Willian Nalepa Oizumi

Identification and Refactoring of Design Problems in Software Systems

Tese de Doutorado

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
April 2022

Willian Nalepa Oizumi

Identification and Refactoring of Design Problems in Software Systems

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the Examination Committee.

Rio de Janeiro, April 28th, 2022

Willian Nalepa Oizumi

The author obtained a master's degree in Informatics from PUC-Rio (2015). He also holds a bachelor's degree in Informatics from UEM (2013). His main research interests are: Software Design, Design Problems, Refactoring, and Quality Attributes.

Bibliographic data

Oizumi, Willian Nalepa

Identification and Refactoring of Design Problems in Software Systems / Willian Nalepa Oizumi; advisor: Alessandro Fabricio Garcia. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2022.

v., 235 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Software Design – Teses. 2. Design Problems – Teses. 3. Search-based Software Engineering – Teses. 4. Recommendation Systems – Teses. 5. Refactoring – Teses. 6. Projeto de Software;. 7. Problemas de Projeto;. 8. Engenharia de Software Baseada em Busca;. 9. Sistemas de Recomendação;. 10. Refatoração.. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to thank my advisor, Alessandro Garcia, for giving me the opportunity to work with him. His patience, energy and good will are incredible. I also thank professors Thelma Colanzi, Marcos Kalinowski, Bruno Cafeo, Juliana Pereira, Baldoino Fonseca, and Alberto Raposo, for dedicating their precious time to the evaluation of this PhD thesis. The feedback provided by them was very valuable to me and to this thesis.

I would like to thank my beloved wife, Julia, who supported me during the conduction of this work. There is no words to describe her love, patience and kindness. My deepest gratitude goes to my whole family, who always supported me. A special thank goes to my aunt, Leticia, who was responsible for raising and educating me. Without her I would not have come this far. I also thank my friends Roberto, Givanilde, Carlos, Frank, and Hélio and my former students Flávio, Navarro, Lucas, Angélica, and Henrique.

A special thank also goes to Leonardo Sousa, Anderson Oliveira, Santiago Vidal, Anderson Uchôa, Rafael Mello, Caio Barbosa, and Diego Cedrim for the great collaborations during this PhD research.

I thank all the professors from PUC-Rio for their contribution to my education. My appreciation also goes to my professors from UEM. I also thank my friends and colleagues from the OPUS Research Group and from the Software Engineering Laboratory.

Finally, I thank CNPq and PUC-Rio for financial aid, without which this work would not have been possible.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

Abstract

Oizumi, Willian Nalepa; Garcia, Alessandro Fabricio (Advisor). **Identification and Refactoring of Design Problems in Software Systems**. Rio de Janeiro, 2022. 235p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software projects impacted by Design Problems (DPs) may become difficult to maintain and evolve. The identification of DPs may occur through symptoms such as code smells. After such identification, developers can remove the DPs through refactorings. However, deciding where and how to refactor is a challenging task. Thus, several refactoring recommendation techniques have been proposed. Nevertheless, there is still little consensus on which requirements must be satisfied by them. In this thesis, we are proposing four empirically identified requirements that any DP removal technique should follow. First, each single DP is usually related with multiple types of symptoms in the source code and they should be considered altogether for generating recommendations. Second, a recommendation technique should allow the selection of possible candidate contexts for refactoring. Fourth, the technique should consider the features of undergoing changes to create useful recommendations. Finally, developers do not always conduct the most effective refactorings in practice, quite often unconsciously, resulting in the incomplete removal of DPs. Thus, they need assistance to remove DPs. There are techniques partially fulfilling the aforementioned requirements, though none satisfactorily meets them all. Thus, we propose the *OrganicRef* technique. *OrganicRef* is intended to help developers in removing DPs in their contexts of interest. *OrganicRef* finds the contexts by capturing the features affecting relevant code elements using a topic modeling algorithm. Then, it collects multiple symptom types affecting the code elements. To recommend effective refactorings, *OrganicRef* combines rule-based and feature-driven heuristics. It also uses search-based optimization to derive multiple possible recommendations. To evaluate *OrganicRef*, we conducted an empirical study with six open source projects. Results showed that *OrganicRef* recommendations significantly improves the design of refactored elements.

Keywords

Software Design; Design Problems; Search-based Software Engineering; Recommendation Systems; Refactoring.

Resumo

Oizumi, Willian Nalepa; Garcia, Alessandro Fabricio. **Identificação e Refatoração de Problemas de Projeto em Sistemas de Software**. Rio de Janeiro, 2022. 235p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Sistemas impactados por Problemas de Projeto (PPs) podem se tornar difíceis de manter e evoluir. A identificação de PPs pode ocorrer por meio de múltiplos sintomas, tais como code smells. Após tal identificação, pode-se remover os PPs por meio de refatorações. No entanto, decidir onde e como refatorar é uma tarefa desafiadora. Assim, técnicas de recomendação de refatoração têm sido propostas. Apesar disso, ainda há pouco consenso sobre quais requisitos devem ser atendidos por elas. Nesta tese, estamos propondo quatro requisitos empiricamente identificados que tais técnicas devem seguir. Primeiro, cada PP geralmente está relacionado a vários tipos de sintomas no código-fonte e eles devem ser considerados juntos para gerar recomendações. Além disso, uma técnica de recomendação deve permitir a seleção de contextos específicos para refatoração. Quarto, também deve-se considerar as funcionalidades modificadas para criar recomendações úteis. Finalmente, os desenvolvedores nem sempre conduzem as refatorações mais eficazes na prática, muitas vezes inconscientemente, resultando na remoção incompleta de PPs. Assim, eles precisam de assistência para remover os PPs. Existem apenas técnicas que atendem parcialmente aos requisitos mencionados. Sendo assim, nós propomos a técnica *OrganicRef*. *OrganicRef* destina-se a ajudar os desenvolvedores na remoção de PPs em seus contextos de interesse. *OrganicRef* encontra as funcionalidades dos elementos de código usando um algoritmo de modelagem de tópicos. Em seguida, ele coleta múltiplos tipos de sintomas que afetam os elementos do código. Para recomendar refatorações, *OrganicRef* combina heurísticas baseadas em regras e baseadas em funcionalidades. *OrganicRef* também aplica otimização baseada em busca para derivar várias recomendações possíveis. Para avaliar o *OrganicRef*, realizamos um estudo experimental com seis projetos de software. Os resultados mostraram que as recomendações do *OrganicRef* melhoram significativamente a qualidade dos elementos refatorados.

Palavras-chave

Projeto de Software; Problemas de Projeto; Engenharia de Software Baseada em Busca; Sistemas de Recomendação; Refatoração.

Table of contents

List of figures

List of tables

List of Abreviations

CEM – Communicability Evaluation Method
DP – Design Problem
FP – False Positive
GA – Genetic Algorithm
HCI – Human-Computer Interaction
HW – Health Watcher
ICPC - International Conference on Program Comprehension
ICSME – International Conference on Software Maintenance and Evolution
IDE – Integrated Development Environment
ISSRE – International Symposium on Software Reliability Engineering
JBCS – Journal of Brazilian Computer Society
KLOC – Thousands of Lines of Code
LENOM – Latest Evolution of Number of Methods
MM – Mobile Media
MOEA – Multi-Objective Evolutionary Algorithm
MOSA – Multi-Objective Simulated Annealing
MSR – Mining Software Repositories
NSGA – Non-dominated Sorting Genetic Algorithm
OO – Object-oriented
OODT – Object Oriented Data Technology
PR – Pull Request
QMOOD – Quality Model for Object Oriented Design
RAM – Random Access Memory
RQ – Research Question
SA – Simulated Annealing
SBES – Brazilian Symposium on Software Engineering
SBSE – Search-based Software Engineering
SDB – Subscribers Database
SRP – Single Responsibility Principle
TP – True Positive
UML – Unified Modeling Language

1 Introduction

Design Problems (DPs) occur when stakeholders make decisions that negatively impact quality attributes such as modifiability, modularity, and the like (Li, Avgeriou and Liang 2015), (Lim, Taksande and Seaman 2012). Software systems may be discontinued or redesigned when DPs are allowed to persist (MacCormack, Rusnak and Baldwin 2006). In addition, the introduction of DPs is linked to: (1) the rejection of contributions in open source projects (Oliveira, Valente and Terra 2016), and (2) increased costs in industrial software projects (Curtis, Sappid and Szynkarski 2012). Therefore, DPs should be properly handled by software developers.

