD from being incurred in future developments
- **TD repayment**, which aims at resolving or mitigating TD in a software system by techniques such as reengineering / refactoring.

From the aforementioned activities, in the rest of this chapter we focus on four activities that are relevant for this thesis: TD Identification (see Section 1.1), TD Prioritization and Repayment (see Section 1.2), and TD Prevention (see Section 1.3).

### 1.1 TD Identification

There are many research efforts that focus on proposing mechanisms to identify technical debt. In particular, Alves et al. (2016) have identified a number of TD indicators in the literature (e.g. code smells, documentation issues, pattern grime, etc.) and how they map to specific TD types. TD indicators allow the discovery of TD items when analysing the different artifacts created during the development of software products. The results of Alves et al. show that some TD types, like design TD, have a fair number of indicators. Interestingly, the TD types are not orthog-
nal; for example in some cases there is a thin line between code and design TD: indicators of design debt are extracted from the source code, but some of them cannot be considered indicators of code debt because they reflect the lack of object-oriented design practices (Alves et al. 2016). Among the identified indicators, the most cited and analysed TD indicator is code smells, which can be explained by the fact that some specific code smells, like the God Class, are easy to identify and connect with TD effects.

In the literature there is a variety of tools that is able to identify the existence of code smells (e.g., long methods (Tsantalis and Chatzigeorgiou 2011a), feature envy (Fokaefs et al. 2007), duplicated code (Roy and Cordy 2008), conditional complexity (Tsantalis and Chatzigeorgiou 2010), etc.). However such tools usually focus mostly on one or two code smells. According to two recent secondary studies on TD management by Ampatzoglou et al. (2015) and Li et al. (2015), SonarQube is the most frequently used tool for identifying Technical Debt Items (TDIs) in source code. SonarQube is a more comprehensive solution, in the sense that it identifies TDIs of various types: (a) code smells, (b) vulnerabilities, (c) bugs, (d) duplicated lines density, and (e) low test coverage. Apart from offering an overview on how many TDIs exist in the system, SonarQube provides the opportunity to focus on specific artifacts that suffer from TD, and also provides a listing of the kind of problems that are identified.

1.2 TD Prioritization and Repayment

Having identified TD, the next step is to prioritize among the items that are going to be repaid, and eventually repay those items according to their priority. TD prioritisation can be performed in three ways (Seaman and Guo, 2011): based on the amount of interest, principal or interest probability. We have already discussed the concepts of principal and interest in Section 1.1, so here we examine interest probability, i.e. the probability of an artifact that contains TD to change. Change proneness is the susceptibility of software artifacts to change, without differentiating between types of change (e.g., new requirements, debugging activities, and changes that propagate from changes in other classes) (Jaafar et al. 2014). Therefore, change proneness can be considered as a proxy of interest probability. More specifically, it can be assumed that the more change-prone an artefact is are, the more likely it is to accumulate interest than less change-prone ones; that is because interest manifests only during maintenance activities (Ampatzoglou et al. 2015c). Addi-
tionally, a class that will never change along evolution, regardless of how poorly designed it is (i.e., high principal) will never produce interest if it is not maintained.

As for repayment, the secondary study of Li et al. (2015) defines TD repayment as the resolution or mitigation of TD in a software system and suggests that there is significant amount of literature on TD repayment approaches, compared to other TD activities. These approaches are classified into seven categories, among which we present the three most prominent ones:

- Refactoring: The application of changes to the code, design, or architecture of a software system in order to improve its internal quality, without however altering its external behaviour.
- Rewriting: Writing from scratch the code that contains TD
- Repackaging: The simplification of the source code, by grouping together cohesive modules with manageable dependencies.

Among these categories of TD repayment approaches, refactoring is the most used (Li et al. 2015). This could be related to the fact that the majority of studies found in the literature concern code TD, mainly due to: (a) the availability of tools supporting the identification, measurement, and repayment of this type of TD; (b) the fact that code TD is concrete and easy to understand; and (c) the experience that practitioners have into using code analysis and code refactoring tools in their everyday work (Li et al. 2015). However, other types of software artifacts can also be refactored (i.e. design artifacts like design patterns, object-oriented database schemas, and software architectures; and software requirements) (Mens and Tourwe 2014).

