Inferring Bad Smell Removal Recipes from Repository Mining

Luis Mayorga
Ansymo group, University of Antwerp

August 22, 2014

Abstract

There are refactoring recipes that define possible ways to remove bad smells. The purpose of this work is to find these recipes using an empirical approach, finding the refactorings performed in actual software projects. This is done by mining software repositories and analysing their history. This paper describes the process followed to analyse four open source projects and to collect the refactorings observed within them as well as the results obtained.

1 Introduction

As a software system evolves, it may grow in size and become more complex, which can have consequences over the quality of its design. The design flaws present on the software can have an impact over the quality attributes related to the maintainability of the system [1, 2].

In object-oriented programming, bad smells, also known as antipatterns, are a surface indication that usually corresponds to deeper problems in the system [3]. A great number of these antipatterns are characterized, which means that they can be detected attending to certain rules and metrics. Generally, for each software component affected by a bad smell it is possible to transform its code into an equivalent solution that removes the problem that the antipattern makes evident.

The technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour is known as refactoring. Thus, refactoring can be applied as a solution to bad smells, removing the problems caused by a bad software design. As well as there are several identified bad smells, there exist as well numerous well-known refactoring recipes to remove each one of them.

The purpose of this work is to analyse the strategies followed in actual software projects by checking the solutions that were applied. There exist refactoring guides that collect the solutions proposed by certain developers. The purpose of this work is to empirically gather evidence of how bad smells are removed in practice, obtaining if possible rules that could help us to decide which refactoring strategies could be applied applied to solve them. The approach is similar to the one draft idea presented in [4] but we do not go that far as to formalise our findings into refactoring strategies.

The procedure to conduct this study will involve mining some mature open source repositories. A revision interval belonging to each one of these repositories will be analysed and the bad smells present in each one of those revisions collected. This makes it possible to know at which point in time they disappeared or were intentionally removed. The last step would be finding the refactoring sequences associated to each one of those cases and studying them.

2 Background

2.1 Quality Attributes

The main quality attribute affected by a software bad design is maintainability. Although improving some attributes would require a change in how software behaves, it is not the case of maintainability, which can be improved just by modifying its structure. This quality attribute is addressed in several quality models and is a desired trait of software systems. Some of these are, for example the McCall and the ISO 9126 quality models. The first one, the McCall model, shown in Fig. 1, has three axes or viewpoints, one of them corresponding to revision [1]. That axis is composed of factors such as maintainability, testability and flexibility. The second one, the ISO 9126 possess a maintainability factor that has attributes such as analizability, changeability, stability, testability and maintainability compliance [2]. These quality models both describe how software should be, easy to maintain and change, easy to test and modular (stable). They turn maintainability into something that can be measured and sought, as well.

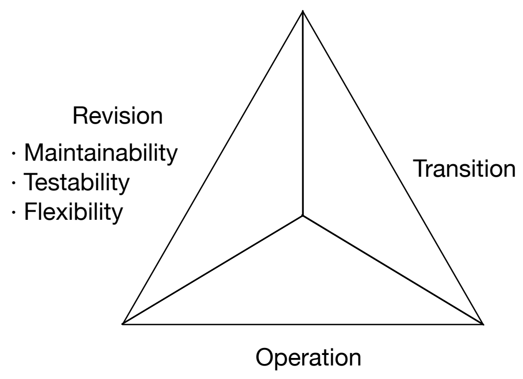

Figure 1: McCall model

2.2 Bad smells

A bad smell (also known as antipattern or disharmony [5, 6]) is a surface indication that could correspond to deeper problems in the system [3]. One of its traits is that it can be easily identified, as it “stinks” and stands out of the design.

However, a bad smell does not necessarily indicate a problem but suggests its possibility. Bad smells can be detected by means of metrics composition, what means that when certain values of these metrics are over a certain threshold, a rule is triggered and the presence of a certain antipattern is assumed. As a consequence, minor code changes can make a smell appear or disappear, even if this was not the developer’s intention, when detected through automated methods.

Some cases of bad smells that could be found are *Duplicated code*, *Long Method*, *God class* and *Data class*. If read by someone familiar with software design, most of these bad smell names will bring to his mind problems that can arise as development continues, most of them of the nature explained in section 2.1.

The way to proceed with these smells is removing them through a process in which an equivalent design with no smells is obtained. The reason for doing this would be getting a positive effect over the quality metrics that describe the attributes previously explained [7].

