CHAPTER

Repaying Technical Debt in Practice

8

In the earlier chapters, we discussed design smells and explained how they contribute to technical debt by impacting the design quality. We also looked at a number of refactoring techniques that help repay the accumulated technical debt. However, in practice, refactoring is not an easy or simple task. There are a number of challenges that have to be overcome while refactoring in a real-world context. In this chapter, we leverage our experience to provide practical guidelines to address these challenges.

Refactoring is a vast area and includes a wide array of topics. Bearing in mind that this book is about design smells, in this chapter, we focus specifically on refactoring for design smells as a way to repay technical debt. We center our discussion along three dimensions: tools, processes, and people (Figure 8.1). We begin with an overview of the different kinds of tools that help detect, analyze, and address smells. Since an important consideration in real-world projects is getting buy-in from the management for refactoring, we next outline a few practices that can be adopted in a real-world setting to ensure backing from the management. We also present a structured process to guide effective refactoring in practice. Ultimately, the key to developing high-quality software is to have a team of competent and committed software engineers. Hence, we conclude this chapter with a brief discussion on some effective ways of building such a team.

8.1 THE TOOLS

There are various kinds of tools available to help you identify design smells and candidates for refactoring, quantify their significance, visualize the accumulated technical debt, and perform refactoring to repay debt. Specifically, we will discuss the following kinds of tools in this section:

- Comprehension tools
- Critique tools, code clone detectors, and metric tools
- Technical debt quantification and visualization tools
- Refactoring tools

Refactoring for Software Design Smells. http://dx.doi.org/10.1016/B978-0-12-801397-7.00008-4

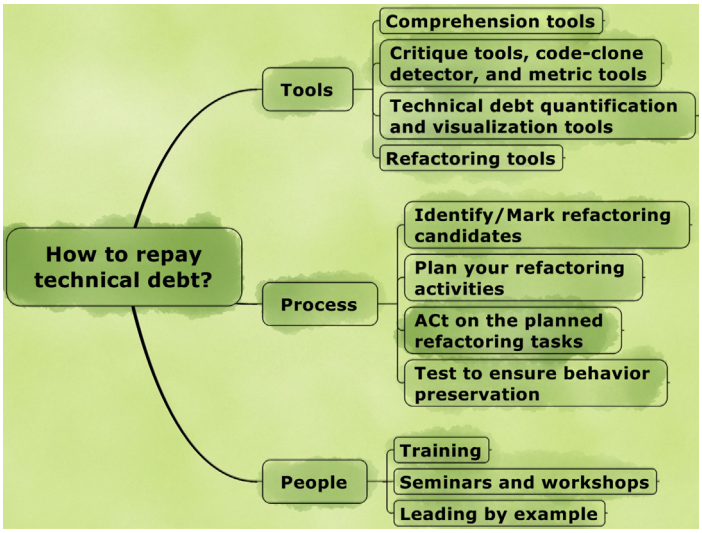

FIGURE 8.1

Key aspects for repaying technical debt.

In this section, we provide only a general description of these tools. Please check Appendix B for a detailed listing of commercial and open source tools.

8.1.1 COMPREHENSION TOOLS

An important prerequisite for proposing and making proper changes in the code base of any real-world software system is to understand the internal structure of the code. Large software systems (the ones that span millions of lines of code) are difficult to comprehend due to their size and complexity. To address this, you can employ comprehension tools to understand the internal structure of the code (through visual aids such as control structure visualization, data flow analysis, call-sequence analysis, and inheritance graph).

8.1.2 CRITIQUE TOOLS, CODE CLONE DETECTORS, AND METRIC TOOLS

Critique tools analyze the software and report problems in the form of violations of predefined rules. Depending on the granularity of the smells they report, they can be classified as:

- **Architectural critique tools** that report architectural violations
- **Design critique tools** that report design smells
- **Code analysis tools** that identify potential bugs (aka "bug patterns") at source code level

The presence of code clones indicate the possibility of smells such as Duplicate Abstraction and Unfactored Hierarchy. It is, therefore, important to detect code clones so that they can be addressed. In this context, code clone detectors can be used to analyze source code and identify code clones.