1.3 TD Prevention

TD prevention refers to averting potential TD from being incurred (Li et al. 2015). According to Li et al., such activities are classified in the following categories:

- *Development process improvement*: The improvement of current development processes in order to prevent the occurrence of certain TD types.
- *Architecture decision making support*: The identification of where TD is located and the evaluation of different architecture design alternatives in order to choose the option with less potential TD
- *Lifecycle cost planning*: The development of cost-effective plans that consider the overall minimization of the system TD, i.e. throughout its lifecycle.
• **Human factors analysis**: The creation of a culture that minimizes the unintentional TD caused by human factors, e.g., indifference or ignorance.

There are no supporting tools for TD prevention reported in the literature; this may be due to the fact that TD prevention can be supported mainly by software development process improvement (e.g. the adoption of continuous integration requires a high coverage of automated unit and integration tests which would significantly prevent test TD) (Li et al. 2015).

According to Van Vliet the later a change comes to software, the costlier it becomes (2008). Therefore, it makes sense to apply TD prevention strategies as early as possible in the software development lifecycle.

### 1.4 Research Design

The research project reported in this dissertation aims at investigating and providing potential solutions to the main problem that is formulated in Section 1.4.1. In order to organise the research activities that were required to be conducted for this purpose, we adopted the Design Science framework, as described by Wieringa (2014), which is described in Section 1.4.2. A detailed description of the research design, and a thorough decomposition of the problem statement into specific research questions following the Design Science framework, are provided in Section Error! Reference source not found.. Section 1.4.4 elaborates on the research methods used for answering the research questions. Finally, Section 1.4.5 gives an overview of the dissertation.

#### 1.4.1 Problem Statement

As mentioned in the beginning of this section, the five core types of TD are related to design, architecture, code, test and documentation. To study technical debt management, as a phenomenon in its entirety, one would need to study all of these five types of TD. That would give the opportunity to examine the phenomenon across all development phases. However it is not possible to go into much depth in all five types within the context of one dissertation. Therefore we decided to focus on three of these types (design debt, code debt and documentation debt), and effectively select specific sub-types of TD from each type (e.g., the subtype of code smells for the code TD type). The justification for the three types, as well as the selection of subtypes are explained in the next paragraphs.
Design and architecture debt can easily get classified together, as they both deal with structural characteristics of the software, so there is a thin line that differentiates them. In this thesis, we focus on design debt, as its identification and refactoring can be automated through source code analysis tools without the need of artifacts (UML diagrams or architectural decisions) that are not readily available. Similarly, when it comes to test debt and code debt, we chose to investigate code debt for two reasons: first, Alves et al. (2016) show that code smell artifacts constitute by far the most common existing TD indicator; second, there is higher availability in studying code artifacts rather than test artifacts. Finally, we included documentation debt within the scope of this thesis as it is a broad topic that can be identified throughout the software lifecycle; this mitigates, to some extent, leaving out the types of TD from the architecting and the testing phases.

In terms of code TD we focus on a specific subtype, namely code smells. As already mentioned, code smells are the most frequent indicator of code TD (Alves et al. 2016). Although there is an abundance of literature on the topic, the majority of the existing work is limited in terms of accuracy (i.e. false positive identification of code smells). This lack of accuracy could lead to unnecessary refactoring actions in the sense that some of the identified code chunks do not actually include TD and thus would not require refactoring. On top of that, existing methods and tools that provide recommendations as refactoring opportunities, usually present a long list of code smells as candidate solutions for refactoring. Therefore, by considering the limited time for selecting among these choices and applying the refactoring per se, there is a need of improving the accuracy of these suggestions by providing a shorter list of refactoring opportunities (i.e., TD repayment actions), that is prioritized based on the urgency of the refactoring to be applied.

