Zones of Pain: Visualising the Relationship between Software Architecture and Defects

Jean Petrić, Tracy Hall, and David Bowes

Lancaster University, Lancaster, UK
{j.petric}{d.h.bowes}{tracy.hall}@lancaster.ac.uk

Abstract. Substantial development time is devoted to software maintenance and testing. As development resources are usually finite, there is a risk that some components receive insufficient effort for thorough testing. Architectural complexity (e.g. tight coupling) can make effective testing particularly challenging. Software components with high architectural complexity are more likely be defect–prone. The aim of this study is to investigate the relationship between established architectural attributes and defect–proneness. We used the architectural attributes: abstractness, instability and distance to measure the architectural complexity of software components. We investigated the ability of these attributes to discriminate between defective and non-defective components on four open source systems. We visualised defect–proneness by plotting architectural complexity and defectiveness using Martin's 'Zones of Pain'. Our results show that architecture has an inconsistent impact on defect–proneness. Some architecturally complex components seem immune to defects in some projects. In other projects architecturally complex components significantly suffer from defects. Where architectural complexity does increase defect–proneness the impact is strong. We recommend practitioners monitor the architectural complexity of their software components over time by visualising potential defect–proneness using Martin's Zones of Pain.

Keywords: software defects · software architecture · software evolution

1 Introduction

We aim to investigate the effect of architecture on defect–proneness. We build on previous work which looked at the relationship between some aspects of architecture and defects. Elish et al. compared the ability of three metric suites, which capture various static features of code, to predict pre– and post–defects [8]. Elish et al. demonstrated that Martin's suite of metrics [14] significantly outperformed the other two metric suites analysed. Jaafar et al. examined the impact of design patterns on defect–proneness and reported that components with anti–patterns are more defect–prone than others [12]. Jaafar et al. further demonstrated that components with anti–patterns are also those most involved in structural changes. However, it remains unclear if a complex architecture increases the likelihood of defect–proneness. To the best of our knowledge, no study has investigated the impact of architectural complexity on defect–proneness.

We used three metrics from Martin's [14] suite of metrics to measure the architectural complexity of software components. These are *abstractness*, *instability* and *distance from the main sequence* (short, *distance*). *Abstractness* (A) is defined as the ratio of interfaces and abstract classes in a component to the total number of classes in the component. *Instability* (I) is defined as the ratio between outward dependencies of a component and the total number of dependencies entering the component. Finally, *distance* (D) is defined as the absolute value of A and I which represent the distance from the main sequence in the '*tension plot*' (i.e. Figure 1). We also used Martin's notions of the "zone of pain" and "zone of uselessness", collectively called the "zone of exclusion", to categorise components by their architectural complexity. We investigated whether defect–proneness is more likely to occur in the zone of exclusion. We further investigated the likelihood of defect–proneness for components in the zone of exclusion compared to other components.

We set out to answer to research questions: **RQ1.** *What is the effect of architectural attributes on the defect-proneness of software components for the investigated open source systems?*; **RQ2.** *What is the proportion of defective components in the zone of exclusion for the investigated open source systems?* Our contributions are three–fold. Firstly, we show that architectural complexity is a promising indicator of defect–proneness. Architectural complexity may give complementary information with the addition of other metrics to defect prediction models. Secondly, we show that the relationship between architectural complexity and defect–proneness is not simple. Future studies are needed to understand which factors affect the relationship between architecture and defect–proneness. Thirdly, we provide all tools and data to the community for future analysis and replication.

The rest of this paper is structured as follows. The next section discusses the background to this work which is followed by a detailed methodology section. Section 4 then presents and discusses the results. Section 5 outlines related work, followed by the conclusions in Section 6.

2 Background

Many code design approaches to building reusable, maintainable and testable software have been proposed over the years. For example, Gamma et al. [9] documented over 20 reusable solutions for object–oriented systems, whilst Jaafar et al. conducted an empirical study to investigate the impact of design patterns on software maintenance and defectiveness [12]. Other work has focused on investigating problematic coding approaches that may hamper reusability, maintainability and testability. For example, Khomh et al. showed that classes containing anti–patterns are more frequently changed and more defect–prone than others in almost all releases of the four systems they analysed [13]. Hall et al. demonstrated that some code smells have a small but significantly negative effect on software defects [10]. Bavota et al. demonstrated that test smells impede the maintainability of software tests [6].