The identification of DP usually occurs through symptoms such as abnormal code measures and code smells (Sousa *et al.* 2018). Refactoring (Fowler 1999) is a practice adopted by many developers to remove DPs. Nevertheless, deciding where and how to refactor is far from trivial. Software projects often suffer massive changes, preventing their developers from keeping track of the source code locations impacted by DPs. Moreover, there is evidence that even when the locations of DPs are known, refactorings performed in practice may be unable to completely remove them (Oizumi *et al.* 2019), (Bibiano *et al.* 2020). In fact, developers consider that better techniques are necessary for tasks such as identification and removal of DPs (Rebai *et al.* 2020), (Lim, Taksande and Seaman 2012), (Ernst *et al.* 2015).

Given such a need, there are multiple techniques for assisting developers to identify and remove DPs through refactoring recommendations (Rebai *et al.* 2020), (Alizadeh *et al.* 2019), (Ouni *et al.* 2017), (Lin *et al.* 2016), (Xiao *et al.* 2016). There are also guidelines for building refactoring recommendation techniques and tools (Tsantalis, Chaikalis and Chatzigeorgiou 2018), (Bavota *et al.* 2014). However, recent studies have shown the importance and necessity of requirements that are still not widely met by existing techniques (Peruma *et al.* 2022), (Lacerda *et al.* 2020). In addition, many of the existing techniques are still poorly known and adopted in practice (Lacerda *et al.* 2020), (Pinto and Kamei 2013).

In this thesis, we have identified, at least, four minimum requirements that a recommendation technique should address, namely: (1) consideration of heterogeneous information, (2) context-sensitive detection, (3) feature modularity awareness, and (4) effective recommendations. Next, we briefly describe each of them.

Consideration of Heterogeneous Information. *Refactoring recommendations should be generated and provided based on information extracted from multiple and diverse sources.* Before refactoring, developers need to consider and understand various types of information, including the relations between symptoms, the corresponding DPs, and their negative consequences (Sousa *et al.* 2018, Sousa *et al.* 2017). Without understanding these information and all these relations, the developer may not be confident enough to conduct refactorings. Besides that, each DP is usually related with multiple symptom types, which should be considered for effectively detecting DPs and generating refactoring recommendations (Oizumi *et al.* 2020, Oizumi *et al.* 2019, Oizumi *et al.* 2016).

Context-Sensitive Detection. *Recommendation techniques should provide mechanisms to select and focus on a specific context.* To provide refactoring recommendations, a technique need to rely on the detection of DP symptoms. However, detecting DP symptoms and generating recommendations for the whole project is not an effective strategy (Oizumi *et al.* 2019, Alizadeh and Kessentini 2018, Rebai *et al.* 2020, Vidal *et al.* 2019). In fact, developers usually avoid changing code elements that are out of their context of interest (Alizadeh and Kessentini 2018, Alizadeh *et al.* 2019c). Therefore, an effective technique should provide a flexible mechanism for helping developers to focus on their contexts of interest. Examples of context include (1) the components developed and maintained by the developer, (2) the code elements being changed in a task, and (3) code elements that will be changed in a future task.

Feature Modularity Awareness. *Besides using rule-based heuristics, recommendation techniques should consider feature modularity for creating refactorings.* Certain recent studies have been using information about features for creating refactoring recommendations (Bavota *et al.* 2013), (Nyamawe *et al.* 2019), (Rebai *et al.* 2020). Such a tendency is justified by the fact that developers often focus on refactoring code elements that realize features affected by ongoing change tasks (Alizadeh and Kessentini 2018). Therefore, besides restricting the recommendations to a specific context, a technique should employ feature-driven refactoring heuristics that maximize the chances of modularizing the features involved in an ongoing change.

Effective Recommendations. *Developers need assistance for effectively removing DPs through refactorings.* The removal of a DP usually involves the execution of a sequence of multiple refactorings (Oizumi *et al.* 2020, Sousa *et al.* 2020a, Cedrim 2018). In this work, we call such refactoring sequences as *composite refactorings.* Despite composites being widely investigated in the literature (e.g., (Brito, Hora and Valente 2020), (Bibiano *et al.* 2020), (Bibiano *et al.* 2019), and (Sousa *et al.* 2020a)), there is evidence that many refactorings performed in practice are not effective (Cedrim 2018, Rebai *et al.* 2020). As a result, DPs end up not being completely removed. Moreover, some refactorings may even worsen the design quality (Sousa *et al.* 2020a, Cedrim 2018). Some of the reasons for the lack of effectiveness are that developers usually (1) perform incomplete composite refactorings (Bibiano *et al.* 2019, Bibiano *et al.* 2020) and (2) select combinations of refactorings that are not the most effective for their context (Rebai *et al.* 2020, Alizadeh and Kessentini 2018).

Given the aforementioned requirements, there is a growing need for new techniques to better support the identification and removal of DPs. In the following section, we present a hypothetical example that illustrates the importance of addressing the aforementioned requirements to provide effective support for the removal of DPs.

1.1 Motivating Example

Figure 1.1 uses a UML-like notation to show a partial view of a hypothetical system called *UniM.* The main objective of UniM is to provide support for the management of academic activities in universities. UniM allows its users to perform operations such as course management and student enrollment.

UniM developers perceived that including new features in the *Service* component were often time consuming and error prone. A traditional DP detection tool reported more than 1000 symptoms occurring in the UniM system. However, many of the symptoms were not related to the *Service* component. Only after analyzing dozen symptoms occurring in the *Service* component, developers managed to find where refactorings should be performed to improve the modifiability and modularity of *Service.*

Based on occurrences of the *Insufficient Modularization* and *Broken Modularization* symptoms, they discovered that some course management services were scattered in classes like the *InstitutionalEnrollmentService.* After that, the developers had to decided by themselves which composite refactoring should be

