Comparing The Accumulation Of Technical Debt Between Two Applications Developed With Spring Web MVC And Apache Struts 2

Georgios Digkas [a]
g.digkas@rug.nl

Alexander Chatzigeorgiou [b]
achat@uom.gr

Apostolos Ampatzoglou [a]
a.ampatzoglou@rug.nl

Paris Avgeriou [a]
paris@cs.rug.nl

[a] Department of Mathematics and Computer Science, University of Groningen, Netherlands
[b] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

Abstract

This paper presents the results of an observational study that investigates the differences between two widely used software development frameworks for Java EE applications. Also, it presents the accumulation of Technical Debt and the evolution of code quality metrics of software developed using these frameworks. Considering that web applications hold the lion's share of today's IT industry, this study focuses on two widely popular Java EE frameworks, namely Spring Web MVC Framework and Apache Struts 2. In particular, we have developed one system over four versions in both frameworks while monitoring Technical Debt and code quality metrics. The findings indicate that software developed based on these frameworks is relatively free of Technical Debt. Moreover, we have not noticed any significant differences between the two frameworks in terms of Technical Debt, from the perspective of source code metrics. Finally, conducting this study, we realized that if the framework is properly used it can potentiality lead to high quality and maintainable systems.

1 Introduction

Technical Debt (TD) is a software engineering metaphor that has been coined by Ward Cunningham [Cun92] in 1992 as: "*Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this*

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016 (sattose.org), Bergen, Norway, 11-13 July 2016, published at http://ceur-ws.org

transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise". Additionally, software engineers and developers often compromise internal quality to follow shortcuts in the development to meet close deadlines, at the expense of long-term system understandability and maintainability. This happens both in industrial settings where deadlines are pushing people to deliver debt-ridden solutions as well as in open-source projects where release cycles or pure desire to implement a feature fast introduce similar shortcuts.

One way to build systems faster is to use a framework. A framework is a set of classes and libraries which have been written by other developers and the programmers can use them to create enterprise applications. Currently, there is a plethora of frameworks and the number is growing. The developer can gain several benefits from the usage of a framework such as: easier and faster software development a more robust architecture and extensibility. The frameworks provide the infrastructure allowing the developer to focus on the business logic to be implemented. Proponents of frameworks claim that building software using frameworks results in higher quality code/products. This is happening because most of the frameworks are based on widely-acceptable design patterns.

The purpose of this study is to compare two of the most widely-used frameworks for Java EE applications. The analysis focuses on the resulting software quality and Technical Debt.

2 Background Information

Although the research community has not agreed upon a common way of assessing technical debt, there are some tools that provide TD estimates such as the ones that implement the SQALE method [Let12]. SQALE is a generic and language independent method. SQALE stands for Software Quality Assessment based on Lifecycle Expectations and is delivered under a Creative Commons license. One of the most widely used tools for the calculation of TD is the SonarQube [son] platform. The calculation of TD is based on rules and issues. For each rule there is an estimation of time that is required to fix the corresponding issue. The SonarQube sums the estimated time for each of the issues and calculates the Technical Debt in man-days.

3 Case Study Design

The present study aims at investigating the question: "Are there any significant differences between the two most widely-used Java these EE frameworks, in terms of software quality and accumulation of Technical Debt?". The study aims to *analyze* software systems developed using these Java EE frameworks ***for the purpose*** of measuring software quality *with respect to* the estimated TD and selected metrics, *from the point of view* of software researchers *in the context of* web application development [BCR94].

In order to compare the two Java EE frameworks, we have developed two CRUD web applications with state-of-the-art technology. CRUD is a term that is used in computer programming and stands for Create, Read, Update and Delete tasks which are most often associated with the administration of a database. We have developed one application based on the Spring Web MVC Framework [Jo05] and the other one on Apache Struts 2 [str]. Both applications have evolved over four versions. In each generation we were adding the same new features and functionality into the two applications.