In terms of design debt, we also focus on a specific subtype, namely TD caused by wrongful application of design patterns. We focus on design patterns, since their use by practitioners is very common: up to two thirds of the classes of an object-oriented system may be participating in design patterns according to (Feitosa et al. 2019). Given their pervasiveness, a potentially bad instantiation of a design pattern can have negative effects on the quality (i.e., increased unnecessary complexity) and create TD, a phenomenon called pattern grime (Izurieta and Bieman 2013). At a system level, this may amount to a high amount of unintentional technical debt and has grave consequences on the maintenance costs. However, there is no guidance on how to identify badly instantiated patterns and even more importantly on how the repayment of these TD items can be performed (i.e., refactoring to a non-
pattern version). In practice, this process depends mostly on gut feeling and the experience and expertise of software developers.

Regarding **documentation TD**, as in the two previous types, we focus on one specific subtype, namely TD in requirements documentation. We selected requirements, since according to the literature the lack of quality in requirements is related to 9 out of 11 most notable reasons for software project failures (Van Vliet 2008). Documentation TD in general is caused by insufficient, incomplete or outdated documentation (Li et al. 2015). For the specific sub-type of requirement documentation TD, insufficient or incomplete requirements refer to pieces of specifications that are developed either at low quality or do not describe the system under development. (Van Vliet 2008). Outdated requirements refer to cases in which specifications have been developed at an appropriate level of quality (in the early releases of the system), but subsequently the specifications are not updated with new requirements, or changes in existing ones (Van Vliet 2008). Insufficient, incomplete or outdated requirements documentation causes major problems, such as building the wrong system, developing features that will not be used, or doing excessive rework to adapt the system functionality. It is thus of paramount importance to proactively *prevent* the occurrence of insufficient, incomplete or outdated requirements, rather than reacting when they already occurred. However, we currently lack tools that prevent this phenomenon.

Summarizing and synthesizing the above individual problems for the three types of TD, this dissertation aims at addressing the following problem:

*Although in the literature there is a variety of approaches for managing code, design and documentation TD, these approaches suffer from various limitations:*

\[ \text{a) In code TD, tools for identifying, prioritizing, and resolving bad smells lack in accuracy,} \]
\[ \text{b) In design TD, systematic support for identifying incorrectly instantiated patterns is lacking, as well as guidance on how to refactor the design,} \]
\[ \text{c) In documentation TD, we lack tools for preventing the occurrence of insufficient, incomplete or outdated requirements documentation.} \]

**1.4.2 Design Science Framework**

The research project, documented in this dissertation, adopts the design science framework as described by Wieringa (2009). Design Science was initially intro-
duced by March and Smith (1995) aiming to assist stakeholders create or improve “things” in their technological solutions, that serve human purposes. According to the design science framework there are two core activities that can contribute to a domain’s knowledge base: build an artifact for a specific human purpose and evaluate how well the artifact suits this purpose. Although the framework was introduced in the domain of Information Systems, it is sufficiently general to be employed in other disciplines. In particular, the framework was later refined by Wieringa (2009) to also suit software engineering research. The core elements of Wieringa’s version of the Design Science framework, as a refinement of Hevner et al.’s framework (Hevner et al. 2004), are depicted in Figure 1.1.

There are two sources of information for the project, namely environment and knowledge base. The environment can be considered as the starting point for the design science process, since it provides access to the problem domain and gives input in terms of the purpose and the constraints for the artifact to be designed. The knowledge base is the set of theories, scientific knowledge, and existing tools that are relevant to the problem under investigation. In short, the environment drives the design activities, whereas the knowledge base steers the way that a specific solution can be performed, in the sense that the existing knowledge shows what has to be synthesised (e.g., questions to be answered and tools to be created). Design science is an iterative process, during which the researcher analyzes a practical problem from the environment, proposes a solution, evaluates the solution, and then starts over again. By the end of the research project, the main outcome, which is a possible solution, is returned to the environment. This artifact is also added in the knowledge base, together with the rest of the knowledge generated during the process (e.g., answers to research questions).

In Design Science there are two main components aiming to define and explore the problem of interest; the practical problems and the knowledge questions. A practi-
cal problem is defined as “a difference between the way the world is experienced by stakeholders and the way they would like it (the world) to be” (Wieringa 2009). The analysis of a practical problem and the evaluation of a proposed solution are both knowledge questions, which usually emerge through repetitive rounds of activities, known as design cycle. A knowledge question is defined as “a difference between current knowledge of stakeholders about the world and what they would like to know” (Wieringa 2009). Knowledge questions are answered by using analytical or empirical research methods. The two are seamlessly nested to develop the artefact that was initially requested.