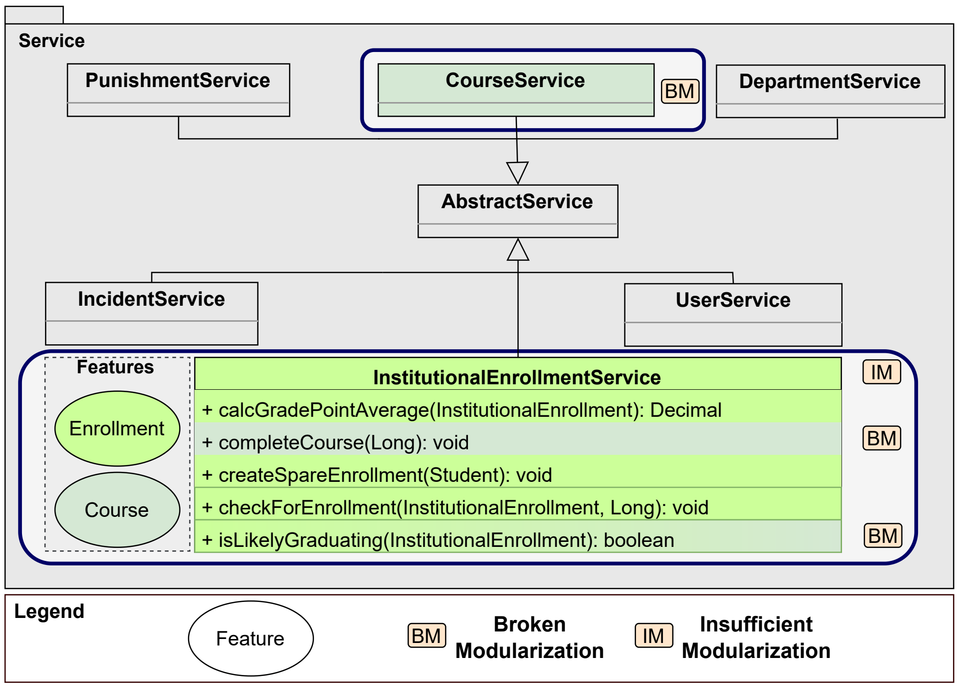

Service
PunishmentService
CourseService
BM
DepartmentService
AbstractService
IncidentService
UserService
Features
Enrollment
Course
InstitutionalEnrollmentService
+ calcGradePointAverage(InstitutionalEnrollment): Decimal
+ completeCourse(Long): void
+ createSpareEnrollment(Student): void
+ checkForEnrollment(InstitutionalEnrollment, Long): void
+ isLikelyGraduating(InstitutionalEnrollment): boolean
IM
BM
BM
Legend
Feature
BM
Broken Modularization
IM
Insufficient Modularization

Figure 1.1: Hypothetical example of a DP that should be early identified and removed

performed. Nevertheless, performing effective refactorings also ended up not being a trivial task. Next, we describe how the challenges faced by developers in this example are related to the requirements we previously presented.

First, the DP detection tool revealed and reported only code smells as DP symptoms. Therefore, the developer had more work to manually extract and analyze others, such as the distribution of responsibilities (concerns) in the affected code elements. As we previously discussed, developers usually need to combine multiple symptom types for diagnosing DPs. Therefore, a tool that provides *heterogeneous information*, could help the developer to analyze and understand the DP more easily.

Second, the DP was introduced together with the creation of the *CourseService* and *InstitutionalEnrollmentService* classes. However, due to the lack of *context-sensitive detection*, the developer was overload with too many symptoms. Thus, the relevant symptoms were unconsciously ignored and the DP was allowed to remain until its effects were perceived. If the DP had been identified and removed earlier, it would probably have caused fewer side effects. In addition, the developers would not have spent time analyzing symptoms unrelated to the DP in the *Service* component.

Third, developers had to decide and execute the sequence of refactorings they believed to be the most appropriate. In the example of Figure 1.1, at least three refactorings could be performed. First, the *isLikelyGraduating* method contains implementations that should be in the *CourseService* class. Thus, an *Extract Method* and a *Move Method* should be performed to move the "Course" implementations to the *CourseService* class. Besides that, another *Move Method* should be performed to move the *completeCourse* method to the *CourseService* class. After this sequence of refactorings, the "Course" responsibilities would be entirely modularized into the *CourseService* class. Such a refactoring sequence is not always easy to decide and execute. Therefore, developers would benefit from *effective refactoring recommendations*.

Finally, developers that are not fully aware of the implemented features could perform refactorings that negatively impacts *feature modularization*. For example, they could have moved the *isLikelyGraduating* and *completeCourse* methods to a class that is not related to the implemented features. As a result, the classes involved in the refactoring could become less cohesive. Indeed, such undesired refactorings could be avoided through the use of feature-driven refactoring heuristics.

1.2
Problem Statement and Research Questions

To effectively support developers in the task of identifying and removing DPs, we need a technique that addresses the challenges illustrated by our example. This need motivates our main goal, which is *to provide effective support for developers in the identification and refactoring of design problems*.

We divided our main goal into three specific goals. The first specific goal is focused in *helping developers to effectively identify design problems*. With this goal we tackled the first challenge of this research. To guide this research towards such a goal, we defined our first research question as follows:

RQ1. How to effectively support developers in the identification of design problems?

There is growing evidence that each DP manifests itself in the system through multiple symptoms (Oizumi *et al.* 2016, Sousa *et al.* 2018). For instance, in a study from my Master's dissertation (Oizumi *et al.* 2016), there was a consistent finding that DPs are often indicated by symptoms that flock together in program locations such as classes, hierarchies, and components.

Therefore, to answer RQ1, we started with a multi-method study to investigate the use of symptoms combinations for diagnosing DPs (Chapter 3). Such study provided evidence that the effective support for finding DPs requires the exploration of multiple characteristics of DP symptoms. Symptoms are often inadequately explored and presented by existing tools, which tends to frustrate developers. As we have observed (Chapter 3), heterogeneous information is necessary so that developers can reason properly about potential refactoring opportunities.

Nevertheless, we also observed that only using heterogeneous information is not always enough. Indeed, our findings indicate that besides using heterogeneous sources of symptoms, developers need assistance for ranking and filtering the recommendations to specific contexts (Oizumi *et al.* 2018). Such a need is corroborated by multiple studies from the literature (e.g., (Alizadeh and Kessentini 2018, Rebai *et al.* 2020)).

Thus, we conducted an empirical study for investigating the filtering and ranking of refactoring candidates to specific contexts (Chapter 4). In this study, we evaluated the use of multiple criteria related to aspects, such as the number of symptoms, the implemented features, and the history of changes. Our results showed that no criteria is uniformly effective on any project. Nevertheless, we observed that considering the number of symptoms for filtering refactoring candidates is often a good choice.

Aforementioned studies provided us with an initial comprehension about the effective identification of DPs. However, our main goal is also focused in the refactoring of DPs, which was not directly covered by them. Therefore, to move towards our main goal, we define a second specific objective as *defining the key requirements for effectively supporting the identification and refactoring of design problems.*

In fact, despite the existence of multiple studies and techniques related to DP identification and refactoring, there is little to no consensus on which are the key requirements for supporting developers in such tasks. Therefore, an investigation about such requirements is essential. To guide this investigation, we defined our second research question as follows:

RQ2. Which are the key requirements for supporting the identification and refactoring of design problems?

For answering RQ2, we expanded our previous investigations through an empirical study about the relation of symptoms, DPs, and refactorings performed in practice (Chapter 5). We investigated DPs and refactorings in the context of open-source C# and Java projects. This study, provided us with important insights about the removal of DPs through refactorings. As a result, we defined an initial set of requirements for refactoring recommendation techniques.

Next, we proposed and evaluated a novel recommendation technique (Chapter 6). Such a technique explored the use of code smells and refactoring patterns for generating effective recommendations. Besides presenting promising results, such study also expanded our understanding about the requirements for recommendation techniques. More than that, we found evidence on how such requirements could be met.

As a result, the studies reported above helped us to define four key requirements, namely *Heterogeneous Symptoms*, *Context-Sensitive Detection*, *Feature Modularity Awareness*, and *Effective Recommendations*. We presented and discussed all of them earlier in the beginning of this chapter. Such requirements answered RQ2 and motivated our third specific goal, which is *to propose and evaluate a technique based on the key requirements for refactoring recommendation techniques.*

Given aforementioned goal, we defined our third research question as follows:

RQ3. What is the effectiveness of using feature-driven and context-aware strategies in search-based refactoring recommendation?

As we previously discussed, developers usually avoid refactoring code elements that are out of their context of interest (Alizadeh and Kessentini 2018, Rebai *et al.* 2020). Context-sensitive detection is helpful for limiting the scope of analysis to specific program locations and also to rank and filter the refactoring candidates provided to the developers (Oizumi *et al.* 2019, Vidal *et al.* 2019). A code review task, for example, is focused on analyzing the quality and impact of a specific set of changes. In this case, it would be helpful to limit the DP symptoms and, consequently, the refactoring recommendations to the code elements changed in the reviewed task.