To evaluate the quality of both systems that we have developed, we used the metric suite that Chidamber and Kemerer [CK94] proposed, which we augmented with a set of 2 additional source code metrics that are complementary to them. Through metrics, project managers can have an overview of the evolution of the projects. Below, we provide the description of some of the metrics that we will analyze in section 4. Number Of Classes (NOC) counts the number of classes in a project. Lines Of Code (LOC) counts the number of lines of code (in a package, classes or method). This metric also includes the white-lines and the commented lines. Weighted Methods Per Class (WMC) calculates the sum of complexity for all methods of a class. Each method is weighted by its cyclomatic complexity. Only methods specified in a class are included, that is, any methods

inherited from a parent are excluded. Coupling Between Object Classes (CBO) calculates the number of other classes to which a class is coupled. Primitive types, types from the java.lang package and supertypes are not counted. Response for a Class (RFC) calculates the number of methods that can potentially be executed in response to a message received by an object of that class. This measure is calculated as the 'Number Of Operations'+ 'Number Of Remote Methods'. Lack of Cohesion of Methods (LCOM) calculates the lack of cohesion of methods in a class with respect to its attributes.

Moreover, we have used the SonarQube platform to calculate the Technical Debt of these projects. During the development of the two projects, we tried to follow as closely as possible the suggestions of the frameworks on how design and implementation should be carried out. In the final step of our observational study, we repaid the identified Technical Debt, measured the required time and compared the actual time with the time that SonarQube estimated that it would take us to repay the Technical Debt.

The application that we have developed simulates a simplified information system of a university and was developed incrementally in four versions, with increasing functionality. The functionality of the different versions is the following:

v.1 The user of the application has the ability to retrieve general information about the university. For example, she can see the courses that are taught (name, description, ECTS credits and semester), the professors of the university (name, surname and contact number) and also which courses are taught by each professor. Finally, she is able to retrieve some general information about the secretaries of the university (name, surname and contact number).

v.2 Authentication and role authorization was added to the project. Also, additional functionality allowed each of the users of the web site to update their personal details.

v.3 Added new functionality for the secretaries allowing them to create, modify/update, delete, assign and remove courses to the professors of the university. Finally, the secretaries have the ability to modify/update the data of all users.

v.4 The students that are logged into the system with their credentials, are able to update/modify their personal information, see the courses that they are enrolled in and the grades of the courses that they attended. Also, they have the ability to enroll to new courses as well as be removed from the courses that they are already enrolled. If a professor logs into the application with her credentials, she is able to update/modify her personal information and obtain a list with all students that are attending her courses and finally, she is able to assign grades to the students.

Figure 1 shows the evolution of the applications and the features which have been added to each version.

4 Results and Discussion

In this section we are going to discuss the results and the findings of our observational study. Firstly, we will discuss the metric results and then the Technical Debt for the two projects that we have built.

4.1 Source code metrics

As already mentioned the two Java EE projects have the same functionality and evolved over four versions. The goal was to investigate if the source code metrics will be the same or if there is a comparative advantage by using one of the frameworks. Figure 2 shows the charts of the six of the metrics that we calculated.

- Number Of Classes (NOC). The number of classes that exist in both projects is almost equal. This apparently is happening due to the fact that both projects have the same functionality. However, the project that was developed in Struts 2, has a smaller number of classes when