The workflow addressed in the design science framework is well-suited for describing long-term research such as PhD projects, since it facilitates the evolution of research questions and solutions at the same time. In particular, it suggests the decomposition of an initial problem statement into practical problems and knowledge questions which become more concrete as the researcher gains more domain knowledge, acquires a better understanding of the problem and progresses with the research solutions.

1.4.3 Practical Problems and Knowledge Questions

This section presents an analysis of the research questions (RQ) addressed in this thesis. As overview of the research questions placed within the context of TD types and TD activities is illustrated in Figure 1.2. As indicated in the problem statement, there are three main TD types within scope of this thesis (left area of the Figure): code TD, design TD, and documentation TD. For each TD, we deal with one or more TD activities (middle vertical area): identification, prioritization, repayment, and prevention. TD activities are connected with flow arrows indicating sequence from one activity to the other (e.g. first we identify design TD and then we repay it). For each pair of TD type and TD activity, we have stated one or more research questions (right area), either as knowledge questions (dark grey boxes) or practical problems (white boxes).

The problem statement that was discussed in Section 1.4.1, drives the formulation of the research questions. Specifically the problem statement concerns the limitations of current approaches for managing three different types of technical debt: 1) in code TD, the lack in accuracy of tools that identify, prioritize and resolve bad smells; 2) in design TD, the lack of systematic support for identifying incorrectly instantiated patterns and refactoring the design; 3) in documentation TD, the lack
of tools for preventing the occurrence of insufficient, incomplete or outdated requirements documentation. These three limitations lead to a number of knowledge questions and practical problems as elaborated in the following paragraphs.

Regarding code TD, despite the variety of methods and tools for dealing with code smells, the lack of accuracy while identifying, prioritizing, and refactoring code smells, leads to inefficient code TD management. Since the in-depth exploration of all code smells would be an unrealistic target, we further focus our investigation on long methods, which is one of the most common and persistent code smells (Chatzigeorgiou and Manakos 2004). As a first step, we focused on improving the accuracy of identifying the existence of long method code smells. To do so, we first looked into which structural quality metrics can be employed for identifying long methods (RQ1.a); then we evaluated the accuracy of identifying long method instances through these structural metrics (RQ1.b) in order to determine metrics that yield the highest accuracy. The next step after identification is prioritization of the identified TD occurrences, in order to repay them in the most efficient order. This question is answered by investigating the interest probability of concrete TD items (TDI), i.e., specific code smells like Long Methods, Conditional Complexity and Code Clones (RQ1.c). The interest probability of a TDI is calculated based on the average change frequency of the artifacts that suffer from the smell and the density of the smell in the system. Therefore TDIs with the highest interest probability should be refactored first due to their frequency and urgency to resolve. The final step in this process is to improve the repayment of code TD. Again, the focus of this work lies on the refactoring of the Long Method smell. Thus, RQ1.d aims at proposing an approach that can be used for extracting Long Method opportunities, and RQ1.e aims at validating the value of the proposed approach.

In the case of design TD, the problem statement focuses on the lack of systematic support for identifying and refactoring incorrectly instantiated design patterns. To this end, RQ2.a attempts to support the identification of design TD, focusing specifically on improper instantiations of the Decorator pattern. We selected the Decorator pattern, because: (a) it is a complex pattern in structure (since it offers two levels of inheritance and self-reference) and thus its instantiation is tricky; and (b) it is not studied in related literature (more details in Section 5.2). The idea is that the application of design patterns can have either positive or negative effect on the quality of a design, depending on certain instantiation parameters (e.g., number of classes, methods, etc.). By considering the values of these parameters in a given design pattern instance (i.e., comparing the values to the thresholds obtained by the
Although in the literature there is a variety of approaches for managing code, design and documentation TD, these approaches suffer from various limitations:

a) In code TD, tools for identifying, prioritizing, and resolving bad smells lack in accuracy.
b) In design TD, systematic support for identifying incorrectly instantiated patterns is lacking, as well as guidance on how to refactor the design.
c) In documentation TD, we lack tools for preventing the occurrence of insufficient, incomplete or outdated requirements documentation.