In fact, in this thesis, we evaluated several criteria for filtering and prioritizing refactoring candidates (Vidal *et al.* 2019). However, our results show that the effectiveness of the different criteria varies according to certain factors, such as the project characteristics and the developer involved in the refactoring task. There is also evidence from the literature showing that refactorings tend to be motivated more by changes in features than by the occurrence of DPs symptoms (Silva, Tsantalis and Valente 2016). DPs tend to be resolved along changes motivated by feature additions, enhancements or deletions (Paixão *et al.* 2020).

Given all the reason mentioned above, we observed that an effective refactoring technique must support filtering of DP symptoms based on recurring contexts, such as code elements modified in a task, code elements realizing one or more features, modules owned by the developer (or the team), code elements with high change- or bug-proneness, among others (Oizumi *et al.* 2019). In addition, the developers must be able to customize their context of interest if necessary. Finally, the recommendations should help the developers to modularize the implemented features.

As we previously discussed, many refactorings performed in practice are not effective (Bibiano *et al.* 2019, Bibiano *et al.* 2020, Cedrim 2018, Rebai *et al.* 2020). In fact, we have observed that developers tend to refactor code elements affected by multiple and diverse symptoms (Oizumi *et al.* 2019). Nevertheless, such refactorings are often unable to completely remove the DPs. Thus, besides improving the process of identifying DP symptoms, it is necessary to help developers in the decision of which refactoring sequences should be performed for removing each potential DP.

There are multiple studies – e.g., (Bavota *et al.* 2013, Alizadeh and Kessentini 2018, Alizadeh *et al.* 2019, Alizadeh *et al.* 2019b, Rebai *et al.* 2020, Yamanaka *et al.* 2021) – that propose automated techniques for refactoring recommendation. However, none of them addresses all the requirements previously presented in this thesis (see Section 2.6). Therefore, as a last step of the research, we developed a new technique called *OrganicRef*. The purpose of *OrganicRef* is to bring together in a single technique all requirements that we conjecture are essential for a refactoring recommendation technique.

For fulfilling the requirements, *OrganicRef* detects DPs through information extracted from the project's design and source code. It relies on a topic modeling algorithm for finding existing features in the project. The features information is then combined with abnormal quality measures and code smells to find DPs. For creating refactoring recommendations, *OrganicRef* relies on a new refactoring recommendation strategy, which combines rule-based and feature-driven refactoring heuristics. Finally, we included in *OrganicRef* the use of search-based algorithms for deriving improved refactoring recommendations.

To answer RQ3, we implemented a reference tool for *OrganicRef* and conducted an empirical evaluation involving six open source projects. This evaluation showed that *OrganicRef*'s feature-driven and context-aware recommendation strategies are able to significantly outperform a baseline strategy (Chapter 7).

1.3 Summary of Contributions

We present in Figure 1.2 an overview of the contributions of this research. Rounded rectangles represent our specific goals, while the rectangles in the center of the figure represent the contributions and the related publications. The arrows indicate the relationships between the goals and contributions. All of them are related to our main goal and to the research questions presented in the previous section.

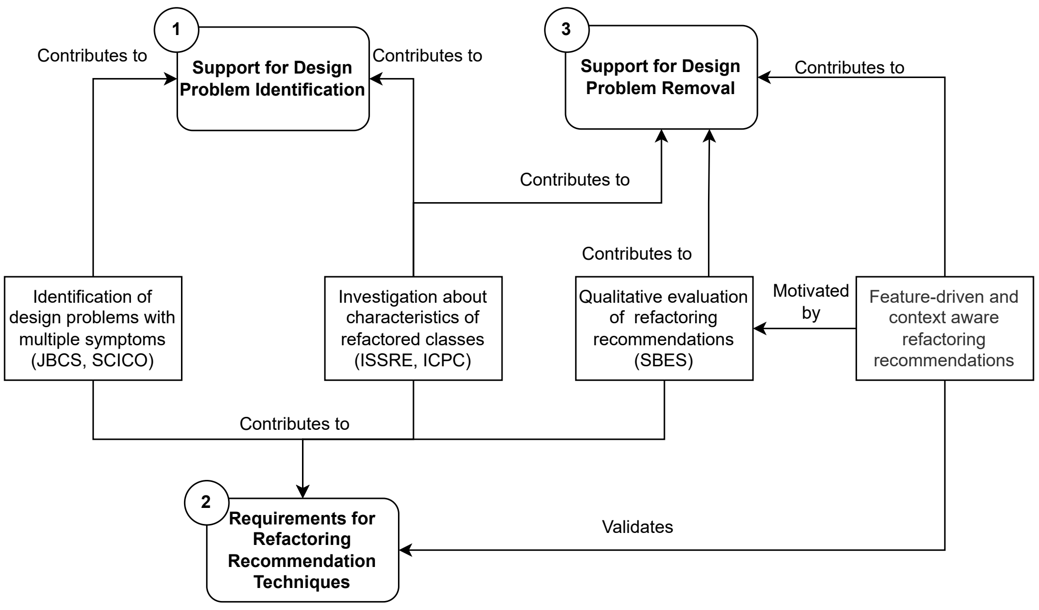

1
Support for Design Problem Identification
3
Support for Design Problem Removal
2
Requirements for Refactoring Recommendation Techniques
Identification of design problems with multiple symptoms (JBCS, SCICO)
Investigation about characteristics of refactored classes (ISSRE, ICPC)
Qualitative evaluation of refactoring recommendations (SBES)
Feature-driven and context aware refactoring recommendations
Contributes to
Contributes to
Contributes to
Contributes to
Contributes to
Contributes to
Motivated by
Validates

Figure 1.2: Overview of the contributions of this research

As depicted in Figure 1.2, our first goal was to provide explicit **support for design problem identification**. We conducted a study that provided evidence on the use of multiple co-located symptoms for identifying design problems (Oizumi *et al.* 2018). Such a study was based on two experiments involving professional software developers. This study provided empirical evidence about the effective identification of design problems (Oizumi *et al.* 2018).

One of the main findings of the first study was regarding the need for efficient mechanisms for filtering symptoms and recommendations to delimited contexts. Thus we conducted a subsequent study to propose and evaluate multiple ranking and filtering criteria (Vidal *et al.* 2019). In such study, we found different scenarios in which each criterion may be applied. Such a result is also related to our first contribution.

In addition to the contributions already presented, our studies revealed the need for understanding the requirements for techniques focused in both the identification and refactoring of design problems. Thus, to obtain a complementary point of view on the use of multiple co-located symptoms, we conducted a study on the characteristics of classes that were refactored to remove design problems (Oizumi *et al.* 2019, Eposhi *et al.* 2019). This investigation provided us with multiple insights about the effective identification and refactoring of DPs. As a result, we achieved our second specific goal, which is the definition of **key requirements for refactoring recommendation techniques**.

Given that we identified aforementioned requirements, we continued our research by focusing in the investigation of appropriate **support to remove design problems through refactorings**. Therefore, we proposed and evaluated a technique aimed at providing developers with refactoring recommendations. This technique was developed in the context of one of our collaborations, resulting in publications at MSR (Sousa *et al.* 2020a) and SBES (Oizumi *et al.* 2020). My exclusive contribution in those collaborative studies was the understanding of (un)desirable characteristics of refactoring recommendations. In particular, our main contribution in such studies – which is part of this thesis – was to conduct a qualitative assessment of the refactoring recommendations. This assessment helped us to reveal multiple factors that are related to the acceptance or rejection of refactoring recommendations.