2.3 Refactoring

Fowler defines refactoring in his book [3] as:

A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

This refers to the process and transformations that take place in order to obtain a software design without bad smells. This is necessary to suppress the bad smells that hinder quality, as explained in sections 2.1 and 2.2. Some common refactorings are *Extract class*, *Move method* and *Replace constructor with factory method.*

3 Experimental setup

3.1 Objectives

The objective of this experiment is studying the refactorings performed by developers in software projects that had as a consequence the disappearance of a bad smell and find out whether that information can be used on other software developments. Such information would be the refactoring sequences that remove certain smells and the strategy for their application. This work can be taken as a feasibility study, in which this will be done for four projects, analysing the refactorings found and how effective is to generalise them as a solution.

Thus, the research question that will be answered in this experiment is the following:

Is it possible to mine refactoring recommendations from software version repositories?

3.2 Process overview

The process followed to answer the previous research question is the following and is performed on a per-project basis. First, the repository of the project is retrieved. Then, the chosen revisions are consecutively checked out, running a bad smell detection tool on them. After the results have been obtained for the desired range of revisions, the last appearances of bad smells on the reports are traced back to refactorings by using a refactoring detection tool. The concrete setup needed to carry the experiment depends on the tools selected, which are specified in the next section.

3.3 Tool selection

3.3.1 Bad smell detection

There are several tools available that make it possible to find flaws in software, such as JDeodorant[1], PMD[2] or inFusion[3]. However, some of them have some limitations.

It is possible to use JDeodorant in headless mode but the number of bad smells that it detects is reduced and the information cannot be exported for its later use. PMD is a tool that detects code defects, however, the scope of these flaws is small and does not take into account the relationships of high order entities (e.g. between classes). For this reason the information that it provides is not enough to detect bad smells. Finally, inFusion by intooitus can be programatically launched for a project, generating an xml report. This makes this last option the best for the approach taken in this work.

3.3.2 Refactoring detection

Refactorings can be detected with several tools [8] as well as by analysing the variations of code metrics [9, 10].

From all the refactoring detection tools available refFinder was considered the most suitable solution [11]. It can identify 63 refactoring types, which makes it a good choice for its comprehensive coverage. Although it is intended to be used interactively, a tool could be build to automate the necessary interaction and thus to use it programmatically. However, due to the limitations in time to carry the experiment it was done manually, thus ensuring that the results were correct. Knowing how accurate these results are makes it possible to automatize it in a future.

3.4 Software version repositories

For this work, four open source projects were selected. These projects needed to have their source code available in order to perform the required analysis, therefore open source was a requirement.

Mature projects were sought, as the information due to the high number of revisions would be greater on these. In order to analyse the explained concepts, it was necessary for them to be written in an OO language. The tools selection implied having to use only Java projects, as this is the only language that it is supported by all of them. Furthermore, there were some tools already developed by other researches at Ansymo that worked only for subversion (SVN) at that moment. Although finally these tools were not used, the project selection was intended to comply with this. Some of these projects are no longer in development or switched to git as version control software. In those cases only the information contained in SVN will be used.

The projects that were selected and their characteristics were:

Hibernate

- **Description:** Hibernate is an open source Java persistence framework project. It allows to perform object relational mapping and query databases using HQL and SQL.

[1]`http://www.jdeodorant.com/`

[2]`http://pmd.sourceforge.net/`

[3]`http://www.intooitus.com/products/infusion`

- **Overview:** It does not have tagged releases, which makes it difficult to analyse complete versions.
- **Repository:** `http://anonsvn.jboss.org/repos/hibernate/`

JEdit

- **Description:** jEdit is a mature programmer's text editor with years of development.
- **Overview:** The same case as Hibernate except for its last version.
- **Repository:** `http://sourceforge.net/p/jedit/svn/HEAD/tree/jEdit/`

JHotdraw

- **Description:** JHotDraw is a Java GUI framework for technical and structured graphics.
- **Overview:** A longtime project with all of its versions tagged, however, there was not a lot of activity in the last years.
- **Repository:** `http://sourceforge.net/p/jhotdraw/svn/HEAD/tree/`

PMD

- **Description:** PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth.
- **Overview:** The tags in the SVN repository of PMD were created late, all in consecutive revisions, which impedes using them as boundaries.
- **Repository:** `http://pmd.svn.sourceforge.net/`

Table 1 shows an overview in metrics of the aforementioned projects for the last revision of the subversion repositories.

