STATIC CONSTRUCTS: EVOLUTION AND IMPACT ON SOFTWARE QUALITY ASPECT

PhD thesis – Summary

for obtaining the Scientific Title of PhD in Engineering from
Politehnica University Timișoara
in the Field of Computers and Information Technology

author eng. Cosmin MARȘAVINA
PhD Supervisor Prof.dr.habil.eng. Mihai V. MICEA

month January year 2022

The current thesis is comprised of 7 chapters, 173 pages, 56 tables, 79 figures and diagrams, and 102 bibliographic references. Its goal is to study 1) the way in which static construct are utilized in complex software project, 2) how they have evolved throughout the lifespan of a system, and 3) the degree to which they affect a series of software quality aspects.

Chapter 1 introduces the concepts that will be presented in the thesis. First, we discuss the problem that is going to be addressed. Then we specify the research questions that were formulated and highlight their importance. Additionally, we explain the relevance of our work and mention the contributions that we expect to bring. The main objectives that were set and the way in which we planned to achieve them are also discussed. The last section of this chapter describes how the rest of the thesis is structured and the content of each of the following chapters.

We begin by explaining the importance of testing; it represents a vital part of the software development life cycle. Studies have shown that more than half of the effort required for implementing complex software systems is spent on testing [1]. In this thesis, we will focus on unit testing in an object-oriented context as these tests are directly related to specific parts of the production code. There are 3 software quality aspects that are closely related to the testing process: testability, change-proneness, and defect-proneness. We start by defining them; testability is “the capability of the software product to enable modified software to be validated” [2]. Change- / defect-proneness are characteristics of software artifacts that represent their susceptibility to modifications / errors.

Design flaws are violations of design practices and principles that make software systems harder to understand, maintain, and evolve. Some of them have already been proven to have a negative impact on specific software quality aspects. However, there are still numerous other design flaws / quality aspects that have not been investigated thus far. Static constructs are a category of source code entities in which the *static* keyword is used. We have already managed to prove that some of these constructs have a detrimental effect on testability [3] and defect-proneness [4]. In this thesis, we want to investigate each type of static construct both in terms of presence / usage and regarding their impact on the 3 quality aspects presented above. For studying their presence and usage, we will not concentrate solely on the latest version of a system; the entire history of a project will be considered.

To investigate all these aspects, we formulated the following research questions:

- Are static constructs used in complex software systems? For this question, we try to establish: 1) if instances of static constructs actually do appear in the production code; 2) how are these instances utilized by other classes; 3) whether or not their clients are localized in a small number of packages or spread throughout the source code.

- How have static constructs evolved throughout the lifespan of a project? We are keen to observe: 1) if instances of certain types are still being created even though they were proven to be problematic; 2) whether or not the number of client classes increases as a system is growing in size.
- Do static constructs have a negative impact on software quality aspects? We want to study the effect of using static constructs on the software quality aspects of interest, namely: 1) testability, 2) change-proneness, and 3) defect-proneness.

By researching all these aspects and answering the 3 research questions, we expect to bring the following contributions:

- a general methodology for studying different types of design flaws, their evolution, and the impact they have on software quality;
- a model for quantifying class testability based on the quantity and the quality of its corresponding unit tests;
- a process for determining: 1) the fine-grained source code changes that were performed during a commit and 2) if errors were fixed in the respective commit;
- a tool that incorporates all these aspects;
- an empirical study through which we answer the proposed research questions.

In order to bring the contributions enumerated above, we have set a series of objectives that must be achieved. The main objectives would be:

1. studying the state of the art for the topics of interest, namely: design flaw detection (with an emphasis on static constructs) and evolution, models for quantifying software quality aspects, and design flaws that have a negative impact on the investigated aspects;
2. categorizing the static constructs and defining detection strategies through which we can identify instances of each type. Furthermore, analyzing the presence and usage of these instances both for the latest version of a project and for a number of versions throughout its entire lifespan;
3. developing procedures through which the investigated quality aspects (testability, change- and defect-proneness) can be evaluated. Additionally, establishing whether or not the static constructs from each category have an effect on them.