Despite having proposed the refactoring recommendation technique described above, our qualitative evaluation showed that the key requirements were not fully met by it. For example, we observed that the sole use of rule-based refactoring heuristics is not enough for completely removing DPs. We also observed that no existing technique satisfactorily fulfills all the requirements (Section 2.6). Therefore, to fill such a gap in the literature, we proposed and evaluated the *OrganicRef* technique.

As we previously described, *OrganicRef* addresses four key requirements, which are (1) consideration of heterogeneous symptoms, (2) context-sensitive detection, (3) feature awareness, and (4) effective recommendations. To meet such requirements, *OrganicRef* allows the use of multiple context-selection strategies. It also relies on Topic Modeling (Silva, Galster and Gilson 2021) for identifying the features implemented by code elements. Then, *OrganicRef* combines feature-driven and rule-based heuristics to generate refactoring recommendations for a delimited context. The generated recommendations are later improved through the use of search-based algorithms. An evaluation of *OrganicRef* showed that its recommendations are able to significantly improve the design of refactored elements. Among the evaluated search-based algorithms, we found robust evidence indicating that NSGA-II best fits the purpose of feature modularization. This happens because NSGA-II is able to explore a larger set of possible solutions. As a result, it is able to find solutions with higher positive impact on the modularization of features.

1.4 Thesis Outline

This thesis is organized as collection of papers, which were published or submitted during this PhD research. To make the text more coherent and fluid, the chapters presented here were adapted from the original manuscripts. We adjusted the terminology and removed content unrelated to this thesis.

The chapters are organized as follows. Chapter 2 presents the basic background and related work for the whole thesis. In Chapter 3 we present our study on the use of multiple symptoms for effectively identifying DPs. Chapter 4 is dedicated to presenting our study on filtering and prioritization criteria for refactoring opportunities. Chapter 5 presents an empirical study on the relation of symptoms, DPs, and refactorings. Next, in Chapter 6 we propose and evaluate a rule-based refactoring recommendation technique. Chapter 7 is dedicated to presenting our final contribution, which consists of an improved refactoring recommendation technique and its evaluation. Finally, we conclude this thesis in Chapter 8.

2
Background and Related Work

The literature on software design and refactoring is extensive and may contain different definitions for concepts used in this thesis. Therefore, this chapter presents the key concepts for our research along with their definitions. We organize the presentation of such concepts as follows. Section 2.1 presents our definitions for software design, quality attributes and other related concepts. In Section 2.2 we present our definition of design problem. The types of design problem symptoms considered in this work are described in Section 2.3. Section 2.4 contains the definition of refactoring. Besides the key concepts presented here, other concepts and definitions that are specific to each study are presented in their respective chapters.

Moreover, this chapter provides an overview of the main studies related to this thesis. Such an overview of the literature is useful for mapping what are the main differences and contributions of this research in comparison to the literature. Thus, we performed a review of the literature on DPs and refactoring. The result of such a review is summarized in this chapter in Section 2.6.

2.1
Software Design and Quality Attributes

Software design is the result of multiple design decisions taken by the project's stakeholders to fulfill the desired quality attributes (Bass *et al.* 2003, Booch 2004, Freeman and David 2004, Taylor *et al.* 2009, Sousa *et al.* 2018). Quality attributes determine which characteristics should be taken in account when evaluating the quality of a software project (ISO-IEC 25010 2011, Bass *et al.* 2003). Examples of quality attributes include maintainability, reliability, portability, among others (ISO-IEC 25010 2011).

There are two main stages in which design decision may be taken (Booch 2004). The first one is called *early software design* (or software architecture design) and is focused on defining the overall organization of a software system into components (or sub-systems), interfaces and their relationships (Booch 2004, Taylor *et al.* 2009). Therefore, the early design is also responsible for defining the *intended design* of a project (Bass *et al.* 2003, Perry and Wolf 1992). The intended design is the result of design decisions that should be followed by the project's implementation.

The second stage of software design comprises the so called *detailed design*, which is focused on achieving more specific decisions governing the design of each design component (Booch 2004, Taylor *et al.* 2009). Design components are elements which address one or more *features* (Taylor *et al.* 2009), which are also called *concerns*. Each *feature* represent a functional or non-functional requirement that the project should satisfy.

2.2 Design Problems

A Design Problem (DP) occurs when stakeholders make decisions that negatively impact quality attributes (Li, Avgeriou and Liang 2015), (Lim, Taksande and Seaman 2012), (Besker, Martini and Bosch 2017). An example of DP is the so called Fat Interface (Martin and Martin 2006). This form of DP occurs when a single interface provides multiple and unrelated operations, making it difficult to use and increasing the chance of introducing defects to its clients (Martin and Martin 2006, Oizumi *et al.* 2016). Due to the negative impact caused by DPs, software systems have often been discontinued or redesigned when DPs were allowed to persist (MacCormack, Rusnak and Baldwin 2006). Thus, to be able to maintain the system's quality, developers need to identify and to confirm the existence of DPs. In this thesis we investigated multiple symptoms as we describe in details along the next chapters. Nevertheless, our main contributions are focused on three types of DPs, namely Feature Overload, Scattered Feature, and Complex Component. We present below a short description of them:

- **Feature Overload** occurs when a design component is overloaded with multiple unrelated features (e.g., authentication and billing) (Brown *et al.* 1998), (Oizumi *et al.* 2016). As a result of this problem, changes related to one feature may cause side effects to other unrelated features located in the same component.
- **Scattered Feature** occurs when multiple non-cohesive design components are responsible for realizing the same feature (Brown *et al.* 1998), (Oizumi *et al.* 2016). Changing the scattered feature tends to be chal-

lenging since developers need to change all the components involved in realizing it.

– **Complex Component** occurs when the implementation of a component has a high cyclomatic complexity (Lanza and Marinescu 2006). Components affected by this problem tend to be error prone and difficult to maintain.

We decided to focus on those DP types due to the following reasons. First, Feature Overload and Complex Component are frequently considered relevant by developers (Palomba *et al.* 2014). Second, there is evidence that Scattered Feature and Feature Overload reflect important maintainability aspects (Yamashita and Moonen 2013). These three DP types are also of major severity if compared with many others, and their resolution would eliminate/reduce other inner problematic structures (e.g., a *Long Method* related to a Feature Overload) (Abbes *et al.* 2011, Cedrim *et al.* 2017). Finally, their removal is usually not trivial, requiring a combination of multiple refactorings (Sousa *et al.* 2020a, Bibiano *et al.* 2020).

Next, we present an illustrative example to show how quality attributes may be impacted by DPs. Figure 2.1 shows a partial view of the OpenPOS (Oizumi *et al.* 2019) system before and after a degraded structure has been refactored. OpenPOS is a system that provides sales features. One of the functionalities of OpenPOS comprises the generation of payment slips. In Brazil, payment slips serve for clients to make payments at any bank. Developers of OpenPOS implemented this feature in the *PaymentSlip* sub-component. To protect system information, this sub-component was strongly coupled to the *Authentication* component.

Unfortunately, the strong coupling with the *Authentication* component led to a side effect on the reusability of *PaymentSlip* sub-component. Reusability is a sub-category of maintainability that indicates the degree to which a component can be re-used in two or more systems (ISO-IEC 25010 2011). Since *PaymentSlip* was so coupled to the *Authentication* component, it could not be reused in other systems. In this context, developers have to refactor the *PaymentSlip* sub-component to reduce the coupling with *Authentication* component. Additionally, refactoring this type of DP is fundamental to avoid code duplication among systems and rework. In Section 2.4, we explain the structure obtained after the refactoring.

