Actionable Measurements – Improving The Actionability of Architecture Level Software Quality Violations

Wojciech Czabański
Institute for Informatics
University of Amsterdam
Amsterdam, the Netherlands
Email: wojciech.czabanski@gmail.com

Magiel Bruntink
Software Improvement Group
Amsterdam, the Netherlands
Email: m.bruntink@sig.eu

Paul Martin
Institute for Informatics
University of Amsterdam
Amsterdam, the Netherlands
Email: p.w.martin@uva.nl

***Abstract*—When system components become more coupled over time, more effort must be dedicated to software architecture refactoring. Tightly coupled components present higher risk—they make the system more difficult to understand, test and modify. In order to allocate the refactoring effort effectively, it is crucial to identify how severely components are coupled and which areas of the system involve the most risk to modify.**

In this paper we apply the concept of architecture hotspots together with the Software Improvement Group Maintainability Model to identify violations in architecture design. We propose a prototype tool that can identify and present architecture smells to developers as refactoring recommendations. We then apply the tool to open-source projects, validating our approach through interviews with developers. Developers found the hotspots comprehensible and relevant, but there is room for improvement with regards to actionability.

I. Introduction

Software maintainability is an internal quality of a software product that describes the effort required to maintain a software system. Low maintainability is connected with low comprehensibility. Glass argues that the most challenging part of maintaining software is understanding the existing product [1]. What follows is that code which is hard to understand is also difficult to modify in a controlled manner and test for defects. If changes are difficult to introduce and code hard to understand, the probability of bugs being introduced is very high, which raises the cost of further developing and maintaining the system.

We focus on the Maintainability Model developed by the Software Improvement Group (SIG) [2]. From this model, 10 guidelines have been derived to help developers quickly evaluate the maintainability of their code and provide actionable suggestions to increase its quality, such as keeping complexity of units low and interfaces small. The guidelines are implemented in the Better Code Hub tool [3], which applies them to provide feedback to developers. In particular we look at *component independence*, because it is considered the most challenging to improve, based on user feedback. Our goal is to provide developers with more actionable feedback in addition to the diagnosis provided by Better Code Hub, so that they can improve the maintainability of the code.

Currently, Better Code Hub provides an overview of components and the interactions between them, such as incoming and outgoing calls to and from modules in other components. It does not however provide specific guidance as to how the developer can reduce the component coupling and improve their independence. Attempts have been made to generate suggestions for improving modularity by suggesting move module refactoring operations, framing the problem as a multi-objective search [4]. Such refactoring operations however may either not improve modularity or make the codebase less semantically consistent. Identifying patterns in poorly modularized code can be a starting point for devising better recommendations as to how the components can be decoupled better. Thus we apply the *architecture hotspot patterns* described by Mo et al. [5] to conduct a study on open source projects in order to evaluate whether hotspots can be found in open source projects and used to provide refactoring recommendations. Furthermore we investigate whether presenting quality violations based on hotspots helps developers decrease coupling between components. In order to validate our approach, we construct a hotspot detector, integrate it with the Better Code Hub analysis tool and visualise the hotspots. Based on initial feedback from developers, indicating that the suggestions are comprehensible and relevant we finally consider how to build upon our work in future. We look to improve the tool by adding more detailed information which triggers the hotspot detection.

II. Background

Program analysis is the process of analysing the behaviour of a software program with regards to a certain properties such as correctness, robustness or maintainability [6]. There exist a number of means of program analysis already defined in research literature, including both static and dynamic analysis, maintainability guidelines and detection of ‘code smells’. We survey a few of these approaches below.