The last section of this chapter explains how the thesis is structured. For the remaining chapters we briefly discuss what each section contains. Furthermore, at the end of each chapter there is a fragment that summarizes all the things discussed in the respective chapter.

Chapter 2 presents a literature review on the state of the art in design flaw detection and evolution, models for evaluating software quality aspects, and design flaws that affect these aspects. In the first section, we discuss different detection strategies that have been proposed and tools developed towards this end. We focus on metrics-based strategies, such as the ones proposed in [5], because we will employ a similar approach to identify different types of static constructs. Another article of interest is [6], where besides the strategies the authors also mention repair techniques through which the detected design flaws can be removed. In the second part of this section, we present a series of smells that might appear in test classes; they represent deviations from the guidelines proposed to aid developers in creating adequate test suites. Just as for design flaws, different categories of test smells were identified and detection strategies have been proposed (e.g., in [7]). The presence of these problems in the test code has been correlated with the existence of design flaws in the production classes [8]; we will take this aspect into account when evaluating the quality of the testing performed on a particular class.

The second section addresses the way in which certain design flaws have evolved. There are a considerable amount of articles that investigate this aspect (such as [9]), but none of them have analyzed any kind of static construct. We will study the evolution of these constructs in a

similar manner to the one presented in [10], where we researched the co-evolution between production and test code.

In the next section, we discuss the models that have been proposed to evaluate the software quality aspects of interest. The majority of models for testability (e.g., the one described in [11]) start from the design of the classes / the entire system, not from the source code. In an article that compares different models for evaluating this quality aspect it was concluded 1) that there is no superior model and 2) that the model used should be selected based on the particularities of the analysis that is going to be conducted [12]. For change- / defect-proneness, most of the publications try to predict whether or not a certain class will be modified / repaired in the near future. There are very few articles that propose models for quantifying these 2 quality aspects based on the history of a system. One such example would be [13], where Java classes are categorized as defective / defect-free using 2 sets of metrics (product / process related).

The impact of certain design flaws on the 3 quality aspects mentioned above is discussed in the last section of the chapter. We could not find any articles that investigate this for different types of static constructs. One of the categories studied in [14] is mutable state (more specifically, static non-final attributes and singletons); the authors concluded that it may have an effect on the testability of software systems. In another paper [15], Hevery proposes a tool that can be used to evaluate this quality aspect; however, that tool does not start from Java source code, but rather from the bytecode.

In **Chapter 3** we explain the approach adopted in order to: 1) categorize and detect static constructs, 2) study their evolution, 3) quantify the 3 quality aspects, and 4) implement the entire data collection process. Considering the variable granularity of the constructs that use the *static* keyword, we decided to perform a multilevel categorization. At class level, we distinguish between 3 types of static constructs: singletons (both stateful and stateless), utility classes, and the rest of the classes that contain smaller instances. For the latter category, the categorization was done based on the types of the instances present, namely: static methods that access / modify the state of the class in which they are declared, static methods that solely operate on parameters, static non-final attributes, constants, and static initialization blocks. For each of the aforementioned categories we proposed detection strategies through which the respective instances can be identified. For example, 3 conditions have to be met in order for a class to be categorized as a singleton:

1. there are no public constructors within the class;
2. the class has a private static attribute (the "singleton instance") and a public static accessor method that performs lazy instantiation on this attribute and returns it;
3. the aforementioned method is the only way in which the respective attribute can be accessed.

This detection strategy corresponds to the general form of Singleton (Lazy Instantiation); the strategy was extended so that it can detect a series of variations of the pattern (the ones discussed in [16]). Furthermore, the singletons were divided into 2 categories, stateful and stateless.

