CHAPTER

1 Technical Debt

The first and most fundamental question to ask before commencing on this journey of refactoring for design smells is: What are design smells and why is it important to refactor the design to remove the smells?

Fred Brooks, in his book *The Mythical Man Month*, [6] describes how the inherent properties of software (i.e., complexity, conformity, changeability, and invisibility) make its design an "essential" difficulty. Good design practices are fundamental requisites to address this difficulty. One such practice is that a software designer should be aware of and address *design smells* that can manifest as a result of design decisions. This is the topic we cover in this book.

So, what are design smells?

> *Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality.*

In other words, a design smell indicates a potential problem in the design structure. The medical domain provides a good analogy for our work on smells. The symptoms of a patient can be likened to a "smell," and the underlying disease can be likened to the concrete "design problem."

This analogy can be extended to the process of diagnosis as well. For instance, a physician analyzes the symptoms, determines the disease at the root of the symptoms, and then suggests a treatment. Similarly, a designer has to analyze the smells found in a design, determine the problem(s) underlying the smells, and then identify the required refactoring to address the problem(s).

Having introduced design smells, let us ask why it is important to refactor[1] the design to remove the smells.

The answer to this question lies in *technical debt*—a term that has been receiving considerable attention from the software development community for the past few years. It is important to acquire an overview of technical debt so that software developers can understand the far-reaching implications of the design decisions that they make on a daily basis in their projects. Therefore, we devote the discussion in the rest of this chapter to technical debt.

[1] In this book, we use the term *refactoring* to mean "behavior preserving program transformations" [13].

Refactoring for Software Design Smells. http://dx.doi.org/10.1016/B978-0-12-801397-7.00001-1

1.1 WHAT IS TECHNICAL DEBT?

Technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions.

Technical debt is a metaphor coined by Ward Cunningham in a 1992 report [44]. Technical debt is analogous to financial debt. When a person takes a loan (or uses his credit card), he incurs debt. If he regularly pays the installments (or the credit card bill) then the created debt is repaid and does not create further problems. However, if the person does not pay his installment (or bill), a penalty in the form of interest is applicable and it mounts every time he misses the payment. In case the person is not able to pay the installments (or bill) for a long time, the accrued interest can make the total debt so ominously large that the person may have to declare bankruptcy.

Along the same lines, when a software developer opts for a quick fix rather than a proper well-designed solution, he introduces technical debt. It is okay if the developer pays back the debt on time. However, if the developer chooses not to pay or forgets about the debt created, the accrued interest on the technical debt piles up, just like financial debt, increasing the overall technical debt. The debt keeps increasing over time with each change to the software; thus, the later the developer pays off the debt, the more expensive it is to pay off. If the debt is not paid at all, then eventually the pile-up becomes so huge that it becomes immensely difficult to change the software. In extreme cases, the accumulated technical debt is so huge that it cannot be paid off anymore and the product has to be abandoned. Such a situation is called *technical bankruptcy*.

1.2 WHAT CONSTITUTES TECHNICAL DEBT?

There are multiple sources of technical debt (Figure 1.1). Some of the well-known dimensions of technical debt include (with examples):

- **Code debt:** Static analysis tool violations and inconsistent coding style.
- **Design debt:** Design smells and violations of design rules.
- **Test debt:** Lack of tests, inadequate test coverage, and improper test design.
- **Documentation debt:** No documentation for important concerns, poor documentation, and outdated documentation.

In this book, we are primarily concerned with the design aspects of technical debt, i.e., design debt. In other words, when we refer to technical debt in this book, we imply design debt.