**Figure 1.2: Practical Problems and Knowledge Questions**

proposed method) we can identify if the pattern is properly instantiated; if it is not, then the pattern constitutes a TD item. After identifying such improper instantiations, one can explore alternative designs and investigate if a refactoring (the most common TD repayment activity) to an alternative design solution would be benefi-
cial. RQ2.b thus looks into support for repaying design TD, by investigating the concrete case of the Decorator pattern. The decision to remove or add a Decorator instance (based on its expected effect on software quality characteristics) can lead the repayment process, in the sense that such a refactoring (adding or removing a pattern) would reduce the amount of TD. In particular we investigate the effect of applying the pattern on various quality attributes including extendibility, flexibility and effectiveness (Bansiya and Davies 2002), which are directly related to TD as sub-characteristics of maintainability (Ampatzoglou et al. 2016). The potential improvement of these quality attributes due to the application/removal of the Decorator pattern is then a form of repaying design TD.

Finally, in the case of documentation TD, the problem statement addresses the prevention of insufficient, incomplete, and outdated requirements documentation. A promising approach to address the abovementioned problem is the establishment of requirements-to-code traceability, in the sense that the existence of traces, would allow the consistent update of requirements alongside code changes. More specifically, the main reason that developers tend to update the code without going back to the requirements documentation is that these artifacts are managed in isolation; thus the identification and update of the affected requirement is time-consuming. Therefore, the existence of traces is expected to motivate developers in consistently updating the requirements. As a first step we reviewed the literature for existing traceability techniques (RQ3.a) that may fit the goal of preventing requirements documentation TD. To this end, we decided to conduct a broad-scope review of the literature in the field of traceability among software artifacts. However, no existing traceability approach was found for effectively preventing requirements traceability TD; in particular, current traceability approaches that link requirements to source code, either covered only after-the-fact traceability, or did not provide sufficient validation. Therefore, in this thesis we proposed a requirements-to-code traceability technique that could be smoothly integrated in the daily routine of software engineers (e.g., by integrating it in the programming IDE) to prevent the accumulation of documentation TD (RQ3.b). To validate the usefulness of the proposed technique in RQ3.c we evaluated how the proposed requirements-to-code traceability approach would influence requirements documentation TD.

1.4.4 Research Methods used in the Thesis

After decomposing the problem statement addressed in this dissertation into knowledge questions and practical problems, in this section we will present how
each knowledge question is answered via a study conducted during the PhD. The conducted studies consist both of empirical and analytical studies. The empirical studies were designed based on the practices of evidence-based software engineering (EBSE), a paradigm advocated in the seminal work by Kitchenham et al (2004). The analytical study that was performed is grounded on the mathematical formulation of the problem and subsequently solving the resulting equations. The choice to use an analytical method for answering the corresponding research question, was based on the need for a comprehensive search-space exploration that can systematically support the application of design patterns, according to the values of the explored parameters.

An overview of the research methods that were used for answering each knowledge question is provided in Table 1.1. This is accompanied by a reference to the section of the dissertation where the study design of the respective method is presented. A detailed description of the research methods and the context in which they were applied follows.

Table 1.1: Overview of Research Methodology

<table>
<thead>
<tr>
<th>Code</th>
<th>Research Question</th>
<th>Research Method</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1.b</td>
<td>How accurate is the identification of long methods using these metrics?</td>
<td>Case Study</td>
<td>Section 2.4</td>
</tr>
<tr>
<td>RQ1.c</td>
<td>How to prioritize long methods, conditional complexity and code clones occurrences, in order to repay them in the most efficient order?</td>
<td>Case Study</td>
<td>Section 3.4</td>
</tr>
<tr>
<td>RQ1.e</td>
<td>How beneficial is the repayment of code TD by applying the proposed extract method refactoring approach?</td>
<td>Case Study</td>
<td>Section 4.4.1, Section 4.5.1</td>
</tr>
<tr>
<td>RQ2.b</td>
<td>How can the effect of the Decorator pattern on design-time qualities, inform the repayment of design TD?</td>
<td>Analytical Exploration</td>
<td>Section 5.3</td>
</tr>
<tr>
<td>RQ3.a</td>
<td>Can we use an existing artifact traceability technique to prevent requirements documentation TD?</td>
<td>Systematic Mapping Study</td>
<td>Section 6.2</td>
</tr>
</tbody>
</table>
Each knowledge question is answered via a study conducted during the PhD. The conducted studies consist both of empirical and analytical studies. The empirical studies were designed based on the practices of evidence-based software engineering (EBSE), a paradigm advocated in the seminal work by Kitchenham et al. (2004). The analytical study that was performed is grounded on the mathematical formulation of the problem and subsequently solving the resulting equations. The choice to use an analytical method for answering the corresponding research question was based on the need for a comprehensive search-space exploration that can systematically support the application of design patterns, according to the values of the explored parameters.