The second section of this chapter presents the process through which the evolution of static constructs is studied. We relied on Git to obtain the data necessary for performing this analysis. For each of the analyzed systems, the commits are sampled with a frequency of 1 commit per month. Then we iterate over the remaining commits and determine the differences between each commit and the corresponding one from the previous month; the following differences are recorded: the total number of instances from each category, the number of client classes for each instance, and the average number of clients for the entire project. Additional data regarding each static construct / all of its clients are also recorded along with other useful information (such as a class being marked as Deprecated).

The next section describes the model for quantifying class testability. Unlike the models that have been proposed until now, our model evaluates the testability of a production class based on the quantity and the quality of the corresponding unit tests. Therefore, this aspect is estimated both from a quantitative and from a qualitative perspective. The metrics used for measuring quantity are 1) line coverage and 2) the percentage of production methods addressed by unit tests. The coverage data are obtained through *JaCoCo* [17], a tool that can be utilized on any type of project (Maven, Gradle, etc.) and provides a detailed report which also includes a series of class / method complexity measurements. To evaluate the quality of the testing that was performed on a class, we consider certain smells that might appear in the corresponding test class. Test smells are identified using *tsDetect* [18]; this tool was extended so that it can specify which of the 19 detectable smells are present in a particular unit test. Two metrics are calculated once again: 1) the percentage of unit tests in which smells appear and 2) the number of different types of smells that exist in a test class. Based on these 4 metrics we calculate 2 scores (quantitative and qualitative) which are then aggregated to obtain the overall testability score. This score can be used to compare a production class to another (which is similar to it in terms of size and complexity); if the first class has a higher overall score, then this means that the class is easier to test. To determine the 2 scores mentioned before, we utilize a series of intervals corresponding to threshold values (which are explained in detail in the respective section).

To identify the classes that are susceptible to modifications / errors, we use the procedures described in the fourth section of this chapter. The only difference between the two is that in the procedure for defect-proneness we solely consider the commits that were categorized as bug-fixes. For this categorization, we utilize 2 types of information: 1) the one available in the commit message and 2) additional data collected from the Jira issue tracker associated with the project. The proposed procedure is thoroughly explained in the thesis; with it we can determine with high accuracy the commits in which defects were repaired. To evaluate the 2 quality aspects we must be able to establish exactly which modifications were performed during a commit. To this end we extract the fine-grained source code changes made to the source code using *ChangeDistiller* [19]; these include: the entity that was modified (class, method, or attribute), the type of the change (e.g., modifying a conditional statement in a method), and other details related to it (such as severity). The entire commit history of a particular class is analyzed and compared to the histories of similar classes. If that class 1) was changed more frequently or 2) more modifications were performed on it, then it can be considered as having a higher susceptibility to changes. As previously mentioned, error-proneness is evaluated based on the same metrics, but only bug-fix commits are taken into account when these metrics are calculated.

The last section presents the entire data collection process along with the tool that was developed. This tool is called *DFAnalized* and is implemented as an extension to *Patrools* [20], an eclipse plugin that is already capable of calculating some of the required metrics. We designed the tool so that it has a modular structure; several modules can be combined in order for it to perform the desired analysis. One of the modules contains the detection strategies for the design flaws studied (e.g., singleton). Another module is responsible for quantifying the quality aspect that is investigated. If we want to also analyse the evolution of the respective flaw, then we just have to add the corresponding module. The modules are configurable and can be easily extendable; for example, we could study another design flaw by creating a new module with the appropriate detection strategies.

Chapter 4 explains how the empirical study was conducted. It starts by presenting the main goal of the study together with the hypotheses that were formulated. Afterwards, we describe the independent and dependent variables for each hypothesis along with the procedures through which they can be measured. The criteria based on which we selected the projects

included in the study are also discussed. The last section of this chapter presents the analyses that were conducted as part of the empirical study.

