Managing technical debt: prioritising and quantifying architectural smells

Sas, Darius

DOI:
10.33612/diss.249298785

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2022

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 25-10-2022
Chapter 8
Conclusions and Future Work

This chapter summarises the contributions of this PhD project and concludes the dissertation. Section 8.1 revisits all research question and answers them according to the findings of the respective empirical studies, highlighting the respective contributions. Section 8.2 concludes the chapter by exploring future research directions that stem from the work presented in this dissertation.

8.1 Research Questions and Contributions

The problem statement addressed by this dissertation is stated in Chapter 1 and repeated here for convenience: *The detection of architectural smells alone is not sufficient for practitioners to take informed TD management decisions. Practitioners need to know the amount of TD each instance amounts to, what the available prioritisation strategies are, and the trend of the TD incurred over time. This information can help them better implement TD repayment.* To tackle this problem, we decomposed it into six research questions (five knowledge questions and one design problem) and answered them in Chapters 2 to 7. Each RQ addresses a particular aspect of the problem statement.

RQ1 addresses the lack of existing prioritisation strategies that can be used by practitioners, and provides guidance on what types of smells are more severe than others. Next, RQ2 allows us to better understand the needs of practitioners and the current state of practice with respect to architectural smells management in industry. Then, RQ3 collects information on the evolution of smells in a real world industrial context, which can be used to refine the prioritisation strategies reported in RQ1 for open-source systems. RQ4 addresses the relation of smells with changes in the source code in order to understand the advantages of repaying the ATD generated by architectural smells. RQ5 formulates a solution for the quantification and monitoring of the amount of TD principal by also using the results obtained from previous RQs. RQ6 addresses the knowledge gap of how practitioners perform TD repayment and how prioritising other quality attributes over maintainability affects TD.
8. Conclusions and Future Work

The upcoming paragraphs summarise the answer to each research question based on the empirical evidence collected.

**RQ1: How do AS evolve in open-source systems?**

To answer this research question, we carried out an empirical study that investigated how individual instances of architectural smells evolve over time and persist within the system. To do so, we detected the architectural smells in 524 releases of 14 different Java projects. We also developed an approach to track each instance of an architectural smell with its counterpart from the next version. This allowed us to create a time series data set about architectural smell instances.

The findings showed that different smell types evolve in different ways; for example, most cycles tend to stay constant regarding the number of elements affected (size) but tend to increase in total number of instances. The opposite holds for the Hublike Dependency (HL) smell, which tends to increase in size instead of in number of instances detected. Cycle instances were also found to be less persistent within the system, and after a few releases most of them might disappear as a consequence of the changes done in the system.

Given these findings, we deduced general prioritisation rules:

1. *HL smells should be addressed before cycles* because individual cycle instances are less likely to affect maintenance effort (i.e. TD interest) on the long term;

2. refactoring should not focus on *CD instances that were recently introduced* as many CD instances are likely to disappear in the next releases. This confirms state of the art findings that many CD instances are not critical [Al-Mutawa et al., 2014];

3. not all CD instances have the same severity, while some instances may be intentional. Refactoring activities should prioritise *complex shapes* that affect central parts of the system as they are more likely to incur extra maintenance effort.

The contributions compared to the state of the art are as follows: this is the first study to look at the evolution of *individual* architectural smell instances and at their survival probability within the system. Additionally, it is also the first to deduce general prioritisation rules among the smell types considered in the study.

---

1Note that cycles are replaced by different instances involving the new classes and packages added to the system.
RQ2: How are AS perceived by industrial practitioners?

This research question was investigated by an empirical study. In this case, however, instead of mining and analysing software repositories, we interviewed 21 software engineers and architects on their experience with AS. The data collected through the interviews gave us an insight on how practitioners perceive AS, how they introduce them as well as the maintenance and evolution issues related to smells.

The results showed that practitioners perceive the God Component (GC) smell as a common cause of maintenance issues, mainly as a result of the high level of complexity that characterizes its instances. Cyclic dependencies were perceived as less detrimental than GC, especially among practitioners working with Java systems. A similar opinion was given about the HL smell. Practitioners expressed their concern about change ripple effects associated with the UD smell, but they also reported that this smell type (which theoretically is responsible for an increase in change rate in the system) is rather hard to understand and not very intuitive.

HL and GC smells seemed to be the most commonly labelled as intentionally introduced, although practitioners also mentioned that they grew larger in size than what was intended. This resulted in practitioners being reluctant to refactor them. CD instances were instead attributed to bad initial design decisions that resulted in the introduction of cycles, which made them easier candidates for refactorings.

Another interesting finding is that practitioners perceived the presence of AS, in most cases, as a “necessary evil” in order to be able to meet deadlines and budget limitations. This shows that practitioners are aware and well-informed about good design practices, but they struggle following them diligently. It also confirmed that the problem statement is relevant to practitioners and that they require further support to manage AS.