An overview of the research methods that were used for answering each knowledge question is provided in Table 1.1. This is accompanied by a reference to the section of the dissertation where the study design of the respective method is presented. A detailed description of the research methods and the context in which they were applied follows.

Table 1.1: Overview of Research Methodology

<table>
<thead>
<tr>
<th>Code</th>
<th>Research Question</th>
<th>Research Method</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ3.c</td>
<td>How does the application of the requirements-to-code traceability approach influence requirements documentation TD?</td>
<td>Case Study</td>
<td>Section 7.4</td>
</tr>
</tbody>
</table>

In this thesis two empirical research methods have been used; the Case Study and the Systematic Mapping Study. Empirical methods are research methods that use experiences and/or observations for retrieving evidence from a real-world context or an artificial setting suitable for investigating a phenomenon of interest (Tichy and Padberg, 2007). In Software Engineering the use of empirical methods is common for evaluating whether there is an observable impact when applying alternative software techniques. (Basili and Selby, 1991).

**Case studies** aim at understanding a particular phenomenon in its environment (Runeson et al., 2012) and are usually used for monitoring real-life activities. The setup of case studies usually suggests the use of different data collection methods, aiming to gather evidence from multiple, complementary data sources (a process also known as triangulation). This type of research is more suitable for examining relationships among different attributes and normally it is not used for establishing causality (Wohlin et al., 2012). This empirical method has been used to a large extent in this PhD project since we preferred to study the investigated phenomena within their real-world environment. In particular we use case studies to: evaluate the accuracy of identifying long method smells using a set of selected metrics; determine the priority of refactoring different types of code smells for repaying code TD; investigate the impact of repaying code TD by applying the extract method refactoring; and study how the application of a newly proposed requirements-to-code traceability approach can influence documentation TD.

**Systematic Mapping Studies** (SMSs) and Systematic Literature Reviews (SLRs) propose the use of the same systematic approach for collecting existing (primary) studies on a topic of interest, and aim at aggregating knowledge found in the collected literature (Kitchenham et al 2004, Petersen et al. 2008). One of the main differences between the two approaches is that SMSs aim to identify and classify all research related to a broad software engineering topic, while SLRs have as scope the in-depth review of the existing literature and thus they are usually used for studying narrower topics, with a manageable amount of primary studies. In this
PhD project we conducted a SMS to investigate the state of the art on the broad topic of software artifacts traceability, classifying: the type of artifacts that are linked through traceability approaches; the benefits of using the proposed traceability approaches; the way that the benefit of these approaches is measured; and the research methods used in the primary studies for investigating the topic of software artefact traceability.

*Analytical studies* follow a generic process combining the power of the scientific method with the use of a formal process to solve any type of problem (Kenett et al. 2018). The typical steps of an analytical study include: (a) identifying a problem, (b) choosing the appropriate formulation for the problem and design solution elements, (c) exploring candidate solutions and either accept, reject, or modify them, (d) repeating step-c until a solution is reached and then implementing that solution. In this PhD project we conducted an analytical study for exploring the cases (i.e., values of selected parameters) in which a design pattern instance can introduce technical debt. Given the values of the selected parameters in the system under study, a software engineer can select either to refactor from a design patterns to a simpler solution, or vice-versa.

### 1.4.5 Overview of the Dissertation

The main body of this dissertation consists of six Chapters. Table 1.2 presents the research questions and the chapters, in which they are addressed.