Many static code metrics have been used as a means to assess their impact on defect–proneness. For example, the CK suite [7], the MOOD suite [1], and Martin's suite [14] are amongst frequently used ones. Elish et al. showed that prediction models based on Martin's suite of metrics performed best amongst the three suites [8]. Almugrin et al. modified Martin's suite based on the concept of responsibility [3] and later showed that the modified suite yielded high correlation with respect to maintainability and testability [4]. In this study we focus on the architectural attributes of software. We use three architectural attributes defined by Martin [14] to explore their relationship with defect–proneness of software components.

3 Methodology

3.1 Architectural Metrics

Equations 1 depict *abstractness* (A), *instability* (I) and *distance from the main sequence* (D), respectively. In A, N_a is the number of abstract classes and interfaces in the component, whilst N_c is the number of concrete classes in the component. A is in the range of 0 and 1, where $A = 0$ indicates the component contains no abstract classes or interfaces. On the other hand, $A = 1$ indicates that the component contains nothing but abstract classes or interfaces. In I, Fan_{in} represents the number of inward, whilst Fan_{out} the number of outward dependencies. I value also spans from 0 to 1, where 0 indicates maximally stable component and 1 maximally unstable component. Finally, D calculates the euclidean distance from the main sequence. D also ranges between 0 and 1, where 0 indicates that the component is on the main sequence, whilst 1 indicates that the component is as far away from the main sequence as possible. When $D \approx 1$ the component is inside the zone of exclusion, either in the *ZoP* or *ZoU*. Figure 1 shows the relationship between the three metrics. We anticipate that components on and close to the *MS* should be less affected by defects compared to components in the *ZoE*. We use *tension plots* to visualise defective components across different snapshots of software evolution.

$$A = \frac{N_a}{N_c} \qquad I = \frac{Fan_{out}}{Fan_{out} + Fan_{in}} \qquad D = |A + I - 1| \tag{1}$$

3.2 Experiment

We used four open source projects shown in Table 1. All projects come from the *apache* community. We selected these projects because they use similar development standards which reduces issues that arise from analysing different open source projects. In addition, these projects generally belong to the same domain, i.e. Java libraries, are of reasonable size and widely used in the community. Table 1 summarises the chosen projects. The *# defects* and *# analysed*

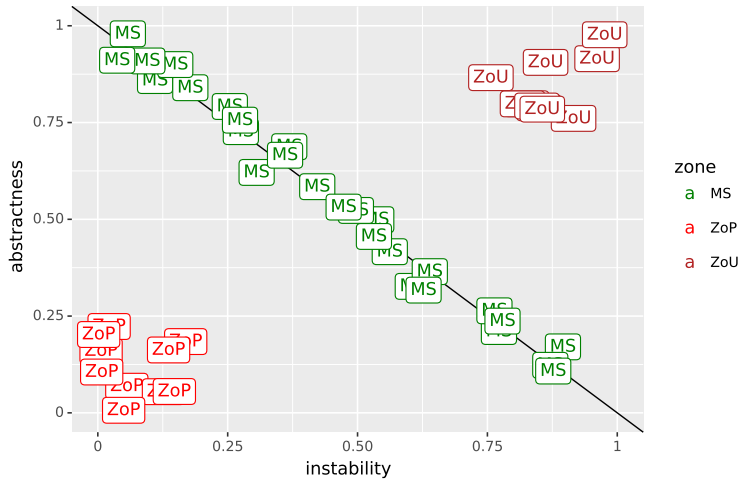

Fig. 1. The *tension plot* showing the relationship between A, I and D. Extreme values of A and I are driving components towards the zones of exclusion (also called, 'zones of pain').

commits columns are the total numbers of defective files and commits, respectively, throughout the project's history. The last two columns represent the average numbers of packages and classes for all analysed commits (these are the numbers per commit across the software history).