Metric tools analyze the code or design and report the results in the form of numeric values. These results then have to be analyzed to garner insights on design quality. Let us consider the example of Weighted Methods per Class (WMC), an object-oriented metric, which is a summation of Cyclomatic complexity of the methods in that class. By looking at the average values for WMC, a designer can get an idea of the level of complexity of the classes in his design. Further, by checking the reported WMC value against a fixed threshold value (say 50), he can detect instances of Insufficient Modularization smell (Section 5.2).

8.1.3 TECHNICAL DEBT QUANTIFICATION AND VISUALIZATION TOOLS

Given the significance of technical debt, it is important to be able to measure it. A prerequisite for measuring something is to be able to quantify it. In this context, technical debt is very difficult to quantify accurately. There are two main reasons for this. First, there is no clear consensus on the factors that contribute to technical debt. Second, there is no standard way to measure these factors. It is pertinent to note that there are some recent attempts that try to quantify technical debt; however, these approaches consider only a few factors which limits the usefulness of these approaches.

8.1.4 REFACTORING TOOLS

Broadly, refactoring tools can be classified into the following two categories:

- **Tools that detect refactoring candidates**—These tools analyze the source code and identify not only the entity or entities that need to be refactored but also the potential refactoring steps that can be applied. However, note that these tools are still in a nascent phase of development.
- **Tools that perform a refactoring**—these tools can be employed to perform the required refactoring in the source code once the refactoring candidates have been identified. There are a few popular IDEs (Integrated Development Environments) that provide some automated support for refactoring; however, their support is limited to executing simple refactoring steps.

8.1.5 APPLYING TOOLS IN PRACTICE

It is important to consider the following practical aspects while using these tools in real-world projects.

- Violations or suggestions generated by critique and refactoring identification tools should be treated only as indications. These must be carefully analyzed

manually by looking at the context (recall the importance of context that we discussed in the previous chapter) to eliminate false positives.

- As of this writing, refactoring tools are still maturing. For instance, there are IDEs that support carrying out refactoring tasks such as "extract class" refactoring, but they do not execute the task flawlessly i.e., they require moving of methods manually and leave broken code throughout the code base. Further, existing refactoring tools do not support refactoring complex smells such as Hub-like Modularization (Section 5.4) automatically. Hence, in practice, you may need to carry out refactoring tasks manually without any tool support.
- Selection of tools should be carefully performed depending on the project needs and organizational context. For instance, one of the authors of this book purchased a UML analysis tool to detect smells. However, this tool did not work with the UML diagrams that were created using Rational Rose and Enterprise Architect tools due to XMI (XML Metadata Interchange) compatibility issues. Hence, it is important to evaluate a commercial tool before purchasing it.

8.2 THE PROCESS

In this section, we discuss how refactoring should be systematically approached in a real-world setting. There are many challenges in taking up refactoring to repay technical debt in large-scale industrial software projects (especially the ones in maintenance phase with the team distributed globally). We first outline a few of these challenges and list a few best practices that could be useful to get buy-in for refactoring from concerned stakeholders. Next, we present a process model for systematic refactoring in a real-world context. We conclude with a few practical suggestions for effective refactoring.

8.2.1 CHALLENGES IN REFACTORING

There are many challenges you may come across when you plan to take up refactoring tasks in real-world projects. Some important ones are:

- **Lack of infrastructure and tools**: It is hard to ensure that the behavior of the software is unchanged post-refactoring when unit tests are missing. However, unit tests cannot be written before making the code/design testable. This leads to a "chicken-and-egg" problem for taking up refactoring tasks (especially in the context of legacy projects, which usually lack unit tests and require refactoring as well). In addition, many projects do not have access to automated analysis tools that identify smells or quantify technical debt. Lack of sufficient infrastructure and tool support makes project teams hesitant to take up refactoring tasks.
- **Fear of breaking working code**: The fear of breaking working code is the major obstacle in taking up refactoring tasks. Many developers and managers

believe in "if it ain't broke, don't fix it" (and by "broke" they mean defects that can be found in the software using testing or reported by customers). Some team members even fuel fears in other team members by arguing that refactoring can break the working software and make it even worse. In cases where it is hard to comprehend the software code base (for instance, in the case of a complex legacy software where the team may have no access to the original developers of the software), the fear is even more pronounced. This fear of breaking code is a legitimate one because even small changes can often break working software thus impacting customers who are using the software. Further, it is often hard to find the root cause of regressions in large code bases.

