Technical Debt: A Clean Architecture Implementation

Alessandro C. Beltrão[1], Fábio de Almeida Farzat[2], Guilherme Horta Travassos[1]

[1]PESC – COPPE – Universidade Federal do Rio de Janeiro (UFRJ)
Caixa Postal 68.501 – 21.941-972 – Rio de Janeiro – RJ – Brazil

[2]ED Company
Rio de Janeiro – RJ - Brazil

{alessandrocb,ght}@cos.ufrj.br, fabio@ed.company

Abstract. Technical Debt (TD) and Technical Debt Management (TDM) are terms that are receiving increasing attention from practitioners and researchers. They reflect a concern on how shortcuts taken during the software development process can incur negative impacts on software maintainability and how practitioners may use tools and techniques to mitigate the effects of the debt over time. A widely used tool to manage TD on an implementation level is SonarQube with the SQALE method, as it allows developers and managers to track debt over time. However, even SonarQube has its weaknesses since it only provides a set of architecture agnostic rules for TD, and the implementation of new rules can prove to be a challenging job. In this paper, we discuss how, during a real industrial project on a Brazilian software house, we developed a set of rules based on the Clean Architecture model, created a plug-in for SonarQube, and integrated it into our development cycle. At last, the preliminary results show that using a rigorous set of rules allows keeping track of TD on an implementation level.

1. Introduction

The term technical debt (TD) is a metaphor on how compromises that yield short term benefits may have negative impacts on the future of software maintainability. Ward Cunningham created the concept of TD around 27 years ago, and by that time, the idea was used only to describe shortcuts on the implementation level of software [Cunningham 1992]. Today, the concept of TD may even be used in the context of requirements and process engineering, architecture definition, and testing [Brown et al. 2010].

Even though TD can be detrimental for a project, if it is kept visible and under control, development and management teams can benefit from intentional TD to speed up development, release new features, and in some cases, help the company to get ahead of the competition. The problem is that not all TD is intentional. While intentional TD can help teams to deliver faster, unintentional TD can accumulate incrementally and harm the maintainability and evolution of software [Li, Avgeriou, and Liang 2015].

One of the most used tools in the context of continuous code quality and to track TD [Li, Avgeriou, and Liang 2015] is *SonarQube,* developed by *SonarSource*. Currently, *SonarQube* supports twenty-six programming languages, has four versions,

and many available plug-ins. It implements the SQALE method, and currently, on the community software version, three indexes are available for the end-user: Maintainability, Security, and Reliability. To *SonarQube* generate the *SQALE* indexes, users must define a set of rules and aggregate them on quality profiles, which should be applied to a piece of code. It will allow that any broken rule generates a TD *issue*. Based on the number of *issues* and their severity, *SonarQube* will create the *SQALE* indexes to inform the user about the source code quality.

Additionally to the standard SonarQube rules, the languages C# and VB.Net adopt alternative processes to building code rules, because Microsoft provides a powerful tool to allow developers to create syntax analyzers using the .Net Compiler Platform (Roslyn). Roslyn provides an interface in such a way developers can write code analyzers for C# and VB.Net by querying the Semantic Model and Syntax Tree of a piece of code (or a Visual Studio Solution) to ask specific questions regarding the source code. When Roslyn identifies a non-compliant code, it can report a diagnostic, on the location of the issue, with an informative message. The *SonarScanner* can automatically import these reports and generate generic issues, or we can package it and create a *SonarQube* plug-in.

On the ED Company[1] (ED), *SonarQube*[2] has been used to keep track of TD and as a driver to source code refactoring. During one of our projects, we found the necessity to implement our own rules**,** based on the Clean Architecture [Martin 2017], to support the refactoring in the development stage of the software. Therefore, this paper offers a report on the conception and creation of a SonarQube plug-in at a software house project in Brazil. Such a *plug-in* accomplishes the first list of rules that the ED code should follow based on the Clean Architecture pattern. Roslyn has been used to analyze the AST of a project implementing that architecture, created a plug-in, and integrated it into our continuous integration process over Gitlab to analyze these rules automatically.

2. Background

2.1. Clean Architecture

Robert C. Martin proposed the Clean Architecture as an alternative to incorporate ideas from other layered architecture models, like the Hexagonal and Onion Architectures. According to [Martin 2017], the architecture objective is the separation of concerns by separating software into layers, separating business rules and interfaces.

Each layer in the Clean Architecture model represents a different software part. The outer layers represent mechanisms, as Web Frameworks and Databases. Business rules and policies are in the inner layers. The fundamental rule for this architecture is the *dependency rule,* which establishes that "*... the name of something declared in an outer circle must not be mentioned by the code in an inner circle. That includes functions, classes, variables, or any other named software entity*" [Martin 2017]. The code dependencies can only point inwards.