We collected two sets of data. The first set of data is a collection of defects for each project in Table 1. We used the SZZ algorithm to extract defective files for each commit throughout the project's history [15]. The second set of data contains the architectural metrics (A, I and D) for each of the four projects. Due to the lack of existing static metric tools that work on the latest Java versions, we developed *JavaMetrics*[1] to collect A, I and D metrics. For each project, we collected the metrics against all *git* commits. Finally, we amalgamated the information from the first and second set of data to get a list of metrics for all defective and non-defective components throughout each project's history. We also cleaned the datasets which scripts are available online[2].

To answer RQ1 we investigated whether complex components are likely to be more defective compared to their simpler counterparts. To reduce the bias we compared only components of similar size. We removed non–defective components which are 30% smaller or bigger than the defective components. Larger thresholds would include more components but would also defeat the purpose of comparing similar sizes. Smaller thresholds leave few components to compare. 30% threshold resulted in the right balance for further statistical analysis. To answer RQ2 we used an approach similar to *binary testedness* previously reported

[1] `https://github.com/lancsunise/JavaMetrics`
[2] `https://github.com/lancsunise/quatic20_replication`

Table 1. Open source projects used in this paper

Project	# defects	# analysed commits	# avg. package	# avg. class
hadoop-common	1617	10509	428	8362
camel	10501	44609	2081	14681
derby	5130	8269	230	2790
hive	11122	14377	621	12703

by Ahmed et al. [2] and Bach et al. [5]. *Binary testedness* separates source code in two (binary) groups. In its original form, one binary group is code covered with tests whilst another group is code with no associated tests. It is then possible, for a given snapshot, to count the occurrences of defective components for covered and uncovered code. If fewer defects end up in the covered compared to the uncovered group, we establish that testing is effective. Note that defects should initially be extracted via some form of defect prediction, rather than exposed by tests (i.e. tests would not be able to uncover any defects in uncovered code). We undertook a similar experiment to validate whether some architectural attributes lead to more defect–prone components. We defined Equation 2 to calculate the defect–proneness of components with $D \approx 0$ and $D \approx 1$. We used three thresholds, 0.2, 0.4 and 0.6 to calculate the ratios defined in Equation 2.

$$R_{D\approx 0} = \frac{N_{d0}}{N_{d0} + N_{nd0}}, R_{D\approx 1} = \frac{N_{d1}}{N_{d1} + N_{nd1}} \tag{2}$$

Each equation represents the ratio of defective components over the total number of components for a particular region of the *tension plot*. $R_{D\approx 0}$ are components similar to the green components, whilst $R_{D\approx 1}$ are components similar to the red components in Figure 1. N_d and N_{nd} are the counts of defective and non–defective components for the specific region of the *tension plot*, respectively.

4 Results and Analysis

Our RQ1 investigates whether architecturally complex components are likely to be more defect–prone compared to architecturally simpler components. If architecturally complex components were more defective on average, we would expect them to be farther away from the *MS*. To test the hypothesis whether defective components tend to have a greater *distance*, we used a one–sided non–parametric Mann–Whitney U test. We used the Mann–Whitney U test because of different numbers of instances between defective and non–defective components. For all projects except *derby* the $p-values$ were at least $5.596E-3$ or lower confirming that there is a statistical significance to conclude that architecturally complex components are more likely to be defective than their simpler counterparts.

Given that there is a significant difference between *distance* and defectiveness in most cases, we investigated the magnitude of this difference. To estimate the magnitude we used the ratios defined in Equation 2 for three different thresholds:

0.2, 0.4 and 0.6. These thresholds represent the maximum "shift" from the *MS* that divides the *tension plot* into two groups, as previously explained in Section 3.2. The expectation is to see $R_{D\approx1} > R_{D\approx0}$ for the thresholds approaching closer to 1. In other words, defectiveness of components increase as they are approaching closer to the *ZoE*. By calculating $\frac{R_{D\approx1}}{R_{D\approx0}}$ it is possible to estimate the magnitude (scale) of the difference between the two groups. Table 2 reports these details.