The novelty of this study is that it is the first to report on the opinion of software practitioners on architectural smells using one-to-one, in-depth interviews as the main data collection method with the participants belonging to different companies. Previous studies used focus groups only and focused on a single company only.

RQ3: How do AS evolve in industrial embedded systems?

To answer this research question, we designed an empirical case study in an industrial setting. This study differs from the one performed to answer RQ1 because it includes both quantitative and qualitative data. Moreover, it focuses on C/C++ industrial embedded systems rather than Java systems. In the study we analysed over 20 millions lines of code, 30 releases, 9 projects, and interviewed 12 software
engineers and architects.

The findings derived from the quantitative analysis showed that most CD instances follow rather different growth patterns that those observed in open-source systems, with a large percentage of CD instances that grow in size over time rather than in number, thus becoming more severe. Cycles were found to be precursors of other smells, whereas HL smells were found to be the most common successor type of smell. Moreover, CD were found to be very common in GC smells, as more than 85% of CD instances co-occur with GC instances. GC instances instead seemed to be least co-occurring with HL smells, with only 10% of GCs co-occurring with HL instances.

The findings of the qualitative analysis highlighted that practitioners are aware of the presence of smells and can use intuition to pinpoint the problem but they need assistance in tracking and quantifying their presence deterministically. Moreover, practitioners also mentioned that change propagation to unknown parts of the codebase is one of their main struggles during typical maintenance activities. These issues were especially associated with the components affected by smells.

The novel contributions of this study compared to the state of the art are as follows: it is the first to present the evolution of individual AS instances in industrial systems. Additionally, this study also focuses on C/C++ programming languages, which is rare to see in the literature as most studies use Java instead. It is also the first study that provides a double perspective on the evolution of the smells in a single paper: the perspective from mining software repositories, and the opinion of the practitioners that worked on the code analysed.

RQ4: How do AS correlate to changes in the source code of the system?

To investigate this research question, we set up an empirical study to analyse 31 Java projects for a total of over 3,900 commits. The analysis compared the frequency and size of changes in affected and non-affected classes and packages. To do so, we used several different statistical tests while also controlling for the confounding factor of the size of the project, classes, and packages.

The results show that in 82% of the analysed commits the proportion of smelly artefacts that change is consistently higher than non-smelly artefacts that change. Medium and large artefacts that are affected by smells are naturally more likely to exhibit this difference in change frequency than small artefacts. Similarly, artefacts that exhibit changes are also more likely to have more smells than artefacts that have fewer smells, regardless of the size of the artefact. Artefacts with smells were also found to have larger change size (code churn) than non-smelly artefacts,
with the effect being more noticeable in larger artefacts. Finally, when a smell is introduced to an artefact, the change frequency of the artefact increases afterwards.

While these findings do not provide evidence that architectural smells are directly responsible for an increase in the frequency of change of the artefacts affected, they do confirm that the presence of architectural smells indicates the presence of hotspots in the design and code of the system.

Compared to the state of the art, this is not the first study that analyses the correlation between AS and source code changes. However, it is the first that provides a comprehensive analysis of the relation between changes and smells by looking not only at frequency of change, but also at the size of the changes and how the introduction of a smell instance affects the changes of a specific class/package. Moreover, it includes 31 systems in the analysis, making the findings more generalisable, as most other studies had analysed 20 systems at maximum.

RQ5: Design an approach to estimate the technical debt principal generated by AS

This research question represents a rather complicated problem to solve, that required solving first two design sub-problems: (1) how to calculate the severity of an architectural smell, and (2) how to quantify the effort to remove a smell.

To solve the first design problem, we designed a machine learning model that ranks architectural smells based on their severity. This required first to create a data set that comprised examples of smells instances with different severity. The data set was manually annotated using the findings from RQ1 to RQ4 (and the state of the art) as guidelines.

To solve the second design problem, we used the number of lines of code that are responsible for the presence of the smell, and therefore must be understood and changed by the developer. This solution ultimately resulted in a static analysis tool that precisely calculates these values.

Finally, we combined the two solutions to calculate the ATD index and designed a case study to validate our approach. In total, we interviewed 16 practitioners, 9 that worked on open-source projects (either full-time or as a hobby) and 7 from two companies. From the interviews, it emerged that practitioners agreed with the estimations provided in 71% of the cases and that these were representative of the effort necessary to refactor a smell. Moreover, they also mentioned that TD repayment was not always an option for them as it would force them to make a trade-off that they were not willing to.

The main contribution of this study compared to the state of the art, is that it
is the first study that uses machine learning to solve this problem. The advantage provided is that our approach does not require manually-set thresholds or a benchmark of systems to perform the estimations of TD principal. Additionally, our approach is the only one validated through an empirical study that involves practitioners from both industry and open-source systems.