[1] https://ed.company/
[2] https://www.sonarqube.org

This architecture is not limited to the four layers presented in the Clean Architecture; practitioners can tailor and add new layers following the dependency rule. To allow data to cross the boundaries of each circle, developers can create *Data Transfer Objects* (DTOs), or use the function arguments or *HashMap's* to pass data through layers. An essential issue is not passing entire Entities or Database Columns.

2.2. The Case Project

The case project is comprised of the evolution of a medium-sized project for a large Brazilian company that operates on the insurance, financial, and private pension market. It was the first release of the second phase of a broader project. The project had a duration of approximately three months, from 15/07/2019 to 03/10/2019. Five micro-releases with functionalities have been delivered, and after that, there was time to work on the code refactoring activities, which extended the project with two more releases. The Electric Dreams team comprises a project leader, a technical leader, six back-end developers, three testers, and four front-end developers. It composes a group of fifteen professionals, geographically distributed between the states of Rio de Janeiro and Santa Catarina, Brazil.

C#, Microsoft SQL Server, and Microsoft's coding tools, like Visual Studio, represent the primary development platform. For three months, the development team developed approximately thirty-six thousand lines of code, including interfaces for external services, use cases, and business rules. Since this project regards a private pension market, the development team needed to take training sessions on the complex business rules concerned with the problem domain. The technical leader reviewed all the developed code. Next, the testing team tested all the systems. Later, the technological leader oversaw merging the development branches into the stable branch.

2.3. Capturing Technical Debt

The first step to start building the project's TD model was to create a view of the "right code" for the project. To do that, we used our case project.

Using the set of code rules defined by our client and rules defined by the Clean Architecture, the development team aggregated 52 rules for the first version of the plug-in. Each rule is composed of the following parameters

- **Architecture Layer:** The architectural layer that the rule is related to. It can be General, API, Application, Cache, Use Case, Domain, Specification, Event, Dependency Injections, Repository, External Services, and Hosted Services.
- **Identifier:** A unique identifier that represents the rule. Each identifier has a numbering pattern related to the architecture layer rule, followed by an incremental numbering sequence.
- **Description:** A short rule description that will appear on the *SonarQube* interface. It can be used to guide and exemplify to developers on how they can fix an issue.
- **Presentation Name:** The phrase presented to developers when a rule generates an issue.

- **Severity:** The rule of severity. It can assume values of error, warning, info, or hidden.
- **Analysis Context:** The *Roslyn* context that will trigger the rule.
- **Debt Remediation Cost:** An estimated time to fix an issue generated by a specific rule.

3. Preliminary Results

Using the released version of the plug-in, the developers started using it daily after each commit, while a separate group of developers focused on refactoring each of the issues reported by SonarQube. Each commit passed through four stages; these steps are build, unit and integration testing, and using SonarQube to scan the code and generate the report. If the commit was on a release branch, the current version of the software was deployed to a staging environment.

The first run on the release branch revealed 395 code debt issues, with a code debt estimated of 13 workdays. Since *SonarQube* generates reports addressing each issue, the developers could use it as a guide to starting the refactoring process. To tackle many debts created during the development, the team decided to start with the easier in order of difficulty; that way, the first issues solved were related to naming and unused imports left on code. After the first refactoring cycle, we still had 261 code debt issues that required less trivial solutions, like extracting dependencies or change classes that were horizontally traversing the architecture.

The software overall code quality has been improved by continuously refactoring the code over a month approximately. During that time, it was looked to decrease the total number of code smell issues. The code smells reduced until it reached zero, but it grew back after new merges with code from other development branches. After each merge, the refactoring team would start working on the new issues reported until the number of issues decreased to zero again.

References

Brown, Nanette, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, Nico Zazworka, Yuanfang Cai, et al. 2010. "Managing Technical Debt in Software-Reliant Systems." In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research - FoSER '10*, 47. Santa Fe, New Mexico, USA: ACM Press. doi:10.1145/1882362.1882373.

Cunningham, Ward. 1992. "The WyCash Portfolio Management System." *SIGPLAN OOPS Mess.* 4 (2): 29–30. doi:10.1145/157710.157715.

Li, Zengyang, Paris Avgeriou, and Peng Liang. 2015. "A Systematic Mapping Study on Technical Debt and Its Management." *Journal of Systems and Software* 101 (March): 193–220. doi:10.1016/j.jss.2014.12.027.

Martin, Robert C., 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall Press.