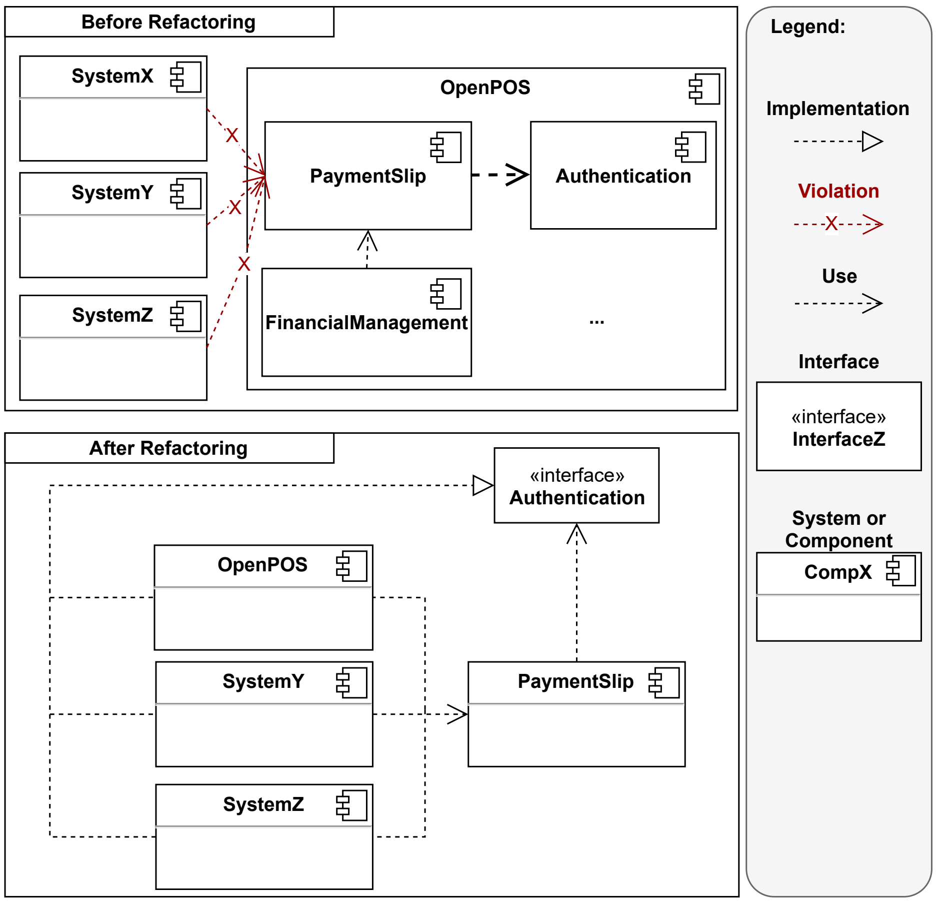


Figure 2.1: Example of design problem impacting reusability

2.3 Design Problem Symptoms

Sousa et al. (Sousa *et al.* 2018) identified five categories of symptoms upon which developers frequently rely to identify DPs. Similarly to other related work (Yamashita *et al.* 2015), (Macia *et al.* 2012a), (Oizumi *et al.* 2016), (Oizumi *et al.* 2018), they observed that developers tend to combine multiple symptoms, taking into account characteristics such as diversity and density to decide if there is a DP or not. For refering to (sets of) code elements impacted by multiple symptoms we use terms like *stink code* and *agglomeration of code smells*. Such terms detailed defined in Chapters 3 and 4.

We selected a sub-set of three symptom categories that can be automatically detected using state-of-the-art tools, namely *abnormal quality measures*, *implementation smells*, and *principle violations*.

Abnormal quality measures indicate violations of characteristics that are considered fundamental for software design, such as coupling, cohesion, and complexity. *Coupling*, for example, indicates the number of classes that a single class uses, and cyclomatic complexity (*complexity* for short) measures the structural complexity of the code.

Implementation smells. which are also called *code smells*, are surface indicators of quality degradation that may be related to DPs (Fowler 1999), (Sharma and Spinellis 2018). This symptom category have been investigated by different researchers (e.g., (Murphy-Hill and Black 2010), (Yamashita and Moonen 2012), and (Moha *et al.* 2010)). An example of implementation smell type is the Long Method. This type of smell usually leads to DPs related to modifiability.

In object-oriented systems, DPs usually impact object-oriented design characteristics, such as abstraction, encapsulation, modularity, and hierarchy. Therefore, the second symptom category we used comprises the **principle violations**, which are symptoms that may indicate the violation of common object-oriented principles (Martin and Martin 2006), (Sharma and Spinellis 2018). An example of object-oriented principle is the *Single Responsibility Principle* (SRP). The SRP determines that each class should have a single and well defined responsibility in the system (Martin and Martin 2006). An example of symptom that may be used for finding SRP violations is the *Insuficient Modularization* (Suryanarayana, Samarthyam and Sharma 2014). This symptom occurs in classes that are large and complex, possibly indicating the occurrence of the *Feature Overload* DP.

2.4 Refactoring

Refactoring consists in transforming the source code structure without changing the functional behaviour of the system (Fowler 1999). Thus, we consider that refactoring is any structural software change that is aimed at improving quality attributes of the system's design. There are multiple refactoring types cataloged in the literature (e.g., (Fowler 1999) and (Tsantalis *et al.* 2018)). Each refactoring type is applied to perform a specific structural transformation. For example, Move Method aims at transferring a method from one class to another.

According to Murphy-Hill and Black (Murphy-Hill and Black 2008b), refactoring can be classified into two tactics, which are floss refactoring and root canal refactoring. On one hand, floss refactoring is characterized by refactoring changes intermingled with other kinds of source code changes, such as adding new features and fixing bugs. The aim of floss refactoring is to keep structural quality as a means to achieve other goals. On the other hand, root canal refactoring aims at exclusively removing or reducing the intensity of DPs. A root canal refactoring consists of only refactoring changes; it should not be performed in conjunction with other non-refactoring changes. Based on that, in this proposal, our focus is on root canal refactorings as they are explicitly aimed to remove DPs. Thus, from now on, whenever we talk about refactoring in this work, we'll be referring to root canal refactoring.

To illustrate our definition of refactoring, let's return to the example of Figure 2.1. As previously discussed, multiple different systems of the same company began to require a payment slip feature. Therefore, developers were asked to remove the reusability DP by refactoring the *PaymentSlip* sub-component. The refactorings consisted of extracting an interface for authentication and moving the existing authentication implementation to another component. This way, each system that needs to use the *PaymentSlip* component must specify an authentication component that meets the interface specifications required by *PaymentSlip*. After refactoring the *PaymentSlip* sub-component, besides removing the DP, it is expected the removal of symptoms such as the *Hub-Like Modularization* (Sharma and Spinellis 2018), (Sharma 2020).

2.5 Search-Based Software Engineering

As described in Chapter 7, in this thesis we are using search-based software engineering (SBSE) (Harman and Jones 2001, Harman *et al.* 2012) techniques as a way to optimize refactoring recommendations. Thus, in this section we describe the fundamental concepts of SBSE that are required for understanding our work.

Software engineering problems often depend on competing factors and constraints that are often conflicting (Harman and Jones 2001). In this sense, SBSE techniques can be applied to find near-optimal solutions. In this context, different metaheuristic algorithms can be applied, such as genetic algorithms (GAs), simulated annealing and tabu search (Harman *et al.* 2012). The use of SBSE for refactoring is called Search-Based Refactoring (Harman and Tratt 2007).

Initially, search-based refactoring techniques were mono-objective (Harman and Tratt 2007), which means that the search algorithm performs optimizations based on a single fitness function. A fitness function is defined to assess the quality of the generated (*i.e.*, optimized) solutions based on the goal we want to achieve. However, according to Harman and Tratt (Harman and Tratt 2007), the use of mono-objective algorithms for refactoring is problematic as, among other limitations, they require the complex combination of multiple metrics into a single fitness function.