As was explained previously, the main goal of this thesis is to obtain a better understanding of 1) static constructs, 2) their evolution, and 3) the quality aspects on which they have a negative impact. To reach our goal, we formulated 3 research questions corresponding to each of these aspects. The main objective of the empirical study is to obtain answers to the research questions. This is why we analyzed the aspects in isolation. For each of the research questions we formulated 2 hypotheses for the the null and alternative variants. As an example, for the first research question the null variant would be "Static constructs rarely appear in complex software systems.", while the alternative one is "Static constructs are present in the production code and there are other classes that utilize them.".

Each of the hypotheses is discussed in detail and then we present the independent and dependent variables for them. For the pair of hypotheses corresponding to the first research question the independent variables are the specific characteristics of the studied system, while the dependent ones would be the different types of static constructs that appear / are utilized by the production classes. Besides the general characteristics, such as size and complexity, we also investigate several other characteristics (e.g., key functionalities or the fact that a project is structured as a library).

When choosing the systems for the empirical study we took into account a number of criteria, including the ones discussed in [21]. The projects needed to be:

1. relevant in terms of size and complexity;
2. available through Git and have a considerable number of versions;
3. extensively covered by unit tests;
4. associated with Jira issue trackers that contain the problems encountered during their development.

Based on these criteria, we selected 11 open-source systems to include in the study. We tried to choose projects that differ in terms of 1) size and complexity, 2) development practices, and 3) the effort that was put into testing them, while still meeting the criteria enumerated above. A preliminary analysis on these aspects is included in the respective section.

In the last section, we discuss the analyses performed on the systems presented before, namely:

- a preliminary analysis on the size and structure of the systems, their history, and the effort that was put into testing them;
- an analysis on the presence / usage of different types of static constructs;
- an analysis on the evolution of each of the respective types;
- 3 analyses on the impact of static constructs on the quality aspects.

In **Chapter 5** we present the results obtained for the analyses mentioned. This chapter only includes raw results; their interpretation is provided in the following chapter. In the first section, we study the static constructs identified in the latest version of each of the 11 projects. Besides the number of instances, we wanted to investigate the way in which they are utilized and observe if the client classes are localized in a limited number of packages or spread throughout the entire system. For each category of static constructs, we compared the clients (and their localization) to those of the remaining entities of the same type; for example, static methods that access the attributes of the classes in which they are declared / that solely operate on parameters are compared to the non-static methods of the system with regard to these 2 aspects. For each of the 11 projects, we provided a table that contains all the data mentioned above.

The second section of this chapter addresses the evolution of different types of static constructs. Just as for the previous analysis, we did not solely focus on the number of instances of a certain type that are present in a version of the system; we tried to determine the reasons

why particular instances (or their clients) were added / removed. Additionally, for the class-level constructs (singletons and utility classes) we also investigated how they were used throughout the lifespan of a project. For each type of static construct, we created a graph that depicts the total number of instances / the percentage of production classes that utilize instances of that type (y-axis) over time (x-axis). When unexpected situations occur, such as a significant decrease in the number of instances of interest or a class losing most of its clients, we tried to understand the reasoning behind such decisions.

The next section analyzes the impact of static constructs on class testability. We rely on the testability score to compare the classes that contain static constructs to other classes that are similar to them in terms of size and complexity. As was explained in Chapter 3, this comparison is made both from a quantitative and from a qualitative perspective. Quantity is evaluated based on 1) line coverage and 2) the percentage of methods from a class that are tested. For quality we determine 1) the percentage of unit tests in which smells exist and 2) the number of different types of smells that appear in a test class. Using these metrics we calculate 2 scores (a quantitative and a qualitative one) which are then aggregated into the overall testability score. For each system, we provided a table that contains these 3 scores for the classes with different types of static constructs / the similar classes.

The last section of this chapter presents the results corresponding to their impact on change- / error-proneness. In both evaluations we compare the average number of modifications that were performed on the classes that contain a certain type of static construct / the classes that were categorized as similar to them. We also calculated the average number of changes per commit to determine whether or not the classes with instances of interest were modified in more commits than other similar classes. Additionally, we wanted to observe what types of fine-grained modifications occur most frequently (top 5 change types) and establish if the rankings are different for the classes with various categories of static constructs / similar classes. For each project we created 2 tables, one for change-proneness and another for defect-proneness. As specified in Chapter 3, for the second quality aspect we take into account only the commits that were categorized as bug-fixes.