Project	**KLOC**	**Classes**	**Methods**	**Statements**
Hibernate	135	1842	14198	53876
JEdit	109	1043	6653	50842
JHotdraw	140	1875	17158	62848
PMD	59	828	5694	29711

Table 1: Repository metrics overview.

These repositories often contain also side projects that complement the core functionality but are not part of it. These components can be created in any stage of the development so, in order to analyse the same single component during part of its lifetime without interferences caused by these complements, they will be overseen in the analysis. As a result of this, the revisions that only contribute to these projects will be empty for the core modules analysed. This makes necessary removing these revisions, as the analysis performed is very time-consuming and that could largely increase the time needed for it.

With that purpose we will use the utility `svndumpfilter` that ships with subversion in its version 1.8.8 with the flags `--drop-all-empty-revs` and `--renumber-revs`, creating consecutive valid revisions. Note that the revisions resulting from this process do not keep their original numbering. In order to retrieve the equivalence between both sequences, the output of the aforementioned command can be taken as a correspondence table.

The intervals finally analysed are shown in Table 2. In order to select these, we first aimed for complete versions using tags. As this was not possible for the characteristics of each repository, we attempted to select the latest complete year. This was only possible for JEdit and PMD.

In other cases, such as Hibernate, the number of revisions in that period of time was greater than 1000. Taking into account that for a project of such size, about 20 minutes are needed to analyse it, it would take two weeks to do so for the 1000 revisions. For that reason only the latest half of that year was selected for that project.

In JHotdraw, on the contrary, commits rarely happen and it is difficult to select a period of time with as much revisions as the other projects. Thus, the entire project was selected.

Repository	Start date	End date	Months	First revision	Last revision	Range size
Hibernate	2009-10-06	2010-04-09	6	901	1576	676
JEdit	2013-06-11	2014-06-11	12	7396	7541	146
JHotdraw	2000-10-12	2008-05-28	91	1	346	346
PMD	2011-07-24	2012-07-19	12	4735	5045	311

Table 2: Repository selected data.

Although the data selection is heterogeneous and could not correspond to the latest development of the project in the cases where it was moved to git, note that this does not affect the validity of the data, but the amount of it collected. More elaborated heuristics and more time to process the repositories would lead to larger and better input datasets.

3.5 Data extraction

In order to analyse the repositories, the process is staged in phases. An overview of the procedure can be seen in Fig. 2. The first one is obtaining the metrics from the projects. This is achieved by checking out each version of the repository to be analysed and running inFusion on them, generating xml reports. In this experiment, the revisions were not exhaustively checked, as the operation is very time-consuming as explained in the previous section. Instead, only one of each 5 revisions were analysed. There exists the possibility of a bad smell appearing and disappearing in the gap between checked revisions, however there is not a certain way to remove that problem. In some projects, such as JHotdraw, revisions are so scarce that such possibility exists even within a same revision. However, when having enough time, it is desirable to check each single revision of the repository, using all the information available in order to reduce this risk at its minimum.

After running inFusion, the reports are gathered in a serverless database (SQLite) that makes it easier to query the information. The database is queried in order to get the last appearance of a bad smell. If the value of this last revision is not equal to the latest revision in the range checked, that would mean that the bad smell disappeared at that point in time. Afterwards, the 5 revision range is checked to find the revision where the smell disappearance happened. After doing so, the entities whose bad smells disappeared are analysed for the two last by using refFinder.

A tool was developed to ease this procedure and to make possible extending it to a higher number of projects, making it easier in a future[4]. This tool has subcommands for running inFusion and loading the generated reports into the database as well as managing it. These are properly explained in the repository readme file.

The artifacts that are generated in the whole process are:

1. inFusion XML quality reports.
2. Database containing the information described in 1.
3. Entities and the revision in which they disappeared.
4. Refactorings found by refFinder on such entity.

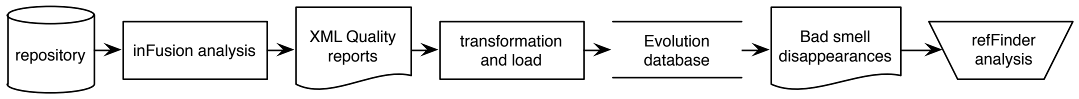

Figure 2: Process overview

4 Data analysis

4.1 Data overview

Figs. 3, 9, 7 and 5 show the distribution of bad smells in form of histograms for the analysed range of revisions of the projects and are followed by the removal histograms.