A. Static and dynamic program analysis

Source code is commonly used as input for static analysis tools. In certain cases other inputs are used as well such as revision history. Syntactic analysis and software metrics computation involves analysing the source code of a system, often represented as a graph. Examples of tools for obtaining the source code graph and metrics include *Understand*[1], the *Software Analysis Toolkit* from SIG[2] and *SonarQube* [7]. Our intention was to improve the actionability of measurements. In this respect, *SonarQube* was aimed at integration within a CI/CD pipeline, making it difficult to use in a research setting, because of the existing pipeline and the time limitations of the project make it unfeasible to modify it. *Understand* exported the dependency graph to a commonly used format and supported a variety of programming languages, but was challenging to integrate with the SIG infrastructure, which left us choosing the Software Analysis Toolkit to pre-process source code for further analysis.

We also investigated dynamic analysis methods reviewing tools such as Valgrind [8], Google Sanitizers [9] and Daikon [10]. We chose to focus however on analysing source code only—to use dynamic analysis, the executable for every project would need to be built locally. In addition to that, the reviewed tools detect possible faults in the code as opposed to analysing maintainability.

B. SIG Maintainability Model

SIG developed a maintainability model for software based on empirical software engineering research [11]. They use a in-house tool, the Software Analysis Toolkit, to conduct static analyses of software systems. The Maintainability Model is also accessible for GitHub developers through the Better Code Hub tool, which conducts an analysis of a repository and evaluates it against the ten SIG guidelines [3]. In our paper we focus on the 'Couple Architecture Components Loosely' guideline, which advises minimising the number of throughput components. These have a high fan-in/fan-out values [12]. Similarly to modules, components that act as intermediaries are more tightly coupled and more difficult to maintain in isolation.

C. Software design defect classifications

Low maintainability can manifest itself by the presence of certain antipatterns, called 'code smells'. The concept of 'code smells' as software design defects was popularised by Fowler [13]. We looked into both architecture and design smells. Suranarayana and Sharma proposed that architecture smells represent design violations impacting both component and system levels [14]. Sharma provided the definition of a design smell [15]; Fontana et al. investigated Java projects for recurring patterns of architecturally relevant code anomalies [16]. Architecture hotspot smells are code anomalies introduced in the paper from Mo et al. which are related to an increased software defect density [5]. Macia et al. designed a DSL for describing architectural concerns and code anomalies [17]. In addition to the source code and metrics, they use the metadata defined using a DSL to detect code anomalies in a tool (SCOOP). Martin focused on framing component design guidelines using 3 principles; violating those principles constitutes an architectural smell. [18] Garcia et al. define four architecture smells [19].

We believe that connecting maintainability measurements with architecture smells will allow us to provide more actionable and relevant refactoring recommendations for developers using Better Code Hub compared with relying on metrics alone. It will also make it possible to offer advice on how to deal with the detected type of smell. Only the classification from Mo et al. draws a clear connection between the architectural smells and maintainability, which is why we chose to use it to enhance the refactoring recommendations generated by Better Code Hub [5].

[1]https://scitools.com/

[2]https://www.sig.eu/

Table I
HOTSPOT INSTANCES OVERVIEW IN SELECTED SYSTEMS

System	Language	LOC (k)	Unhealthy Inheritances (files)	Cross-module cycles (files)	Package cycles (files)
Bitcoin	C++	120	16 (75)	31 (108)	49 (117)
Jenkins	Java	100	80 (170)	10 (403)	513 (372)
jME	Java	240	69 (436)	59 (402)	335 (410)
JustDecompileEngine	C#	115	79 (290)	8 (205)	92 (89)
nunit	C#	59	24 (94)	6 (62)	62 (74)
openHistorian	C#	72	12 (37)	31 (89)	63 (114)
OpenRA	C#	110	19 (150)	35 (273)	202 (206)
pdfbox	Java	150	64 (252)	23 (379)	261 (301)
Pinta	C#	54	17 (57)	12 (112)	109 (91)
ShareX	C#	95	11 (76)	38 (205)	189 (248)

III. EXPERIMENTS

A. Data sources

We selected a number of GitHub repositories that are both available as open source projects and contain the majority of code in languages that are supported by Better Code Hub as data sources to validate our approach. The projects we targeted needed to have between 50k and 200k lines of code, be at least three years old and be written in a strongly and statically typed language supporting inheritance (e.g. Java, C# or C++).