<table>
<thead>
<tr>
<th>Technical Debt Type</th>
<th>Technical Debt Activity</th>
<th>Research Question</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code TD</td>
<td>Identify</td>
<td>Which metrics can be used for identifying long methods?</td>
<td>Chapter 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>How accurate is the identification of long methods using these metrics?</td>
<td></td>
</tr>
<tr>
<td>Code TD</td>
<td>Prioritize</td>
<td>What is the priority of refactoring long methods, conditional complexity and code clones, for repaying code TD?</td>
<td>Chapter 3</td>
</tr>
</tbody>
</table>
In this PhD project we conducted an analytical study for exploring the cases (i.e., values of selected parameters) in which a design pattern instance can introduce technical debt. Given the values of the selected parameters in the system under study, a software engineer can select either to refactor from a design pattern to a simpler solution, or vice-versa.

<table>
<thead>
<tr>
<th>Technical Debt Type</th>
<th>Technical Debt Activity</th>
<th>Research Question</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code TD</td>
<td>Repay</td>
<td>Can we propose a method for extracting long method opportunities? How beneficial is the repayment of code TD by applying the extract method refactoring?</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>Design TD</td>
<td>Identify</td>
<td>Which parameters can be used for identifying improper Decorator pattern instantiations that can lead to design TD?</td>
<td>Chapter 5</td>
</tr>
<tr>
<td></td>
<td>Repay</td>
<td>What is the effect of applying the Decorator pattern, on design TD?</td>
<td></td>
</tr>
<tr>
<td>Documentation TD</td>
<td>Prevention</td>
<td>What techniques have been proposed in the area of requirements-to-code traceability?</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Documentation TD</td>
<td>Prevention</td>
<td>Can a requirements-to-code traceability technique be used to prevent the accumulation of documentation TD? How does the application of the requirements-to-code traceability approach influence requirements documentation TD?</td>
<td>Chapter 7</td>
</tr>
</tbody>
</table>

The presented research work has been submitted for publication in scientific journals or international conferences. Five of them are published, and one is currently under review. In all these publications, the PhD student was the first author and main contributor; other authors include the 2 supervisors, other academic or industrial collaborators, or BSc and MSc students assisting in the research as part of their curriculum. A brief outline of these chapters follows:

- Chapter 2 is based on a paper published in the proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engi-
neering (PROMISE) (Charalampidou et al., 2015). This study empirically explores the ability of size and cohesion metrics to predict the existence and the refactoring urgency of long method occurrences on Java open-source methods.

- Chapter 3 is based on a paper published in the proceedings of the 9th International Workshop on Managing Technical Debt (MTD) (Charalampidou et al., 2017a). This study explores code smells by assessing the associated interest probability aiming to provide input for prioritization purposes when making decisions on the repayment strategy.

- Chapter 4 is based on a paper published in the IEEE Transactions on Software Engineering (TSE) (Charalampidou et al., 2017c). This paper introduces an approach (accompanied by a tool) that proposes Extract Method opportunities for refactoring purposes, and evaluates the benefit of their extraction as separate methods. The paper was selected to be presented as Journal First in the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE ‘17). Both TSE and ESEC/FSE are among the top venues in the software engineering community.

- Chapter 5 is based on a paper published in the proceedings of the 32nd ACM Symposium on Applied Computing (SAC) (Charalampidou et al., 2017b). This paper proposes a theoretical model for understanding the effect of patterns on software quality, and explores in detail the impact of the evolution of the Decorator pattern.

- Chapter 6, is based on a paper currently submitted to the Software Quality Journal (SQJ) (Charalampidou et al., 2019). This study is a secondary study focused on empirical studies on software artifact traceability, exploring the goals of existing approaches as well as the empirical methods used for their evaluation.

- Chapter 7 is based on a paper published in the proceedings of the 44th Conference on Software Engineering and Advanced Applications (SEAA) (Charalampidou et al., 2018). In this study we propose a tool-based approach for preventing documentation TD during requirements engineering, by integrating requirements specifications into the IDE, and enabling the real-time creation of traces between requirements and code. The study also provides an evaluation on how the application of the proposed approach can influence documentation TD, in an industrial context.