Fig. 3 shows how the count of bad smells grows as the development of Hibernate progresses. Its evolution over time is almost constant. This can be assumed to be a consequence of the overall size of the project increasing and not the amount of antipatterns recurrently growing on the same entities. There is not a notable decrease in the amount of bad smells at any time, either in the total count or associated to a concrete revision that could indicate the presence of a refactoring cycle. By looking at the bad smell removals in an isolated manner, as displayed in Fig. 4, the revision ranges that gather most of them can be appreciated more clearly. However, it was not possible to relate any of the major peaks with refactoring activity by looking at the repository activity log.

In JEdit, as seen in Fig. 5 and Fig. 6, happens the same as with PMD and no significant changes take place. There is a small drop in the bad smell count around revision 7456 labeled as "enhancements suggested by IntelliJIDEA".

In Fig. 7 the evolution of JHotdraw can be seen. There is a growing tendency with some sharp drops and spikes. If these changes are studied attending to the bad smell disappearance list we will see that most of these smells are duplication issues in its forms *SiblingDuplication*, *ExternalDuplication* and *InternalDuplication*[5]. Looking at the code reveals that these pronounced changes correspond to the copy and deletion of packages. Some JHotdraw versions are separated into their own folders, which doubles the flaw count and their code is taken as

[4]`http://github.com/rapsioux/nose`
[5]http://www.intooitus.com/products/incode/detected-flaws

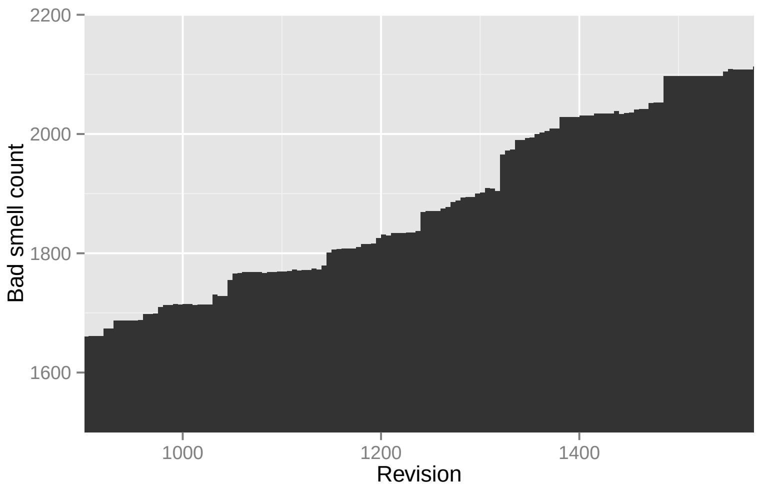

Figure 3: Hibernate bad smell count evolution.

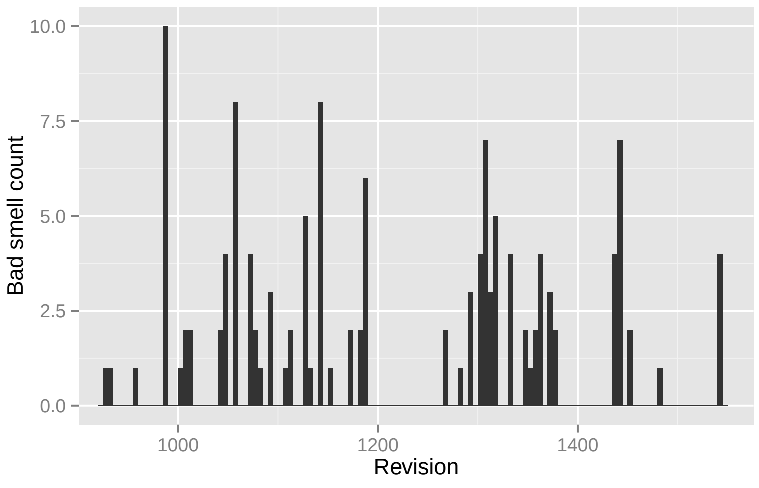

Figure 4: Hibernate removals histogram

duplicated, increasing even more the count. The strongly marked drops in the count are due to the deletion of packages and the big removal shown in Fig. 8 in revision 176 corresponds to a change in the package structure. Thus, the analysis of these revision ranges in order to detect clustered refactorings were in vain.