- **Focus on feature-completion**: Many project managers are often focused on feature-completion and neglect design quality. They don't appreciate the role of refactoring in creating high-quality designs, and consider refactoring as "needless rework." In fact, many managers have an unrealistic expectation that designers should "get the design right the first time" irrespective of project considerations like changing requirements.
- **Mental blocks**: When a team inherits a legacy code base with massive technical debt, some team members (including the manager) may not want to notice or acknowledge the "elephant in the room" (which in this case is the massive technical debt). Or, some team members have a "get-the-work-done-no-matter-what" mindset and lack quality-orientation. Such team members tend to discourage any attempts at refactoring. There are many such mental blocks one needs to be aware of when proposing refactoring tasks in a project.

8.2.2 GETTING BUY-IN FOR REFACTORING

Getting buy-in from project stakeholders is important for repaying technical debt. In fact, the word "technical debt" was coined by Ward Cunningham to convey the extent of the deterioration of the software structure in the form of a metaphor so that managers from non-technical backgrounds can easily relate to the problem.

Below, we outline some approaches that we have found to be effective in getting a buy-in from stakeholders for repaying technical debt:

- **Quantify and visualize debt**: The first step toward getting a buy-in is to show the current state of quality to the stakeholders. This can be achieved in many ways:
  - Quantifying technical debt and putting a monetary value to the debt (though the current tool support is limited) is one of the important ways to show the extent of debt to the stakeholders (see Section 8.1.3).
  - Using various metrics is an alternative way to quantify the quality of a software system. Showing key metric values can attract focus on aspects that need immediate attention. For instance, while consulting in a project, one of the authors showed that 40% of the code was duplicated in the

analyzed code base (against 3% of duplicate code threshold prescribed in their organization's process and quality guidelines). The project management could not ignore these numbers and had to urgently formulate an action plan to address it.

- Presenting numerical values (such as metrics) in a graphical form is an effective approach to visually demonstrate the debt to stakeholders, especially those who do not have a technical background in software development.
- Demonstrating quality-violation trend, i.e., how quality-violations have changed over time for the software could prove an eye-opener for the stakeholders. In one of the projects, an architect known to the authors used a bar chart to plot the measure of total Cyclomatic complexity/LOC (Lines of Code) of the software across its releases and demonstrated deterioration of the quality by showing how the bars grew bigger over time. The graphs drew the stakeholders' attention and subsequently mitigation steps were taken up to address the problem of growing complexity in the code base.

- **Demonstrate the viability of refactoring**: Once stakeholders understand the need for refactoring, the next task is to show them that refactoring is feasible despite the existing challenges that confront a development team. Here is a list of practical guidelines that could help in this context:
  - Plan a refactoring activity on a smaller scale. Take baby steps: for example, only refactor one smell at a time.
  - Quantify design quality before and after the refactoring so that a subsequent comparison of design quality highlights the positive impact of refactoring. For instance, select a set of metrics that you measure periodically and create a visualization showing the trend of quality violation (or selected metrics) over a few builds. If you can showcase successful refactoring at a smaller scale, you can gain the trust of the stakeholders making it easier to get their buy-in for refactoring at a larger scale.

8.2.3 "IMPACT" - A REFACTORING PROCESS MODEL

We have come across a number of cases in industrial projects where refactoring is performed in an ad hoc fashion, resulting in numerous problems. It is, therefore, important to follow a structured approach while refactoring. In this section, we describe a process model called "IMPACT" that provides guidance for systematic refactoring in practice. IMPACT is comprised of the following four fundamental steps that are executed in order:

1. Identify/Mark refactoring candidates
2. Plan your refactoring activities
3. ACt on the planned refactoring tasks
4. Test to ensure behavior preservation