Chapter 6 discusses 1) the interpretation of the results with regard to the research questions that were formulated and 2) the factors that could be considered threats to the validity of the empirical study and the obtained results. We start by looking at the results as a whole, thereby being able to draw meaningful conclusions. For the first research question we observed that 1) static constructs are present in all 11 systems that were analyzed; 2) constants are by far the most common type of static construct, followed by static methods (of both types) and utility classes; static non-final attributes, singletons, and static initialization blocks appear less often, especially in the smaller projects; 3) the number of clients of static constructs and their localization are not much different when compared to other similar constructs (in most cases).

Regarding the second research question, the main observations would be: 1) that there are certain categories of static constructs (such as static non-final attributes and singletons) for which fewer and fewer instances appear in recent versions (compared to those from the beginning / halfway through the development process) and 2) that we identified many cases in which these instances lost most of their clients (or even all of them) before being removed themselves.

For the last research question we established that: 1) static non-final attributes, stateful singletons, and static methods that access state have the highest negative impact on testability; 2) all static construct categories with the exception of constants and static methods that solely operate on parameters affect change-proneness, although for stateless singletons and utility classes the results are contradictory for different types of systems; 3) for error-proneness, the impact of static constructs is not as significant as for change-proneness.

In the second section of this chapter, we present the factors that threaten the validity of the empirical study and the results; we also discuss ways in which we tried to mitigate them. The factors are grouped into 3 categories (based on the categorization proposed in [21]): construction, internal, and external threats. Those from the first category might appear due to problems in the code that was developed to collect the data necessary for the analyses performed. To avoid such problems, the proposed approach was carefully tested using several small-scale systems created specifically for this purpose. As an example, for the detection strategies we added instances from each category in various combinations to ensure that they are identified correctly. Furthermore, we manually checked all the data obtained; to the best of our knowledge, it is correct and complete. The threats from the second category (the internal ones) appear when modifications in the dependent variables cannot be attributed to changes in the independent ones. The main threats from this category are confounding factors, more specifically other variables that could mask an actual association or falsely prove an apparent association between the independent and dependent variables. It is very difficult to identify all the factors that have an impact on these variables, but we tried to discuss as many as possible. As an example, for the first hypothesis there might be other characteristics of a system that affect the presence / usage of different types of static constructs.

We identified a series of external threats that are related to the results of the empirical study, more specifically that they are not generalizable to other settings. A first limitation of this study is that all 11 systems are open-source. We hope to obtain access to at least 2 commercial projects in the near future, thereby being able to eliminate this threat. Another limitation would be that all the systems are developed in Java by following an object-oriented approach. We are planning to reimplement the tool so that we can analyses projects developed in other object-oriented languages (e.g., C# and C++) or through different programming paradigms (such as functional programming). Another factor that should be taken into account is the high granularity of the analyses performed. The study could have been a lot more detailed if we would have analyzed everything at method level; this represent one of the directions on which we will be concentrating our efforts from now on.

Chapter 7 contains the conclusions and future work directions. In the first section, we reiterate the contributions brought, namely:

- the methodology for studying the evolution and the impact on software quality of any design flaw;
- the model for quantifying class testability;
- the process for identifying commits in which bugs were fixed and determining the fine-grained changes that occur between commits;
- the tool for investigating the aspects of interest;
- the empirical study through which we obtain answers to the research questions for different types of static constructs.

All in all, the main goal of the thesis was achieved; we managed to obtain a better understanding of static construct presence / usage, the way in which instances of different types have evolved, and the impact they have on the 3 software quality aspects studied (testability, change- and defect-proneness).