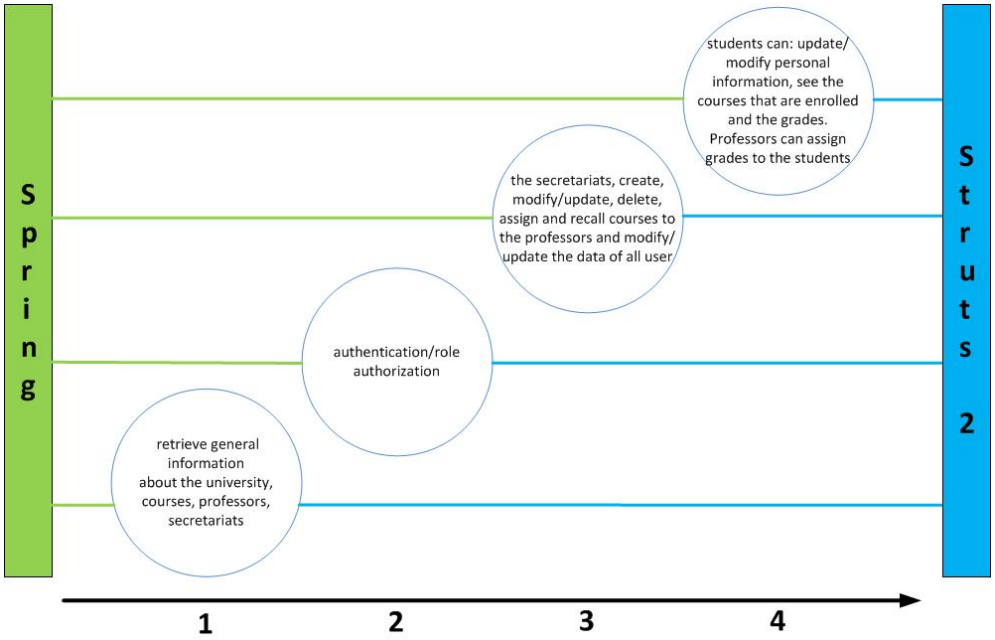

Figure 1: Versions

we compare it with the Spring. Their minimum difference is 1 and the maximum is 5. The maximum difference, occurred when because one of the Action classes of Struts 2, in Spring had to be broken into 3 classes. Moreover, in Spring we had to implement an additional class that had the role of wrapper. Finally, the Spring application has 2 extra classes, one to display the data to non-registered users and the other one to registered users. In Struts 2 there is one class to handle these Actions. Struts 2 gives us the opportunity to use Pointcuts and Wildcard expressions for the management of requests.

- Lines Of Code (LOC). From the LOC chart we can see that the Struts 2 application has about one hundred lines of code less as compared to that developed in Spring. This is in accordance to the lower number of classes.

- Weighted Methods Per Class (WMC). As we can see from the WMPC1 chart this metric for the Spring application does not change during the evolution of the project. But for the Struts 2 application this metric is increasing during the evolution of the application. Struts 2 implements the pull-MVC (or MVC 2) so it requires getters and setters for the view to be able to retrieve the data. Each getter and setter method, is increasing the complexity by 1. This is the reason that the Struts 2 application has higher complexity.

- Coupling between Objects (CBO). From the Figure we can observe that the average coupling for Struts is lower than that of Spring and also slightly increases from version to version. While for the Spring application there is a sizable increase from the second generation to the third and from there to the fourth. This happened because the Spring application has a larger number of classes. This had as a result an increase of the average coupling of the system.

- Response for a Class (RFC). The average value of the RFC metric for the project that is developed in Spring remains nearly constant in all versions of the project. Only the third generation it increased by a unit, while the fourth version it returns to the baseline. On the other hand, the average value of the RFC for Struts 2, is from the first version of the project 10 points higher when compared with that of Spring and we can also observe that

from generation to generation it has an upward trend. This is happening due to the fact that Struts 2 has a significantly lower number of Action classes that should manage the same number of requests.

- Lack of Cohesion of Methods (LCOM). The average lack of cohesion in the project that is developed in Struts 2, is far higher than in Spring, but it remains constant in all versions of the project. The improved cohesion in Spring is probably due to the larger number of smaller classes which therefore tend to be more cohesive.