To better understand design debt, let us take the case of a medium-sized organization that develops software products. To be able to compete with other organizations

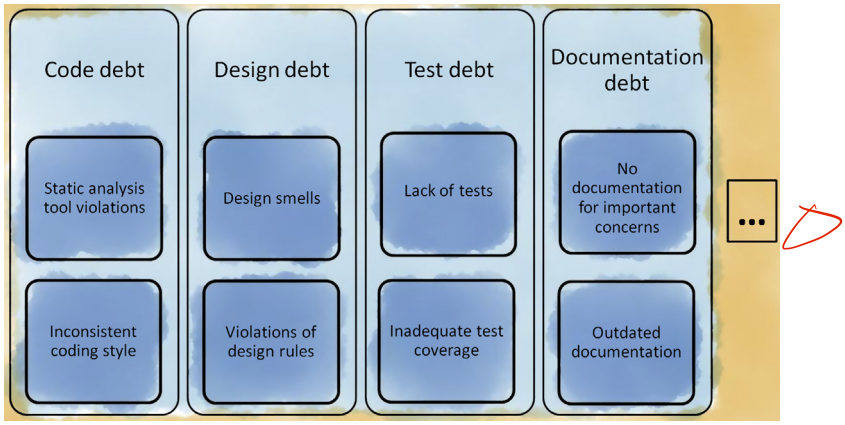

FIGURE 1.1

Dimensions of technical debt.

in the market, this organization obviously wants to get newer products on the market faster and at reduced costs. But how does this impact its software development process? As one can imagine, its software developers are expected to implement features faster. In such a case, the developers may not have the opportunity or time to properly assess the impact of their design decisions. As a result, over time, such a collection of individual localized design decisions starts to degrade the structural quality of the software products, thereby contributing to the accumulation of design debt.

If such a product were to be developed just once and then no longer maintained, the structural quality would not matter. However, most products are in the market for a long time period and therefore have an extended development and maintenance life cycle. In such a case, the poor structural quality of the software will significantly increase the effort and time required to understand and maintain the software. This will eventually hurt the organization's interests. Thus, it is extremely important for organizations to monitor and address the structural quality of the software. The work that needs to be invested in the future to address the current structural quality issues in the software is design debt.

An interesting question in the context of what constitutes technical debt is whether defects/bugs are a part of this debt. Some argue that defects (at least some of them) originate due to technical debt, thus are part of technical debt. There are others who support this viewpoint and argue that if managers decide to release a software version despite it having many known yet-to-be-fixed defects, these defects are a part of technical debt that has been incurred.

However, there are others in the community who argue that defects do not constitute technical debt. They argue that the main difference between defects and technical debt is that defects are visible to the users while technical debt is largely invisible. We support this stance. In our experience, defects are rarely ignored by the organization and receive much attention from the development

teams. On the other hand, issues leading to technical debt are mostly invisible and tend to receive little or no attention from the development teams. Why does this happen?

This happens because defects directly impact external quality attributes of the software that are directly visible to the end users. Technical debt, on the other hand, impacts internal quality of the software system, and is not directly perceivable by the end users of the software. Organizations value their end users and cannot afford to lose them; thus, defects get the utmost attention while issues related to "invisible" technical debt are usually deferred or ignored. Thus, from a practical viewpoint, it is better to leave defects out of the umbrella of technical debt, so that they can be dealt with separately; otherwise, one would fix defects and mistakenly think that technical debt has been addressed.

1.3 WHAT IS THE IMPACT OF TECHNICAL DEBT?

Why is it important for a software practitioner to be aware of technical debt and keep it under control? To understand this, let us first understand the components of technical debt. Technical debt is a result of the principal (the original hack or shortcut), and the accumulated interest incurred when the principal is not fixed. The interest component is compounding in nature; the more you ignore it or postpone it, the bigger the debt becomes over time. Thus, it is the interest component that makes technical debt a significant problem.

Why is the interest compounding in nature for technical debt? One major reason is that often new changes introduced in the software become interwoven with the debt-ridden design structure, further increasing the debt. Further, when the original debt remains unpaid, it encourages or even forces developers to use "hacks" while making changes, which further compounds the debt.