**RQ6: How are trade-offs between quality attributes currently managed in industry?**

We answered this research question by conducting a case study, set up in an industrial setting. To collect the necessary data, we performed two rounds of interviews and a focus group. In particular, during the first round we interviewed 8 practitioners from 3 different companies operating in the embedded systems domain. Then we performed a focus group with 8 more practitioners from 4 companies, 3 of which were the same from the interviews. We concluded the data collection with the second round of interviews, where we performed 6 more interviews. In total, we consulted 21 software engineers and architects.

The results showed that practitioners prefer prioritising run-time qualities (e.g. performance, availability, etc.) over design-time qualities (e.g. maintainability, testability, etc.) citing, as the main reason, domain aspects (e.g. extreme performance optimisation) and business aspects (e.g. demos, deadlines), which take priority over code and architecture quality. Practitioners also mentioned how some of these trade-offs backfired eventually, hampering their ability to meet existing performance requirements or deliver new functionality on time.

The main novelty of this study compared to the state of the art is that it is the only one that reports examples of actual trade-offs among quality attributes from an industrial setting.

### 8.2 Future Work

The work presented in this thesis provides several opportunities for future work. We group these opportunities into three main research branches: (1) impact of architectural smells on maintainability; (2) impact of architectural smells on other quality attributes; and (3) refactoring support for architectural smells.
8.2 Future Work

8.2.1 Impact of architectural smells on maintainability

One opportunity for future work lies in continuing to study the impact of architectural smells on the maintainability of the system.

While this thesis has proposed an approach to calculate technical debt principal based on architectural smells, we still lack a way to estimate the amount of technical debt interest\(^2\) generated by architectural smells. Namely, to calculate how much “extra work” is done because of the presence of architectural smells. By using technical debt interest, one can calculate the “breaking point”, i.e. when the amount of interest that is being paid is larger than the cost to fix all the issues in the system. For instance, if in one year one pays more on TD interest than the cost of refactoring the system in the same amount of time, then it would be more convenient to pay back the debt. This would allow practitioners to plan ATD repayment activities more efficiently. Additionally, by using TD interest, practitioners can decide to focus refactoring activities only on the parts of the system where the most interest is paid.

Another example of future work is studying how changes propagate within and outside an architectural smell. In particular, it would be interesting to study what exactly happens when a component affected by a smell changes, and whether the change is propagated to the adjacent components that are not smelly, or to the other components that are part of the same smell. To study this aspect, however, it is necessary to detect co-changes\(^3\) very precisely. This type of study can help practitioners predict the files that need to be changed after applying a change to another file.

8.2.2 Architectural smells’ impact on other quality attributes

An interesting line of work is to study the impact of architectural smells on quality attributes other than maintainability.

Previous work has found that lower maintainability levels are associated with higher energy consumption [Papadopoulos et al., 2018]. An interesting future work would be to study whether the presence of architectural smells compromises energy efficiency. More specifically, it would be interesting to study if the refactoring of architectural smells would decrease the energy consumption of the system. The easiest way to study this would be to refactor one architectural smell

---

\(^2\) Also known as the cost of keeping the current solution as is.

\(^3\) Defined as when two files change together, either simultaneously in the same commit or concurrently in two different commits but with a precise pattern (e.g. file A always changes before file B).
instance, measure the impact on energy consumption, restore the system to the
original state, refactor a different smell instance, measure again energy consump-
tion, and then keep repeating the process. However, this methodology still poses
several challenges as one can fix the same smell in multiple ways, each impacting
energy consumption differently. Moreover there are several variables that influ-
ence the energy consumed by a machine, so one must remove as much noise as
possible.

Other trade-offs that could be investigated are whether the presence of archi-
tectural smells increases the chances of detecting security flaws in the affected
components. Intuitively, the presence of smells makes the code harder to compre-
hend, therefore, it is more likely that developers introduce a vulnerability flaw or
fail to detect one. A simple way of doing this would be to use the vulnerabilities
that are certified to be present, then use a historical analysis and see if, over time,
they tend to appear more in components that are affected by smells than in com-
ponents than are not (similarly to what done for source code changes in Chapter
5).

8.2.3 Refactoring support for architectural smells

An interesting opportunity for future work is studying how architectural smell
instances can be refactored automatically. This is a rather ambitious research
direction that may not even be completely feasible. However, even providing
developers with partial refactoring solutions that can be used as a starting point to
design a complete solution would be, by itself, of substantial help. The advantage
of such a solution is simple: if refactoring becomes easier for practitioners to apply,
then they will be more prone to adopt techniques to manage ATD. A first step
towards implementing this solution would be to try and solve the problem from
a structural point of view only. For instance, one can divide a complex smell into
smaller parts that can be individually solved through well-known solutions (e.g.
dependency inversion). Then, a solution for the refactoring of the whole smell can
be found by searching all the possible solutions that solve the individual, smaller
parts while also requiring the minimum amount of changes. Search-based software
engineering techniques have matured quite a lot over the past decades, and would
be a perfect fit to solve this problem.