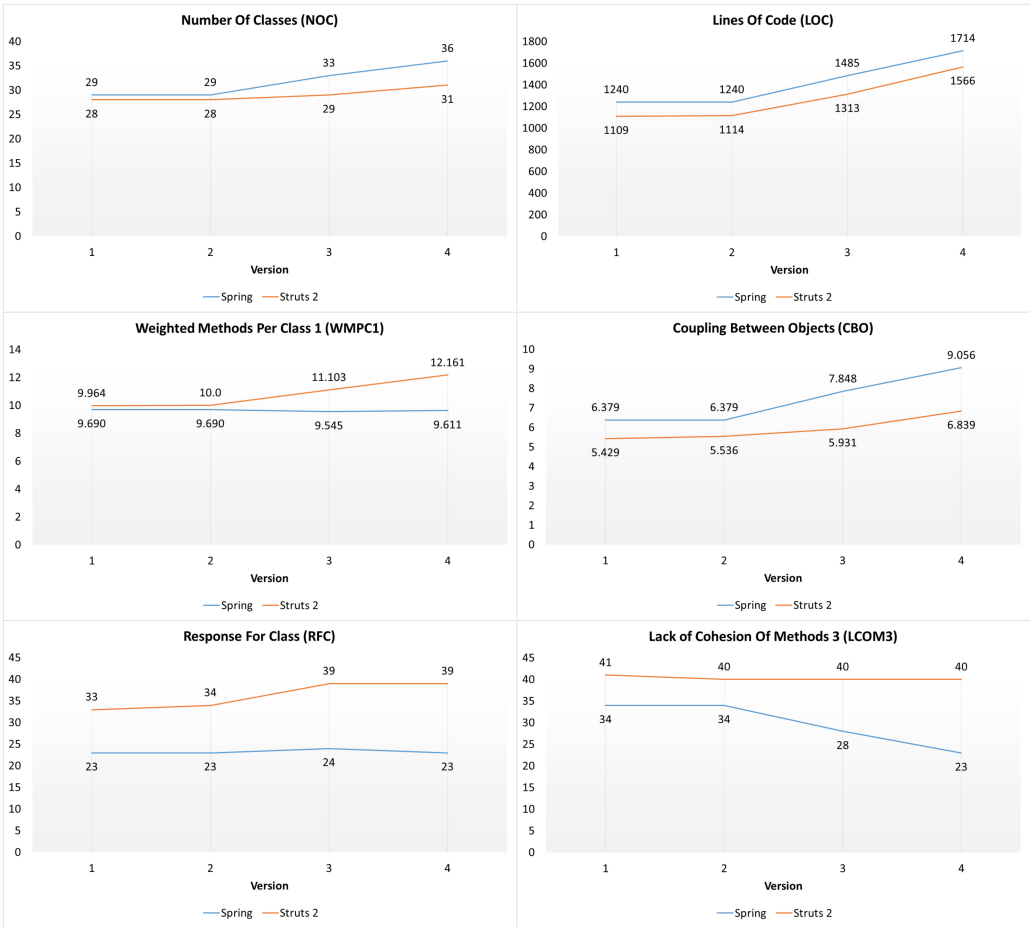

Figure 2: The evolution of a series of metrics over four versions of the two system implementations

4.2 Technical Debt Results

Table 1 summarizes the results concerning the technical debt of both systems along the four versions. As it can be observed, the accumulated technical debt is relatively low: the technical debt ratio (i.e. the estimated technical debt over the size of each application) does not exceed 1.5% for Spring and 2.7% for the Struts application and the corresponding SQALE rating is 'A'. For the Spring application no Blocker or Critical issues have been identified while for the Struts application a few critical issues have been identified. These issues mainly refer to the rule: "*Fields in a Serializable class must themselves be either Serializable or transient even if the class is never explicitly serialized or deserialized. That is because under load, most J2EE application frameworks*

flush objects to disk, and an allegedly Serializable object with non-transient, non-serializable data members could cause program crashes, and open the door to attackers". Each one of these issues increases the debt of the application by 30 minutes. These issues appeared due to the fact that it is a good practice for the Action classes (the Controllers) of a Struts application, to extend the ActionSupport class. The ActionSupport class implements the Serializable interface and this is the reason why the SonarQube counts them as issues.