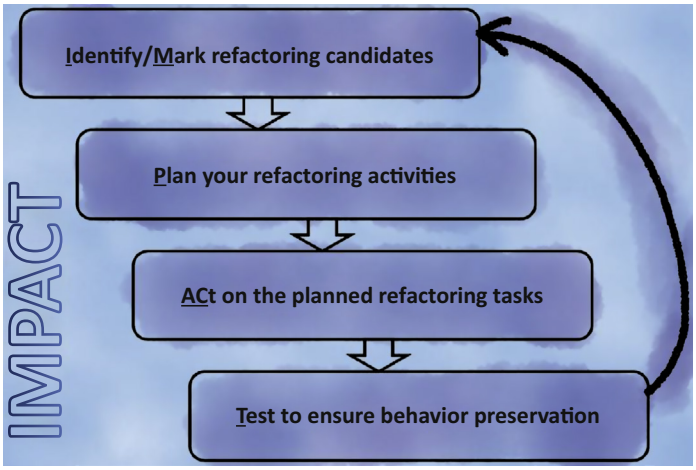

FIGURE 8.2

IMPACT refactoring process model.

These steps are described in the following sub-sections and illustrated in Figure 8.2. Note that these steps are performed in an iterative loop for the following reasons:

- While executing Steps 3 and 4, often previously-hidden refactoring candidates may become visible. This will require revisiting Step 1 to update and possibly re-prioritize the list of refactoring candidates.
- In a real-world context, often project conditions are in a state of flux, i.e., requirements may be added, changed, or even removed, or priorities in the project may change. Thus, Step 2 may need to be revisited to update the refactoring plan (during the refactoring process) to reflect changing project considerations.

8.2.3.1 Identify/mark refactoring candidates

The first step in the refactoring process is to analyze the code base and identify refactoring candidates. Projects can carry out manual code/design reviews and employ automated code/design analysis tools to find smells and determine candidates for refactoring (See Section 8.1 on refactoring tools). In large code bases, executing analysis tools is a faster and less effort-intensive way to find refactoring candidates as compared to performing a manual review. However, manual reviews are more effective and less error-prone since they can consider and exploit domain knowledge, the context of the design, and design expertise more effectively. In our experience, combining manual analysis with tool analysis is a practical and effective way to identify refactoring candidates in industrial projects [4].

8.2.3.2 Plan your refactoring activities

Once you identify smells using design analyzers, clone analyzers, metric threshold violations, or manual reviews, it is important to analyze their impact, prioritize them,

and prepare a plan to address them. To analyze the impact of a smell, consider factors such as severity, scope, and interdependence. For instance, consider the example of Cyclically-dependent Modularization (Section 5.3) smell. If there are multiple cycles among a set of abstractions, then the severity of the smell will be higher than that of a unit cycle (i.e., a cycle of length 1). Similarly, if the participating abstractions in a cycle belong to different packages/namespaces, then the smell would impact a wider scope i.e., architecture of the system.

After analyzing the impact of the identified smells, prioritize them based on the following additional factors:

- The potential gain after removing the smell
- Available time
- Availability of tests for the target module(s).

Based on the prioritized list of identified smells and their refactoring, an execution plan for the refactoring can be appropriately formulated.

8.2.3.3 Act on the planned refactoring tasks

Team members can take up planned refactoring tasks and execute them by carrying out the refactoring in the code. In this process, they can also use any automated refactoring support provided by IDEs to carry out the refactoring tasks.

8.2.3.4 Test to ensure behavior preservation

This is an important step in the refactoring process. Each refactoring activity should be followed by automated regression tests to ensure that the behavior post-refactoring is unchanged.

Often, automated tests may not be available for an entity that needs to be refactored. In such scenarios, refactoring presents an opportunity to create automated tests. A recommended practice for such cases is to first write tests for the entity that needs to be refactored, then refactor the entity, and finally test it to verify the behavior.

Our experience shows that in addition to executing various tests, manual code and design reviews are necessary to ensure that refactoring has been correctly and completely performed (see anecdote below). Hence, we recommend complementing regression tests with manual reviews after refactoring.

ANECDOTE

One of the participants in the Smells Forum shared this experience. His team was working on a large multithreaded application. The architect of the team identified a method in a critical component of that application that needed to be refactored. One of the developers took up that task and extracted methods from that large and complex method to simplify it. All the tests passed for this refactoring, and these changes became a part of the next release.