B. Hotspot distribution

We used the *Understand* code analyser to generate source graphs which we then fed into an existing tool called *Titan* tool [20], which identifies architecture hotspots. We aggregated the hotspot quantities and types per analysed system in Table I. The file count indicates how many distinct files contain hotspots. A file can be a part of multiple hotspots, but we count the files only once.

In order to reason about the impact of hotspots on the overall maintainability of projects, we compare the number

Table II
HOTSPOT IMPACT ON SELECTED SYSTEMS

System	Files analyzed	Files affected by hotspots	% Files affected by hotspots	CI measured by BCH
Bitcoin	675	117	17.33%	0.9894
Jenkins	1112	403	36.24%	0.9868
jME	2077	436	20.99%	0.6812
JustDecompileEngine	814	290	35.62%	0.8311
nunit	781	94	12.03%	0.6329
openHistorian	726	114	15.70%	0.9572
OpenRA	1157	273	23.60%	0.8362
pdfbox	1279	379	29.63%	0.6283
Pinta	400	112	28.00%	0.7421
ShareX	677	248	36.63%	0.6842

of files affected by hotspots with the number of code files in the project. Kazman et al. show that files containing hotspots are more bug-prone and exhibit maintenance problems, from which we infer that higher percentage of files affected by hotspots makes a codebase less maintainable [5]. The percentage of file affected by hotspots is then juxtaposed with the component independence metric (CI - percentage of source lines of code which are interface code or code which is called from outside of the component in which it resides in and also calls code outside of the component) measured by Better Code Hub (BCH) in Table II.

Discussion: We expected the percentage of files affected by hotspots to be negatively correlated with component independence (CI) (see table II). The correlation coefficient is -0.0162, indicating no correlation. Based on the above analysis, this indicates that the overall impact of hotspots on the codebase may not be measurable using the Better Code Hub's Component Independence metric.

C. Prototype

The research environment defined limitations on our inputs and tools that we could use, therefore we decided to implement a detector for Better Code Hub based on the state-of-the-art hotspot approach described in [5]. However, we used the source code graph created by the Software Analysis Toolkit as opposed to the Understand source code analyser.

Overview: The prototype consists of detector and visualisation parts. The visualisation is a part of the Edge front-end component and only consumes the hotspot data produced by the detector. The detector itself is a part of the GuidelineChecker component. In addition, the Scheduler is an orchestrating component which starts a Jenkins task and notifies the Edge component that the analysis is finished. The Jenkins component clones the source code repository, invokes the Software Analysis Toolkit which outputs a source code graph generated from the repository and the GuidelineChecker checks the source graph against the guidelines. Our detector is invoked as a part of the guideline check. Finally, the analysis result is stored in a MongoDB database, where it can be reached by the Edge component and presented by the visualisation part of the prototype.

Detector: The control flow of the detector is as follows: first, the class hierarchies are extracted from the source code graph as separate subgraphs; second, each hierarchy is checked for presence of two types of the Unhealthy Inheritance hotspot: internal and external. Internal Unhealthy Inheritance hotspot is a class hierarchy in which at least one base class depends on or refers to a derived class. External Unhealthy Inheritance hotspot is a class hierarchy which has client classes that refer to both based derived classes of the hierarchy at the same time. While detecting internal hotspots we investigate the classes and edges that belong to the hierarchy. For external hotspots we also check the neighbourhood of the class—clients of the class hierarchy, being classes which have a dependency on any of the classes in the analysed hierarchy.

IV. PROTOTYPE EVALUATION

The prototype evaluation involved integration with Better Code Hub and the gathering of feedback via structured interviews with developers who used the prototype. We intended evaluating the comprehensibility, relevance and actionability of the refactoring recommendations by asking scaled questions with a Likert scale [21]. Furthermore we asked developers what would they need to make the feedback more actionable and how would they address the problem. The integration of the hotspot detection into the existing system involved two steps: generating refactoring recommendations and visualisation.