Concerning the evolution of TD, the estimated effort to repay it and the TD ratio increase with the passage of versions. This should be mainly attributed to the fact that we are adding new features and functionality to our projects. In the final step of our exploratory study we repaid the TD and also we measured the time that took us to do the repayment. The actual time to resolve the reported issues was less than 2 hours. The time is significantly lower than the SonarQube estimates. Our general belief is that no tremendous improvements in quality have been incurred by repaying the accumulated TD.

Table 1: SonarQube report for Spring and Struts 2 applications

Version	Spring 1	Spring 2	Spring 3	Spring 4	Struts 1	Struts 2	Struts 3	Struts 4
Lines of code	1375	1455	1765	2046	1296	1303	1517	1801
Lines	1963	2062	2565	2828	1884	1893	2180	2535
Functions	166	172	194	212	164	165	182	201
Classes	28	29	33	36	28	28	29	31
Duplicated lines (%)	1.9%	1.8%	2.3%	4.4%	2.0%	2.0%	2.9%	2.5%
Duplication Lines	38	38	72	124	38	38	64	64
Duplication Blocks	2	2	4	8	2	2	4	4
Complexity	195	201	225	248	193	194	214	240
Complexity/Function	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
Complexity/Class	7.0	6.9	6.8	6.9	7.0	6.9	6.8	6.9
Issues	82	90	118	140	158	159	171	188
Blocker Issues	0	0	0	0	0	0	0	0
Critical Issues	0	0	0	0	2	2	6	15
Major Issues	13	17	25	33	90	91	95	98
Minor Issues	62	66	86	100	59	59	63	68
Info Issues	7	7	7	7	7	7	7	7
Technical Debt	7h 54min	1d	1d 4h	1d 7h	1d 7h	2d	2d 3h	3d
Technical Debt Ratio	1.1%	1.2%	1.4%	1.5%	2.4%	2.5%	2.5%	2.7%
SQALE Rating	A	A	A	A	A	A	A	A

5 Threats to Validity

In this section, we present and discuss possible threats to validity of our study. First of all, internal validity expresses to what extent the observed results are attributed to the performed intervention, and not to other factors. Then, reliability is linked to whether the case study is conducted and presented in such way that others can replicate it with the same results. Finally, external validity deals with possible threats when generalizing the findings of our study to the entire population.

The main threat to the internal validity is related to the developer who implemented the two applications. The developer had used the Spring MVC framework one time in the past, before he conducted this study. The main reliability threat of our exploratory study is that the two Java EE projects have been developed by the same developer. Concerning external validity, we have identified one possible threat. We developed and compared our Java EE project only in two of a variety of Java Web Frameworks. We chose these two frameworks because they are used widely and also implement the MVC pattern in different ways. One the one hand, Spring Web MVC is a push-MVC framework. On the other hand Struts 2 is a pull-MVC framework.

6 Conclusions and Future Work

This study presents an exploratory study to seek the benefits of framework-based development of Java EE applications. The main finding of this work is that framework-based development does not lead to serious issues leading to a relatively low technical debt. Moreover, after repaying the

TD no tremendous improvement to the quality of the software has been observed. Finally, the required effort for the repaying of TD was significantly lower than the corresponding estimates.

In terms of future work, it would be valuable to generalize this study by analyzing multiple projects and multiple types of data (i.e. source code metrics, issues, commits, etc). Framework-based development could be contrasted to non-framework-based applications to investigate if there is a significant difference between these two types of development.

References

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience factory. *Encyclopedia of software engineering*, 1994.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[Cun92] Ward Cunningham. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 29–30, New York, NY, USA, 1992. ACM.

[Jo05] Rod Johnson and others. Introduction to the spring framework. *TheServerSide. com*, 21:22, 2005.

[Let12] Jean-Louis Letouzey. The SQALE method for evaluating technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 31–36. IEEE Press, 2012.

[son] SonarQube. `http://www.sonarqube.org/`.

[str] Welcome to the Apache Struts project. `https://struts.apache.org/`.