Fig. 9 shows an advanced stage in PMD development, in which almost no changes take place. The bad smell count stays stable except for small (note the scale) variations. Fig. 10 displays the bad smell removals. As can be seen, there was a phase in which 33 bad smells were removed on the measure taken affecting revisions 4881 to 4885. This was traced back to a change in the package structure in revision 4882 done in order to match the Maven structure. From these two figures we can observe as well how in some cases, after a removal takes place, the total amount of bad smells is reduced but also how in other cases it is not affected.

The analysis taken in this section gives an overview of how data is distributed in the samples

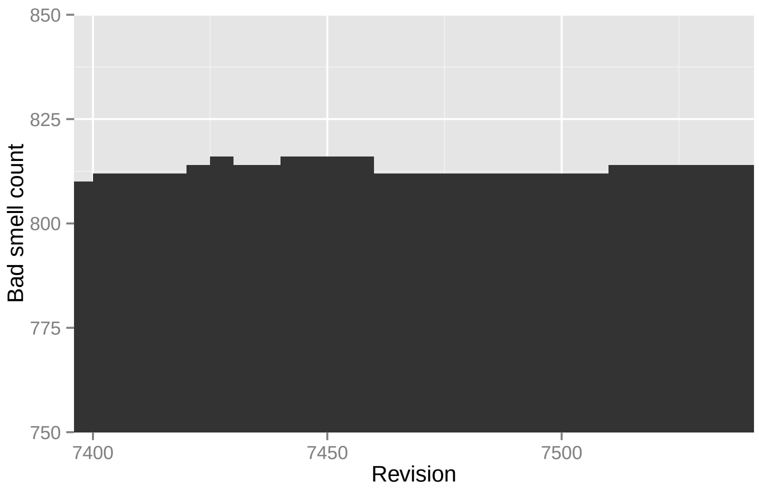

Figure 5: JEdit bad smell count evolution.

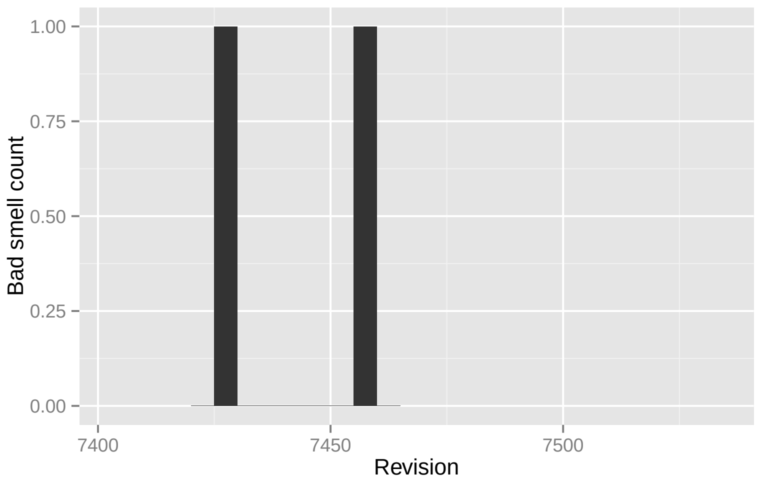

Figure 6: JEdit removals histogram

and the causes of the most notable changes. Although it would be interesting to have more exhaustive information regarding the causes of the increment or stability in the number of bad smells during the lifetime of a software project, it falls out of the scope of this work. For the continuation of this study, the data obtained could be studied along the evolution of size, complexity and bad smell density of these projects, studying how these metrics correlate.