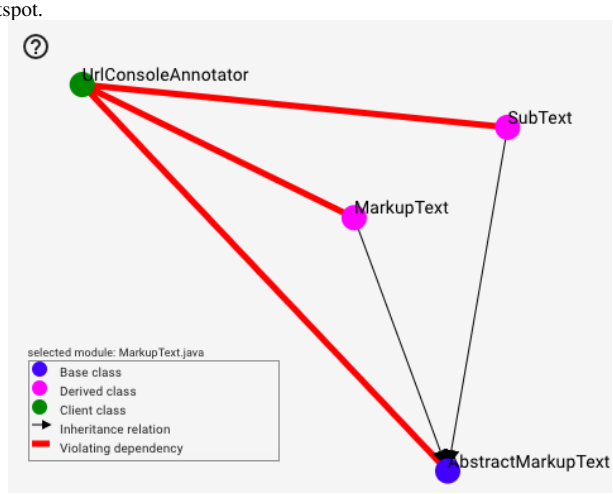
Refactoring recommendations are generated in two stages. First, the detector part identifies hotspots and generates a recommendation for every source node that is a part of a hotspot. Secondly, recommendations are filtered and ordered.

The visualisation for the user contains three additions to Better Code Hub: information about the number of hotspots in the refactoring candidate, a visualisation of the hotspot in the refactoring candidate and contextual information about a specific hotspot: what causes it, what its consequences are and suggested resolution strategies. We chose to visualize the hotspot as edges and vertices. It allows the user to manipulate the graph, by rearranging the nodes. Edges and vertices make it easier to convey more information visually such as type of dependency (inheritance, dependency, violating dependency) or type of source node (parent, child, client). Thus the user process is as follows:

1) As the user logs into the Better Code Hub, a list of repositories is revealed.
2) Once the user enters the repository analysis screen, a list of guidelines is shown with a tick or cross beside each indicating if the code in the repository is compliant.
3) As the user reviews the details of a specific guideline, a list of refactoring guidelines is provided for review.
4) In the hotspot visualisation screen the user can see the graph representing the hotspot visualised as a dynamic force simulation[3] which can then be manipulated.

[3]https://github.com/d3/d3-force

Figure 1. An example for the visualisation of an external Unhealthy Hierarchy hotspot.



5) Finally, we present the hotspot description which our prototype provides upon the user pressing the question mark button in the upper left corner of the visualisation.

In Figure 1 we present an example of an external Unhealthy Inheritance hotspot, a violation where a client class of the hierarchy refers to the base class and all of its children. In this case the client class is *UriConsoleAnnotator*, the base class is *AbstractMarkupText* and the child classes are *MarkupText* and *SubText*. The violations in this case are references from the *UriConsoleAnnotator* to all the classes in the hierarchy.

V. Discussion

For the evaluation we interviewed experienced developers. They had no prior experience with Better Code Hub and the codebase that they were assigned to evaluate the prototype on. Our aim was to devise recommendations which can be useful to a user who does not yet have intimate knowledge of the system architecture and implementation.

We only had time to evaluate the approach on a few systems. We made an attempt to choose systems representing different domains, architectures and languages; a broader test would be necessary to make sure that the conclusions do not stem from the selection bias. We hypothesise that the findings should be applicable to any strongly typed language that supports packages, modules and inheritance.

Even though before applying the method we found a strong correlation between hotspot density and the number of interface lines of code in a component, we did not find a causal link between removing hotspots and a decreased value of the component interface code as measured by the Software Analysis Toolkit. However, Mo et al. did show that the presence of hotspots indicates areas of code which are especially prone to introducing bugs, therefore, even if the removal of hotspots will not be reflected in the measurement, it would still improve the maintainability of the system [5].