Therefore, in this work, we focus on the use of GAs (Goldberg 1989), which are inspired by the Darwinian Evolution theory (Harman *et al.* 2012). Some GAs are adapted for optimizing multi-objective problems. Such GAs are called Multi-Objective Evolutionary Algorithms (MOEAs).

The most popular and largely applied MOEA is the *Non-dominated Sorting Genetic Algorithm* (NSGA-II) (Deb *et al.* 2002). A multi-objective problem depends on multiple factors (objectives) that may be conflicting and, therefore, there is not a single possible solution. Thus, there may be multiple quasi-optimal solutions that represent the trade-off between the different objectives. Such solutions are called non-dominated solutions and form the *Pareto front*.

Using a MOEA, we start from an initial population composed by alternatives for refactoring recommendation. After that, search operators – namely, selection, crossover, and mutation – are applied to evolve the population throughout multiple generations. The selection operator is responsible for selecting solutions that present the best *fitness* values to survive as parents for the next generation. The crossover operator combines parts of two parent solutions to create a new one. Finally, the mutation operator randomly changes a solution.

Based on the application of selection, crossover, and mutation operators a new population is created to replace the parent population. This new population is called *offspring*.

This optimization process is repeated over multiple generations according to the type of algorithm and the parameters defined for its execution. The final result is the Pareto front, which will be composed by multiple quasi-optimal solutions.

2.6
Related Work

In this section we present the literature related to this proposal. In Section 2.6.1 we discuss studies about DP symptoms. Sections 2.6.2 and 2.6.3 are dedicated to discussing studies on the impact of refactoring on symptoms and on DPs.

In Section 2.6.4 we present studies about guidelines for creating refactoring techniques and existing refactoring recommendation techniques. Finally, in Section 2.6.5 we discuss some recent secondary and tertiary studies on DPs and refactoring.

2.6.1 Symptoms of Design Problems

Different studies have proposed and evaluated techniques that use different symptoms for identifying DPs (Moha *et al.* 2010), (Murphy-Hill and Black 2010), (Le *et al.* 2018), (Ran *et al.* 2015), (Oizumi *et al.* 2016). Many of them use only code smells as symptoms for the identification of DPs. Nevertheless, there is consistent evidence (Macia *et al.* 2012a), (Macia *et al.* 2012), (Macia *et al.* 2012b), (Oizumi *et al.* 2015) that individual code smells are not enough to accurately indicate the presence of DPs. For this reason, in a previous study (Oizumi *et al.* 2016), we have proposed an alternative approach to identify DPs with the combination of multiple code smells. In this study, we investigated to what extent code smells could "flock together" to realize a DP. We concluded that certain combinations of code smells are consistent indicators of DPs. Despite such result, in a recent study that is part of this thesis proposal (Oizumi *et al.* 2018), we observed that our previously proposed technique may not be effective in practice. In addition, in (Oizumi *et al.* 2016), we did not verify whether multiple code smells occur more frequently in classes that are actually refactored by developers.

Sousa's et al. (Sousa *et al.* 2018) revealed that, in practice, developers use multiple heterogeneous symptoms to diagnose DPs. However, although they have used a systematic methodology to propose their theory, it was not validated. In this thesis, we intend to overcome such limitation by investigating the use of multiple symptoms for generating effective refactoring recommendations.

Mamdouh and Mohammad (Alenezi and Zarour 2018) conducted an study involving six open source systems to investigate if DP symptoms are removed as the system evolves. They observed that, in general, the density of symptoms undergo few changes throughout the systems evolution. We conducted a similar analysis in one of our studies (Oizumi *et al.* 2019). Nevertheless, our study was focused on refactored classes, while they analyzed all classes of systems indistinctly. Moreover, we intend to go beyond the investigation of DP symptoms through the investigation of refactoring sequences that can be applied for removing DPs.

2.6.2 Impact of Refactoring on Symptoms

The impact of refactoring on symptoms was also vastly studied by different researchers – e.g., (Bavota *et al.* 2015), (Chávez *et al.* 2017), and (Cedrim *et al.* 2017). Similarly to us, Bavota et al. (Bavota *et al.* 2015) investigated whether refactorings occur in classes with symptoms such as the code smells. Overall, they observed that none of the investigated symptoms are strong indicators of need for refactoring. In our case study (Oizumi *et al.* 2019), unlike them, both code smells and principle violations showed to be strong indicators of the need for refactoring. Finally, like us, they also observed that code smells are not usually removed by means of refactorings. However, we are the first to investigate the impact of refactorings on different categories of DP symptoms (implementation smells and principle violations).

Cedrim et al. (Cedrim *et al.* 2017) also investigated the impact of refactorings on code smells. Different from us, they evaluated the impact of refactorings by each individual commit. As in the work of Bavota et al. (Bavota *et al.* 2015), they collected the refactorings using an automated tool. Although this approach results in the collection of a larger volume of refactorings, it is not able to collect only the refactorings intentionally performed by the developers. Also, they did not investigated the symptom characteristics of refactored classes.

2.6.3 Refactorings for Removing Design Problems

Regarding the investigation of refactorings for removing DPs, there are some studies in the literature – e.g., (Lin *et al.* 2016), (Rizzi *et al.* 2018), (Kumar and Kumar 2011), (Zimmermann 2017), and (Rachow 2019). The work of Kumar and Kumar (Kumar and Kumar 2011), for example, reported an industrial case study about the conduction of refactorings in a payment integration platform. Such study presented the key drives that motivated the refactoring, including DPs such as the Feature Overload. According to the presented results, the refactorings led to significant improvement in application stability and throughput. Lin *et al.* (Lin *et al.* 2016) proposed an approach to guide developers in applying refactorings for improving the architectural design. In their approach, the developers indicate the target design, and then the approach suggests stepwise refactorings that will change the source code to meet the target design.

Rizzi, Fontana and Roveda (Rizzi *et al.* 2018) proposed a tool to remove a DP known as Cyclic Dependency. Their tool suggests the refactorings steps that the developer should follow to remove the Cyclic Dependency. Finally, Rachow (Rachow 2019) proposed a research idea to develop a framework that (i) detects a DP, (ii) selects and prioritizes refactorings, and (iii) shows to developers the impact on the design. Although his framework has not been implemented, the author indicates seven types of DPs that he intends to support with his framework.

2.6.4 Refactoring Recommendation and its Requirements

Guidelines for refactoring recommendation. Bavota *et* al. (Bavota *et al.* 2014) presented guidelines to build and evaluate refactoring recommendation tools. However, unlike us, their guidelines are not based on qualitative empirical data. They generated several recommendations based on state-of-the-art papers and based on the authors' prior knowledge on the subject. In addition, according to the authors themselves, there are still limitations and challenges regarding the recommendation of composite refactorings, which are not covered by their guidelines.

Tsantalis *et* al. (Tsantalis, Chaikalis and Chatzigeorgiou 2018) presented lessons learned from 10 years of research with the JDeodorant tool. Their lessons are mostly focused on the adoption of recommendation tools in practice. We, on the other hand, focus on guidelines for the effective recommendation of composite refactorings for removing DPs. As far as we know, we are the first to identify such guidelines through systematic qualitative studies.

Composite recommendation. The removal of DPs may required the application of sequences of refactorings (Oizumi *et al.* 2020, Cedrim 2018), which are know as *composite refactorings*. The proposal and evaluation of refactoring tools have been the focus of several studies (*e.g.*, (Bavota *et al.* 2014, Tsantalis, Chaikalis and Chatzigeorgiou 2018)). Despite making important contributions, such studies have not considered the application of composite refactorings for removing DPs. In fact, when composite refactorings are not properly applied, a DP may not be completely removed (Bibiano *et al.* 2019, Bibiano *et al.* 2020). Other studies focused on proposing recommendations for composite refactorings (Cedrim *et al.* 2017, Bibiano *et al.* 2019).