4.2 Results

After all the bad smells are collected in the database, the bad smells which last revision is below the last revision analysed for their system are selected. That leads to the bad smells disappearances and the revision in which they happen. These disappearances should be reviewed using refFinder in order to determine whether the are intentionally removed by means of refactoring. The results obtained from refFinder are manually inspected in order to ensure that they are

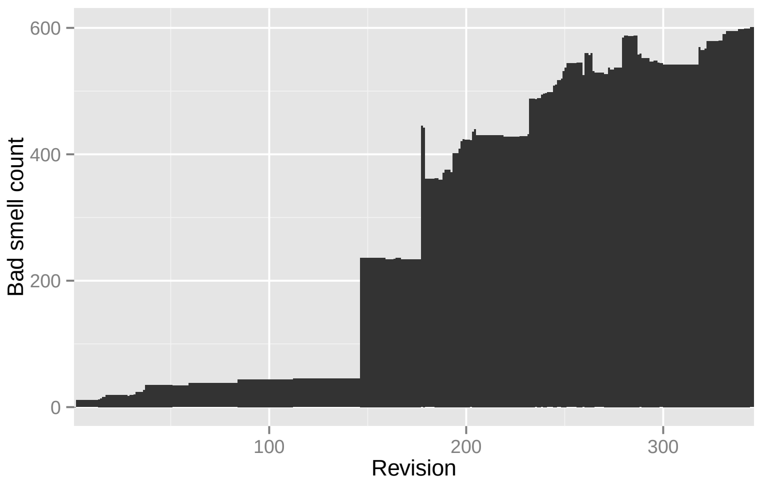

Figure 7: JHotdraw bad smell count evolution.

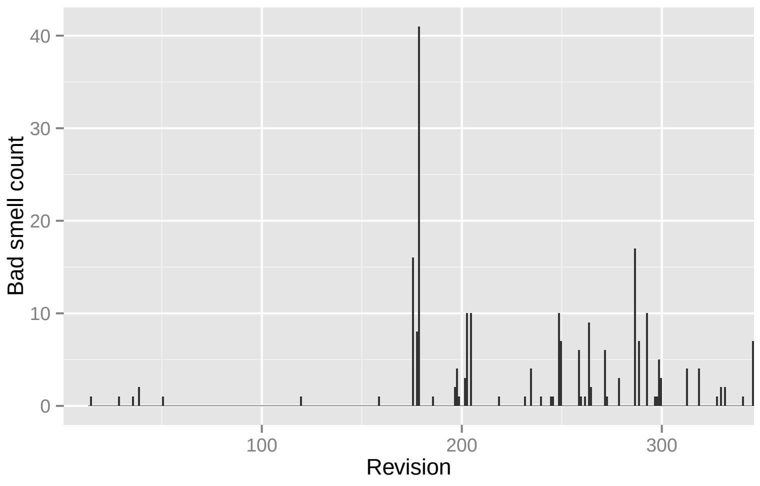

Figure 8: JHotdraw removals histogram

correct, discarding the ones that are not.

The datasets obtained for the study presented in this paper can be obtained from the section *Result processing* from the tool repository[6].

Table 3 shows the refactorings found for each one of the systems. As can be seen, the removal of bad smells that can be associated to a refactoring is very low, being around 3% for the whole dataset. Note that this number also includes trivial refactorings that may not provide useful information.

The refactorings that are not valid correspond to cases in which the bad smell does not disappear but it is not possible to trace it due to a change in its signature. This causes the bad smell to be taken as a different one. Some of these refactorings would be *rename method*, *rename class*, *remove parameter* and *add parameter*.

[6]https://github.com/rapsioux/nose/blob/master/README.md

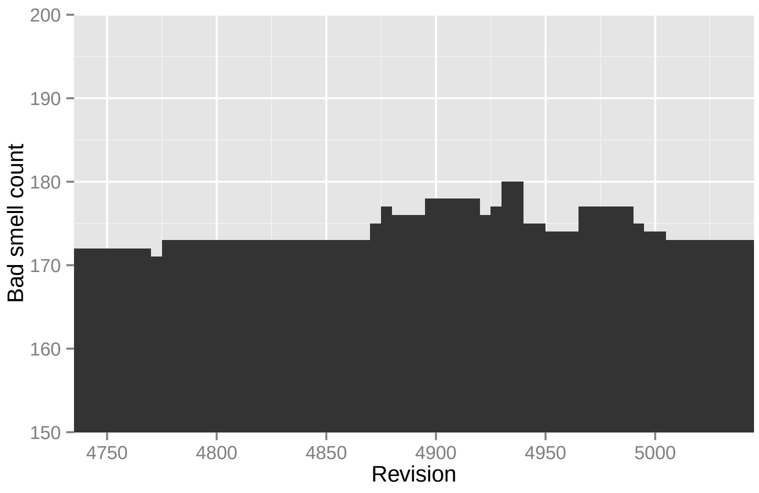

Figure 9: PMD bad smell count evolution.