Jim Highsmith [45] describes how Cost of Change (CoC) varies with technical debt. A well-maintained software system's actual CoC is near to the optimal CoC; however, with the increase in technical debt, the actual CoC also increases. As previously mentioned, in extreme cases, the CoC can become prohibitively high leading to "technical bankruptcy."

Apart from technical challenges, technical debt also impacts the morale and motivation of the development team. As technical debt mounts, it becomes difficult to introduce changes and the team involved with development starts to feel frustrated and annoyed. Their frustration is further compounded because the alternative—i.e., repaying the whole technical debt—is not a trivial task that can be accomplished overnight.

It is purported that technical debt is the reason behind software faults in a number of applications across domains, including financing. In fact, a BBC report clearly mentions technical debt as the main reason behind the computer-controlled trading error at U.S. market-maker Knight Capital that decimated its balance sheet [46].

CASE STUDY

To understand the impact that technical debt has on an organization, we present the case of a medium-sized organization and its flagship product. This product has been on the market for about 12 years and has a niche set of loyal customers who have been using the same product for a number of years.

Due to market pressures, the organization that owns the product decides to develop an upgraded version of the product. To be on par with its competitors, the organization wants to launch this product at the earliest. The organization marks this project as extremely important to the organization's growth, and a team of experienced software architects from outside the organization are called in to help in the design of the upgraded software.

As the team of architects starts to study the existing architecture and design to understand the product, they realize pretty soon that there are major problems plaguing the software. First and foremost, there is a huge technical debt that the software has incurred during its long maintenance phase. Specifically, the software has a monolithic design. Although the software consists of two logical components—client and server—there is just a single code base that is being modified (using appropriate parameters) to either work as a client or as a server. This lack of separation of client and server concerns makes it extremely difficult for the architects to understand the working of the software. When understanding is so difficult, it is not hard to imagine the uncertainty and risk involved in trying to change this software.

The architects realize that the technical debt needs to be repaid before extending the software to support new features. So they execute a number of code analyzers, generate relevant metrics, formulate a plan to refactor the existing software based on those metrics, and present this to the management. However, there is resistance against the idea of refactoring. The managers are very concerned about the impact of change. They are worried that the refactoring will not only break the existing code but also introduce delays in the eventual release of the software. They refer the matter to the development team. The development team seems to be aware of the difficulty in extending the existing code base. However, surprisingly, they seem to accept the quality problems as something natural to a long-lived project. When the architects probe this issue further, they realize that the development team is in fact unaware of the concept of technical debt and the impact that it can have on the software. In fact, some developers are even questioning what refactoring will bring to the project. If the developers had been aware of technical debt and its impact, they could have taken measures to monitor and address the technical debt at regular intervals.

The managers have another concern with the suggestion for refactoring. It turns out that more than 60% of the original developers have left the project, and new people are being hired to replace them. Hence, the managers are understandably extremely reluctant to let the new hires touch the 12-year old code base. Since the existing code base has been successfully running for the last decade, the managers are fearful of allowing the existing code to be restructured.

So, the architects begin to communicate to the development team the adverse impacts of technical debt. Soon, many team members become aware of the cause behind the problems plaguing the software and become convinced that there is a vital need for refactoring before the software can be extended. Slowly, the team starts dividing into pro-refactoring and anti-refactoring groups. The anti-refactoring group is not against refactoring per se, but does not want the focus of the current release to be on refactoring. The pro-refactoring group argues that further development would be difficult and error-prone unless some amount of refactoring and restructuring is first carried out.

Eventually, it is decided to stagger the refactoring effort across releases. So, for the current release, it is decided to refactor only one critical portion of the system. Developers are also encouraged to restructure bits of code that they touch during new feature development. On paper, it seems like a good strategy and appears likely to succeed.

However, the extent of the incurred technical debt has been highly underestimated. The design is very tightly coupled. Interfaces for components have not been defined. Multiple responsibilities

Continued

CASE STUDY—cont'd

have been assigned to components and concerns have not been separated out. At many places, the code lacks encapsulation. As one can easily imagine, each and every refactoring is difficult, error-prone, and frustrating. In short, it seems like a nightmare. The team starts to feel that it would be better to rewrite the entire software from scratch.