VI. Conclusion

To improve the actionability of architecture level code quality violations we created a prototype tool that identifies structural problems in the codebase which impact modularity. We then provided refactoring recommendations based on the identified problems and interviewed developers to gather their feedback on comprehensibility, actionability and relevance of the presented recommendations. The prototype refactoring tool provides the following contributions:

- Detection of architecture smells in source code graphs.
- Refactoring recommendations to the user based on the presence of hotspots.
- Visualisation of hotspots, emphasising those dependencies negatively impacting modularity.
- Guidance for the users regarding the impact and structure of hotspots occurring in the analysed codebase.

As part of our analysis of repositories, we performed:

- A study of the reliability of hotspot detection on statically typed languages (Java, C# and C++).
- An analysis of the overall impact of code containing hotspots on the system's modularity.

A number of areas of future work have been identified:

a) More structural problems: We limited our detector to one kind of hotspot. We also chose to use our own detector as opposed to Titan, with a different source code analyzer, which means that there may be a mismatch between the results [20].

b) More detailed information about the violation: We only outline violating classes and dependencies. Using the same data the feedback can be improved by providing the exact calls along with the code snippets that trigger the violation.

c) Co-evolutionary coupling reasons: Co-evolutionary coupling is a term used to refer to classes which change together in time. It is much more difficult to address co-evolutionary hotspots. Firstly, co-evolutionary relationship data contains more noise. For example, a copyright header update will create a coupling between all the files in the project. Also, co-evolutionary relationships stay in history of the project. Secondly, it is more challenging to reason about the intention of the developer for changing files without a structural coupling together. Nevertheless, it would be interesting identify whether there are common reasons for co-evolutionary hotspot pattern occurrences.

d) Hotspot prioritization: We did not explicitly prioritise hotspots. However, it could be useful as the budget to address technical debt (e.g. architecture smells) is usually limited and decisions need to be made as to which issues should be addressed. A prioritisation could be used to suggest fixing those hotspots first, which exhibit a balance between effort needed to fix them and the impact on the maintainability.

Based on a preliminary evaluation we conducted through interviews with a panel of experts and the analysis of open source repositories we can say that users see the complimentary information as a promising starting point for further investigations, but will need additional work to make the recommendations actionable.

REFERENCES

[1] Robert L Glass. *Facts and fallacies of software engineering*. Addison-Wesley Professional, 2002.

[2] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[3] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. " O'Reilly Media, Inc.", 2016.

[4] Teodor Kurtev. Extending actionability in better code hub, suggesting move module refactorings. Master's thesis, University of Amsterdam, July 2017.

[5] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 51–60. IEEE, 2015.

[6] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[7] Daniel Guaman, PA Sarmiento, L Barba-Guamán, P Cabrera, and L Enciso. Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis. In *7th International Workshop on Computer Science and Engineering, WCSE*, pages 171–175, 2017.

[8] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[9] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[10] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[11] T. Kuipers, I. Heitlager, and J. Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)(QUATIC)*, volume 00, pages 30–39, 09 2007.

[12] Eric Bouwers, Arie van Deursen, and Joost Visser. Quantifying the encapsulation of implemented software architectures. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 211–220. IEEE, 2014.

[13] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[14] Tushar Sharma. Does your architecture smell?, 2017. Last accessed: 2018-06-03.

[15] Tushar Sharma. Designite: A customizable tool for smell mining in c# repositories. In *10th Seminar on Advanced Techniques and Tools for Software Evolution, Madrid, Spain*, 2017.

[16] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. Automatic detection of instability architectural smells. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 433–437. IEEE, 2016.

[17] Isela Macia, Roberta Arcoverde, Elder Cirilo, Alessandro Garcia, and Arndt von Staa. Supporting the identification of architecturally-relevant code anomalies. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 662–665. IEEE, 2012.

[18] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[19] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *International Conference on the Quality of Software Architectures*, pages 146–162. Springer, 2009.

[20] Lu Xiao, Yuanfang Cai, and Rick Kazman. Titan: A toolset that connects software architecture with quality analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–766. ACM, 2014.

[21] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.