Table 2. Ratios of defective and non-defective components based on the distance from the Main Sequence

Project	shift	rdef	rnondef	scale	p–val
hadoop-common	0.2	0.129 (+/-0.129)	0.105 (+/-0.199)	1.223	0.266
hadoop-common	0.4	0.208 (+/-0.233)	0.140 (+/-0.206)	1.492	0.303
hadoop-common	0.6	0.405 (+/-0.45)	0.074 (+/-0.13)	5.449	0.102
derby	0.2	0.036 (+/-0.115)	0.103 (+/-0.119)	0.345	1.000
derby	0.4	0.022 (+/-0.078)	0.075 (+/-0.074)	0.288	1.000
derby	0.6	0.018 (+/-0.107)	0.054 (+/-0.035)	0.33	1.000
camel	0.2	0.060 (+/-0.099)	0.021 (+/-0.061)	2.838	0.000
camel	0.4	0.070 (+/-0.139)	0.028 (+/-0.078)	2.529	1.000
camel	0.6	0.119 (+/-0.315)	0.047 (+/-0.084)	2.54	1.000
hive	0.2	0.082 (+/-0.152)	0.083 (+/-0.157)	0.987	0.839
hive	0.4	0.033 (+/-0.129)	0.088 (+/-0.094)	0.373	1.000
hive	0.6	0.020 (+/-0.088)	0.030 (+/-0.02)	0.657	1.000

Table 2 presents the ratios and scale of the two groups of components for all four projects and the different thresholds. The first column is the project name, *shift* corresponds to the distance from the *MS*, *rdef* and *rnondef* are $R_{D\approx1}$ and $R_{D\approx0}$, respectively. The *scale* represents the magnitude $\frac{R_{D\approx1}}{R_{D\approx0}}$. $scale > 1$ means that architecturally complex components are indeed more likely to be defect–prone, whilst $scale < 0$ shows the opposite. In addition, $scale = 2$ shows that there are two times more defective components in the *ZoE* than around the *MS*. Finally, *p-val* shows whether the differences between $R_{D\approx1}$ and $R_{D\approx0}$ are significant. From Table 2, for *camel*, the scale is close to 3 for all the thresholds which indicates that an architecturally complex component is almost 3 times more likely to be defect–prone. On the other hand, *derby* shows very similar results with the scale close to 0.3 indicating that simpler components are 3 times more likely to be defect–prone. Figure 2 is an example of using the *tension plot* for a real–world project. The figure shows the arrangement of defective and non–defective components for 11 commits of *hadoop-common*. Each subplot in Figure 2 represents the state of defective and non–defective components for one *git* commit. Figure 2 clearly shows that for four commits, *0d5ed9*, *382ec9*, *46a7e0* and *f3a5d1* the most architecturally complex components are defective.