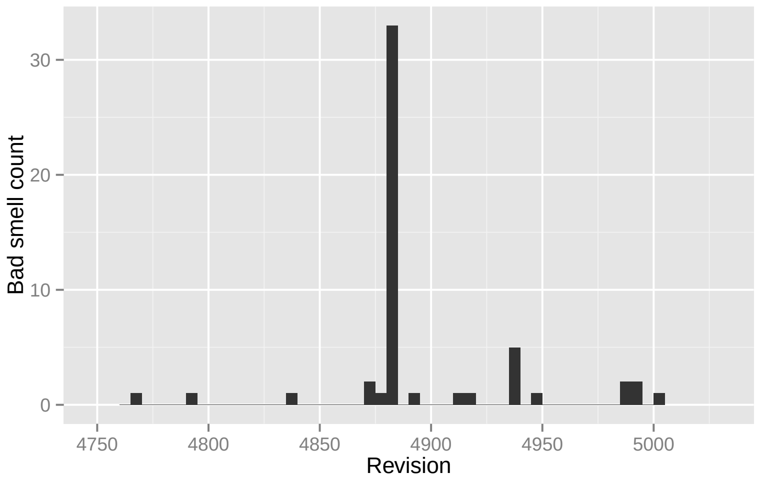

Figure 10: PMD removals histogram

4.2.1 Hibernate

All the refactorings found in Hibernate happen in classes affected by the *Data Class* smell. This bad smell is characterised by the exposure of a lot of information and having a very simple or inexistent logic. The refactorings applied for all of them are the following:

- Move field.
- Move method.
- Extract subclass.

The order of the refactorings must be disregarded as it is different for each one of the cases. By analysing the code we can observe that the three classes that had the smell, *Collection-Statistics*, *EntityStatistics*, *QueryStatistics* became interfaces, and there is a class for each one

Repository	**Smells removed**	**Smells removed with refactorings**	**Smells removed with valid ref.**	**Smells removed with valid ref. (%)**
Hibernate	131	5	3	2,29
JEdit	0	0	0	0,00
JHotdraw	224	4	3	1,34
PMD	53	8	6	11,32
Total	408	17	12	2,94

Table 3: Refactorings found.

of them that now has the implementation. This explains the refactorings found in the code although it does not provide any new solution beyond the one shown by Fowler [3].

4.2.2 PMD

Schizophrenic class

Refactorings found:

- Rename method.
- Extract method.
- Consolidate duplicate conditional fragments.
- Consolidate conditional expression.
- Replace exception with test.

This antipattern can be found in the class *JavaParser*, which is a very complex and probably is a generated class. A schizophrenic class is a class that has been detected to abstract two or more key abstractions. This means that the methods are used in a disjoint way by several groups of calling objects. The nature of the bad smell suggests that it is removed because there is a external change in the way in which the methods of this class are used. Attending at the refactorings found it is not possible to discern a big change in the interface made public by the class.

Blob operation

Two cases were found. The first one being:

- Inline temp.
- Extract method.
- Replace method with method object.

The second one was:

- Extract method.

As a blob operation smell indicates a method which has an over the average complexity, the solution is to modularise it into several methods that hold less responsibility. This is achieved by using *extract method* to abstract some code and, in the case of local variables being used, it can be substituted by a *replace method with method object* refactoring. This can be repeated until there is no need to keep abstracting code within the large method.

Feature envy

Three different refactoring sequences were observed for removing this antipattern. The sequence for the first one was:

- Inline method.

In the second ocurrence it was:

- Extract method.

And in the third one:

- Inline temp.
- Extract method.
- Replace method with method object.

In the cases affected by feature envy, we can observe different causes of the smell disappearance. One of them is *inline method*, which takes envied functionality to the class where it is being used and thus removing the method call. Although this could cause some small code duplication, in this case makes the number of envied features decrease and therefore the presence of the bad smell is disregarded. This can make it a good solution in cases in which the envied portions of code are small and the operation should not be exclusively performed by the other class.

Other cause is *extract method*, which extracts the functionality misplaced and takes it back to the envied class. This matches the solution most commonly suggested in the refactoring catalogs.

The last one can be considered as a composition of the previous ones. However, in this case a new class is created to consolidate the operations and the whole method is substituted by the call made.