By analyzing all the aspects presented before, we were able to draw a series of relevant conclusions. The main ones would be:

- that static constructs 1) are present all throughout the production code and 2) are frequently utilized by other classes;
- that they are started to be used less once a project reaches maturity (compared to the earlier stages of its development);

- that certain types of static constructs, such as mutable global state instances (stateful singletons and static non-final attributes) or static methods that access / modify their class's state, have a negative impact on the 3 quality aspects investigated.

  In the next section, we reflect on what we have accomplished and explain what could have been done better. We end the chapter with some promising future work directions that we are currently considering:
- extending the empirical study by adding new systems (commercial ones and projects implemented in other programming languages / by following other paradigms / through different development methodologies);
- investigating other design flaws, such as object instantiations in constructors / methods or Law of Demeter violations;
- perfecting the models through which we quantify the 3 software quality aspects (e.g., adding more metrics to the testability score);
- studying all these aspects at a lower level of granularity;
- proposing repair techniques for the problematic parts of both production and test code.

References

[1] Brooks Jr, Frederick P. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E. Pearson Education India, 1995.

[2] IEEE Standards Coordinating Committee. "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos." CA: IEEE Computer Society 169 (1990).

[3] Marsavina, Cosmin. "Studying the Evolution of Static Methods and their Effect on Class Testability." *2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 2020.

[4] Marsavina, Cosmin. "Understanding the Impact of Mutable Global State on the Defect Proneness of Object-Oriented Systems." *2020 IEEE 14th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 2020.

[5] Marinescu, Radu. "Detection strategies: Metrics-based rules for detecting design flaws." Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. IEEE, 2004.

[6] Kessentini, Marouane, et al. "Design defects detection and correction by example." Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. IEEE, 2011.

[7] Van Rompaey, Bart, et al. "On the detection of test smells: A metrics-based approach for general fixture and eager test." *IEEE Transactions on Software Engineering* 33.12 (2007): 800-817.

[8] Tahir, Amjed. *A Study on Software Testability and the Quality of Testing In Object-Oriented Systems*. Diss. University of Otago, 2016.

[9] S. Vaucher, F. Khomh, N. Moha, and Y. G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes." *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009.

[10] Marsavina, Cosmin, Daniele Romano, and Andy Zaidman. "Studying fine-grained co-evolution patterns of production and test code." *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014.

[11] Mouchawrab, Samar, Lionel C. Briand, and Yvan Labiche. "A measurement framework for object-oriented software testability." Information and software technology 47.15 (2005):

979-997.

[12] Nikfard, Pourya, et al. "An Empirical Analysis of a Testability Model." Informatics and Creative Multimedia (ICICM), 2013 International Conference on. IEEE, 2013.

[13] Moser, Raimund, Witold Pedrycz, and Giancarlo Succi. ”A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction.” Proceedings of the 30th international conference on Software engineering. 2008.

[14] Wolter, Jonathan, Russ Ruffer, and Miško Hevery. "Guide: Writing testable code." (2009): 1-38.

[15] Hevery, Misko. "Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process." *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2008.

[16] K. Stencel and P. Węgrzynowicz, “Implementation variants of the singleton design pattern.” *OTM Confederated International Conferences* “*On the Move to Meaningful Internet Systems*.” Springer, Berlin, Heidelberg, 2008.

[17] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, “Eclemma-jacoco java code coverage library.” (2011).

[18] A. Peruma, et al. “tsDetect: an open source test smells detection tool.” *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.

[19] H. C. Gall, B. Fluri, and M. Pinzger, “Change analysis with evolizer and changedistiller.” *IEEE software* 26.1 (2009): 26-33.

[20] P. F. Mihancea, “Patrools: Visualizing the Polymorphic Usage of Class Hierarchies.” *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 2010.

[21] L. S. Pinto, S. Sinha, and A. Orso, “Understanding myths and realities of test-suite evolution,” in Symposium on the Foundations of Software Engineering (FSE). ACM, 2012, p. 33.