In the end, in spite of the well laid-out strategy, there is considerable delay in the release of the product. In fact, to reduce further delays in the release, the number of new features in the release is significantly reduced. So, when the product is eventually released in the market, it is 6 months later than originally planned and with a very small set of new features! Although the product customers are not happy, they are promised a newer version with an extended set of features in a few months' time. This is possible because the refactoring performed in the current release positions the design for easier extension in the future. In other words, since part of the debt has been paid (which otherwise could have led the project into technical bankruptcy), it has paved the way for further extension of the product!

1.4 WHAT CAUSES TECHNICAL DEBT?

The previous section discussed the impact of technical debt in a software system. To pay off the technical debt or to prevent a software system from accruing technical debt, it is important to first think about why technical debt happens in the first place.

Ultimately, the decisions made by a manager, architect, or developer introduce technical debt in a software system. For instance, when a manager creates or modifies a project plan, he can decide whether to squeeze in more features in a given time span or to allocate time for tasks such as design reviews and refactoring that can ensure high design quality. Similarly, an architect and a developer have to make numerous technical decisions when designing or implementing the system. These design or code-level decisions may introduce technical debt.

Now, the question is: Why do managers or architects or developers make such decisions that introduce technical debt in the software system? In addition to lack of awareness of technical debt, the software engineering community has identified several common causes that lead to technical debt, such as:

- **Schedule pressure**: Often, while working under deadline pressures to get-the-work-done as soon as possible, programmers resort to hasty changes. For example, they embrace "copy-paste programming" which helps get the work done. They think that as long as there is nothing wrong syntactically and the solution implements the desired functionality, it is an acceptable approach. However, when such code duplication accumulates, the design becomes incomprehensible and brittle. Thus, a tight schedule for release of a product with new features can result in a product that has all the desired features but has incurred huge technical debt.
- **Lack of good/skilled designers**: Fred Brooks, in his classic book *The Mythical Man Month* [6], stressed the importance of good designers for a successful project. If designers lack understanding of the fundamentals of software design

and principles, their designs will lack quality. They will also do a poor job while reviewing their team's designs and end up mentoring their teams into following the wrong practices.

- **Lack of application of design principles**: Developers without the awareness or experience of actually applying sound design principles often end up writing code that is difficult to extend or modify.
- **Lack of awareness of design smells and refactoring**: Many developers are unaware of design smells that may creep into the design over time. These design smells are indicative of poor structural quality and contribute to technical debt. Design smells can be addressed by timely refactoring. However, when developers lack awareness of refactoring and do not perform refactoring, the technical debt accumulates over time.

Often, given the different cost and schedule constraints of a project, it may be acceptable to temporarily incur some technical debt. However, it is critical to pay off the debt as early as possible.

1.5 HOW TO MANAGE TECHNICAL DEBT?

It is impossible to avoid technical debt in a software system; however, it is possible to manage it. This section provides a brief overview of high-level steps required to manage technical debt.

Increasing awareness of technical debt: Awareness is the first step toward managing technical debt. This includes awareness of the concept of technical debt, its different forms, the impact of technical debt, and the factors that contribute to technical debt. Awareness of these concepts will help your organization take well-informed decisions to achieve both project goals and quality goals.

Detecting and repaying technical debt: The next step is to determine the extent of technical debt in the software product. Identifying specific instances of debt and their impact helps prepare a systematic plan to recover from the debt. These two practical aspects of managing technical debt are addressed in detail in Chapter 8.

Prevent accumulation of technical debt: Once technical debt is under control, all concerned stakeholders must take steps to ensure that the technical debt does not increase and remains manageable in the future. To achieve this, the stakeholders must collectively track and monitor the debt and periodically repay it to keep it under control.