Intensive coupling

Refactorings found:

- Extract method.

The heuristics for detecting intensive coupling from one method take into account the number of calls made within it. The disappearance of this bad smell is caused by an *extract method* operation, which causes these calls to be scattered along several points, keeping the coupling of the class at the same level but decreasing the one relative to that particular method and improving the responsibility distribution inside the class.

4.2.3 JEdit

No results were found for JEdit as none of the disappearances of bad smells could be related to a refactoring.

4.2.4 JHotdraw

Blob operation

1. Extract method.
2. Inline temp.

As seen with PMD (section 4.2.2), the way to remove a big operation is to decompose it into smaller ones. This is achieved by the use of the *extract method* refactoring and when the previously conditions are also met, making variables or methods inline.

5 Threats to validity

We have tried to eliminate or minimise the bias and confounding factors when designing and performing this study, although some threats to validity undoubtely remained. One of threats to validity would be the reliability of the collected data. In such aspect, the data collected even for the reduced range of revisions that was analysed is not comprehensive. This means that, as bad smells are detected in some of the repositories in a non consecutive manner, the odds are that some smells could have been removed after one measure and appeared again before the consecutive one was taken. In that case some bad smells removals could have been masked and the solution applied would have been missed. However, such possibility is small due to the size of the increments taken. As mentioned, this affects to the amount of data available but not to degree at which the data can be trusted.

Another threat would be at which extent the results obtained can be generalised for other projects. There are already tested recipes that remove these bad smells and the solutions applied in these cases in several cases partially match them. However, it cannot be stated whether this would be the case for a wider range of projects or if they would give any results at all.

6 Related work

Up to our knowledge we only know of one study similar to ours by Ouni et al. [12], which tries to solve the same problem by using a different approach. Our aim is providing the developer with a catalog of refactorings able to solve a specific problem. However, the objective of the work mentioned is using the history of a project as one of the multiple factors affecting which refactorings will be automatically chosen and applied.

7 Conclusion

This work takes the idea of Pérez et al. for getting refactoring recommendations and studies its feasibility, contributing with the following findings. Although a greater study would be needed in order to understand the reasons behind the measures obtained, we can have an overview of bad smells presence as four software projects evolve. It can be seen how some bad smells may disappear without being involved in an active maintenance process, not following the expected behaviour of continuous growth. With this study it is proven that it is possible to mine repositories in order to get refactoring recommendations, thus positively answering the research question raised. Some of the refactoring solutions observed matched and extended well known recipes, as in the *Blob operation* and *Feature envy* smells. The analysis of a higher volume of data and the study of the environment where these recipes were found could lead to the aforementioned strategies. The tool developed to perform the reported analysis, as well as the data, can be used in a future with this purpose.

In terms of future work, there are some improvements that could be made to the procedure. One of them would be making inFusion able to analyse only increments in the code, which would dramatically decrease the time needed to run the bad smell analysis. Another one would be atomating refFinder, which is provided as an interactive tool and would allow to get the refactorings applied. Finally, the information related to external entities could be also stored. External can be understood as an entity different to the one where the bad smell disappeared. That would help to to explain their disappearance smells caused by changes in other classes or packages that cannot be explained by any refactoring in the original entity.

8 Acknowledgements

I would like to thank Javier for his support during the internship and his great work as tutor as well as the rest of the Ansymo group, of the University of Antwerp for their direction during its accomplishment. Special acknowledgement to Intooitus for their collaboration and for providing us with an inFusion license, making this study possible.

9 References

[1] Jim A McCall, Paul K Richards, and Gene F Walters. *Factors in software quality.* General Electric, National Technical Information Service., 1977.

[2] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[4] Javier Pérez, Alessandro Murgia, and Serge Demeyer. A proposal for fixing design smells using software refactoring history. 2013.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 2006.

[7] Bart Du Bois and Tom Mens. Describing the impact of refactoring on internal program quality, 2003.

[8] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[9] Q.D. Soetens and S. Demeyer. Studying the effect of refactorings: A complexity metrics perspective. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 313–318, Sept 2010.

[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. *SIGPLAN Not.*, 35(10):166–177, October 2000.

[11] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 371–372, New York, NY, USA, 2010. ACM.

[12] A Ouni, M. Kessentini, and H. Sahraoui. Search-based refactoring using recorded code changes. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 221–230, March 2013.