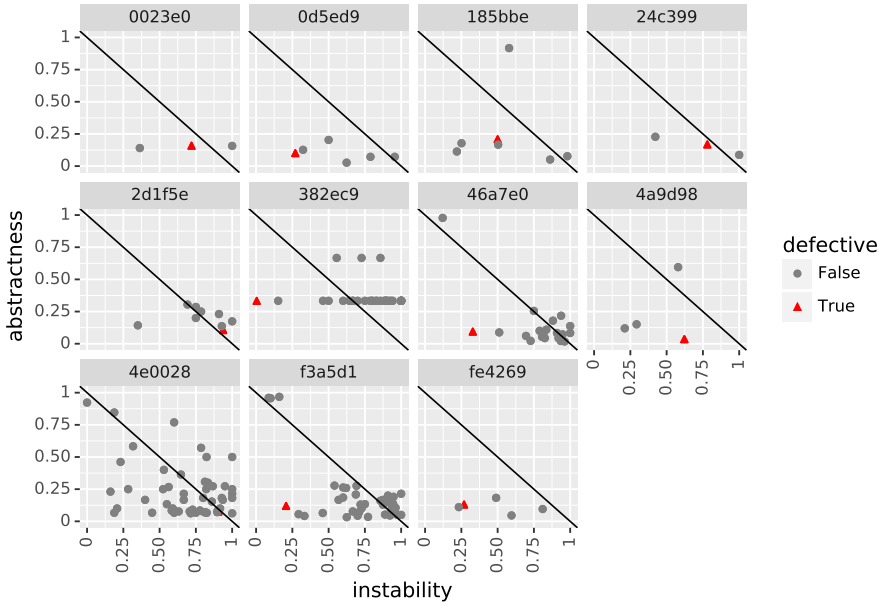

Fig. 2. Abstractness vs Instability for the top defective *hadoop-common* snapshots

5 Conclusions

Our findings suggest that architectural complexity of a component, as defined by Martin [14], does not always increase its likelihood to be defect–prone. There could be multiple reasons why this is the case. One reason is that complex components in some projects are more thoroughly tested compared to complex components in other projects. We suspect this to be unlikely in our analysis, given that we used the projects from the same community which follows the same protocol. Another, more likely reason, could be the difference in responsibilities of components in the *ZoE* compared to components close to the *MS*. As other studies have shown, practitioners spend more time maintaining and testing complex components (e.g. [3, 11]), which may leave more opportunity for defects to slip unnoticed in simpler components.

Overall, our analysis showed that for three out of the four considered systems architectural complexity has a strong relationship with defects. A strategic refactoring of components in the zones of exclusion by introducing abstraction is likely to reduce architectural complexity of components and decrease overall defect–proneness of the system. Visualisation techniques, such as the tension plot, as well as the Martin metrics can be an effective way for practitioners to determine which components require more attention. However, even though the magnitude of defect–proneness in the zones of exclusion can be three times higher, the effect is not consistent across all the systems. This suggests that there

are more factors that affect defect–proneness. For example, components in the *ZoE* may be disproportionally more tested compared to components close to the *MS*. Accounting for the level of testing could be a promising factor to explore in the future.

Acknowledgements

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/S005730/1

References

1. Abreu, F.B.: The mood metrics set. In: proc. ECOOP. vol. 95, p. 267 (1995)
2. Ahmed, I., Gopinath, R., Brindescu, C., Groce, A., Jensen, C.: Can testedness be effectively measured? In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 547–558 (2016)
3. Almugrin, S., Albattah, W., Alaql, O., Alzahrani, M., Melton, A.: Instability and abstractness metrics based on responsibility. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. pp. 364–373. IEEE (2014)
4. Almugrin, S., Albattah, W., Melton, A.: Using indirect coupling metrics to predict package maintainability and testability. Journal of systems and software **121**, 298–310 (2016)
5. Bach, T., Andrzejak, A., Pannemans, R., Lo, D.: The impact of coverage on bug density in a large industrial software project. In: Empirical Software Engineering and Measurement (ESEM). pp. 307–313. IEEE (2017)
6. Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D.: Are test smells really harmful? an empirical study. Empirical Software Engineering **20**(4), 1052–1094 (Aug 2015)
7. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
8. Elish, M.O., Al-Yafei, A.H., Al-Mulhem, M.: Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of eclipse. Advances in Engineering Software **42**(10), 852 – 859 (2011)
9. Gamma, E.: Design patterns: elements of reusable object-oriented software. Pearson Education India (1995)
10. Hall, T., Zhang, M., Bowes, D., Sun, Y.: Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology (TOSEM) **23**(4), 1–39 (2014)
11. Izurieta, C., Bieman, J.M.: Testing consequences of grime buildup in object oriented design patterns. In: 2008 1st International Conference on Software Testing, Verification, and Validation. pp. 171–179 (2008)
12. Jaafar, F., Guéhéneuc, Y.G., Hamel, S., Khomh, F., Zulkernine, M.: Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. Empirical Software Engineering **21**(3), 896–931 (2016)
13. Khomh, F., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering **17**(3), 243–275 (2012)
14. Martin, R.C.: Agile software development: principles, patterns, and practices. Prentice Hall (2003)
15. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories. pp. 1–5. MSR '05, ACM, New York, NY, USA (2005)