A few months later, a high-priority defect was received from a customer who was using that application. The reported problem was that the application crashed unexpectedly when run continuously for a few days. When the team analyzed the problem, they found that the application

ANECDOTE—cont'd

kept creating threads without terminating some of them; eventually, the application crashed because new threads could not be spawned.

A root cause analysis of this problem pointed to the refactoring that the developer had performed. While refactoring the large and complex method, he had accidentally broken the logic that involved thread termination. Since multithreading is non-deterministic, this problem was not discovered while testing. Eventually, the thread termination problem was fixed and a patch was delivered to the customer.

The key take-away from this experience is that post-refactoring, testing alone is not sufficient to ensure the correctness of the behavior. Manual code and design reviews are needed to ensure that the behavior is preserved, especially when refactoring critical or multithreaded components.

8.2.4 BEST PRACTICES FOR REFACTORING TO REPAY TECHNICAL DEBT

In the real world, the business objectives of a project are typically given a higher priority over its quality goals. As a designer or architect, it is important to ensure you are meeting both objectives. We outline a few best practices that we have found useful to help you take care of the quality goals of your project while honoring its business goals:

- Often, you may realize that a part of the design needs to be refactored; however, there are more urgent tasks that need your attention. In such a case, make a note of the pending refactoring task in a TO-Refactor list (similar to a TO-DO list that people maintain for their day-to-day tasks) so that you do not lose sight of the refactoring that needs to be taken up (or assigned) in the future.
- A "Big-bang" refactoring approach does not work well in practice due to various reasons (e.g., tight schedule and continuous integration and release). Therefore, in real world projects, it is wise to refactor small portions of the code base at a time. Additionally, automate tests and run them after every refactoring to ensure that everything still works as intended.
- Since tests are essential to ensure that refactoring does not change the intended behavior of the system, it is important to ensure that unit tests have been created and are maintained. The absence of unit tests is a powerful deterrent for refactoring. Therefore, if unit tests are missing, break the chicken-and-egg problem (discussed earlier in Section 8.2.1) by iterating between creating unit-tests and performing relevant refactoring.
- Allocate sufficient buffer time to address refactoring tasks in your TO-Refactor list (along with corresponding tasks such as updating design documents.)
- In a real-world project with a short time to market, it is not always feasible to allocate a dedicated time for refactoring. This requires a smart approach to refactoring, and you should try and club refactoring with bug-fixes. In

particular, when a bug is assigned to you, first fix the bug and then use the opportunity to refactor relevant parts within the code that was touched during the bug fix. This is in line with the philosophy "check-in code better than what you checked-out."

- Allocate dedicated time after each major release for refactoring tasks to keep technical debt under control before commencing development on the next major release.

8.3 THE PEOPLE

People are the real heart and soul of a software project. It is not surprising to note that a software reflects the quality and commitment of the software development team. It is, therefore, important to educate the team about why refactoring is important and raise awareness about repaying technical debt. In this context, we have found the following approaches to be most effective:

8.3.1 TRAINING

Fred Brooks, in his book "Mythical Man Month," suggests the use of short courses to help train potential practitioners for future assignments [6]. Our experience supports this—in fact, the feedback for employees who have undergone software design training sessions conducted by us suggests that focused workshops can significantly raise their awareness of good design practices. We therefore believe that short and focused (preferably hands-on) training sessions on design principles, design smells, and refactoring can help sensitize developers to good design practices.

8.3.2 SEMINARS AND WORKSHOPS

In our experience, seminars by eminent thought leaders on repaying technical debt and refactoring tend to generate interest and excitement in development teams. Further, internal refactoring workshops provide a platform for project teams to share their real-world experiences and lessons learned.

8.3.3 LEADING BY EXAMPLE

While training sessions and seminars can certainly help increase awareness, what we have found to be most effective in fostering a culture of refactoring is when a role-model, mentor, or team-lead sets an example for the rest of the team by adopting these practices himself. Very often, we have come across instances where a team-lead or architect "preaches" about the importance of following good design practices but fails to demonstrate it in action in his own work. This ends-up demoralizing the team. In conclusion, leading by example is the best way to foster excellence and create a culture of quality in the organization.