Cedrim *et* al. (Cedrim *et al.* 2017), for example, investigated the effect of single refactorings on code smells. They found that single refactorings often introduce or do not fully remove code smells. Then they suggest some composites that could have removed code smells based on Fowler's catalog (Fowler 1999). However, these suggestions were based only on anecdotal evidence. Bibiano *et* al. (Bibiano *et al.* 2019) investigated composite refactorings on 57 software projects. They found some composites that removed code smells, generating recommendations from these results. However, they detected only composites in the scope of individual elements. They did not analyze composites that involve and affect multiple elements.

Composite refactorings were also the focus of Sousa *et* al. (Sousa *et al.* 2020a) in the context of a collaboration of ours. In this study, Sousa *et* al. proposed sets of composite patterns for the removal of different DP types. For creating the composite patterns, we on data mined from 48 industry-strength Java projects. In a subsequent collaborative study (Oizumi *et al.* 2020), we developed and evaluated a pattern-based refactoring recommendation technique. Despite showing promising results, we observed that, in many cases, the pattern-based recommendations are not enough for removing DPs. This helped us to better understand which requirements a technique should met to provide effective recommendations.

Search-based techniques. Multiple studies have used Search-Based Software Engineering techniques to create recommendations of composites – e.g., (Alizadeh and Kessentini 2018), (Ouni *et al.* 2017), (Alizadeh *et al.* 2019), and (Alizadeh *et al.* 2019b). Ouni *et* al. (Ouni *et al.* 2017), for example, introduced an automated approach for refactoring recommendation, called MORE, using a genetic algorithm that is focused on three objectives: (1) improve design quality, (2) fix code smells, and (3) introduce design patterns.

Alizadeh et al. (Alizadeh *et al.* 2019b) proposed a technique called RefBot. This technique is focused in providing context sensitive refactoring recommendations. More specifically, they provide recommendations in the context of pull requests, which is a mechanism frequently used for reviewing and integrating source code changes. Their technique relies on a software bot, which analyzes the changes in a pull request and uses a search-based algorithm for generating refactoring recommendations.

In general, we have observed important differences when comparing existing search-based refactoring techniques to the one proposed in this thesis. First, such techniques usually rely only on traditional metrics (e.g., number of symptoms, QMOOD and OO metrics) for finding refactoring opportunities and evaluating the generated solutions (Mariani and Vergilio 2017). We, on the other hand, rely on the combination of heterogeneous symptoms. Moreover, despite the existence of some techniques (e.g., (Alizadeh *et al.* 2019b)) that provide context-sensitive recommendations, the context is usually fixed rather than flexible as we intend to provide. Finally, few to no techniques take into account feature modularization for generating refactoring recommendations.

Recommendation based on semantic and design information. Some studies go beyond the use of traditional symptoms and metrics for recommending refactorings (Bavota *et al.* 2013, Rebai *et al.* 2020, Yamanaka *et al.* 2021, Nyamawe *et al.* 2019). Bavota et al. (Bavota *et al.* 2013) proposed a technique for recommending *Move Method* refactorings. Their technique is based on Topic Modeling and combines structural and textual information for finding *Move Method* refactoring opportunities. Differently from us, they use Topic Modeling only at the method level. We, on the other hand, apply a Topic Modeling algorithm for finding software features in classes and methods. We use the extracted features for recommending multiple refactoring types, such as *Extract Class*, *Move Method* and *Move Field* (see Chapter 7).

Nyamawe et al. (Nyamawe *et al.* 2019) trained a Machine Learning (ML) model for relating feature requests with the refactorings needed for completing the feature's implementation. Based on such a model, they created a technique which recommends refactorings for enable the implementation of future feature requests. Like us, they aim to provide recommendations within the context of interest to the developers. Unlike them, the technique proposed in this thesis aims to make it possible to recommend refactorings in multiple contexts, not being restricted to feature requests.

The work of Rebai et al. (Rebai *et al.* 2020) is focused in helping developers to find and complement incomplete refactorings. They find incomplete refactorings through the developers' refactoring intention manifested in commit messages. Their work also takes a step towards supporting the recommendation of refactorings within the developers' context of interest. Similarly to them, we also focus in considering the context of interest for recommending refactorings. The main difference between us is that they focus in completing existing refactoring sequences, while we focus in recommending refactoring composites for removing DPs.

Yamanaka et al. (Yamanaka *et al.* 2021) presented a new technique for recommending Extract Method refactorings based on predicted method names for code extraction candidates. The prediction of method names is based on an existing technique called code2seq. This technique relies on a classification model to bind code characteristics with sequences of words. The disadvantage of this work is that its accuracy depends on a large model that is compatible with the domain of the project to be refactored. Furthermore, such a technique only provides *Extract Method* recommendations, which is not always sufficient for removing DPs.

2.6.5
Secondary and Tertiary Studies on DPs and Refactoring

There are also multiple secondary and tertiary studies related to DPs and Refactoring - e.g., (Lacerda *et al.* 2020, Abid *et al.* 2020, Mariani and Vergilio 2017). Lacerda *et* al. (Lacerda *et al.* 2020) conducted a tertiary literature review on DP symptoms and refactoring. Their results presents multiple observations on what is well known about DP symptoms and refactoring and what are the open challenges. Being a tertiary study, the results presented by them were obtained through the synthesis of secondary studies. Therefore, their study provides evidence that helped us in the motivation of this thesis. Indeed, our goal is related to some of the challenges reported by them.

Abid *et* al. (Abid *et al.* 2020) conducted a systematic literature review about refactoring. Based on the investigation of 3,183 papers, they built a taxonomy to classify existing research about refactoring. Based on that, they identified the main trends and the gaps in the software refactoring literature. Their results shows that refactoring is a topic of extreme relevance to the software engineering community and there are still many open challenges in this research field.

In the work of Mariani and Vergilio (Mariani and Vergilio 2017), a systematic literature review on search-based refactoring was presented. The results of this study show that several search-based techniques have been proposed recently. Nevertheless, despite the huge number of studies, there is a limitation regarding the metrics used by existing techniques to evaluate the generated refactoring solutions. Many studies evaluate their solutions based on metrics like the number of DP symptoms, the number of proposed refactorings (i.e., number of changes), and on traditional metrics such as QMOOD and OO metrics. However, as discussed in Chapter 1, effective detection and refactoring of DP problems often requires a combination of heterogeneous symptoms. Therefore, this thesis contributes in this direction through a new technique based on the combination of traditional symptoms and metrics with design information.

3
On the Identification of Design Problems in Stinky Code: Experiences and Tool Support

Developers often have to locate design problems in the source code. Several types of design problems may manifest as code smells in the program. As we described in Section 2.3 of Chapter 2, a code smell is a source code structure that may reveal a partial hint about the manifestation of a design problem. Recent studies suggest that developers should ignore smells occurring in isolation in a program location. Instead, they should focus on analyzing stinkier code, i.e. program locations – e.g., a class or a hierarchy – affected by multiple smells (Section 2.3). There is evidence that the stinkier a program location is, more likely it contains a design problem.

However, there is no empirical evidence on whether developers can effectively identify a design problem in stinkier code. Developers may struggle to make an analysis of inter-related smells affecting the same program location. Besides that, the analysis of stinkier code may require proper tool support due to its analysis complexity. However, there is little knowledge on what are the requirements for a tool that helps developers in revealing stinkier program locations. As a result, developers may not be able to identify design problems due to tool issues.

Given that our first specific objective is to support the effective identification of design problems, in this chapter we proposed Organic – a tool supporting the analysis of stinky code. We applied a mixed-method approach to analyze if and how developers can effectively find design problems when reflecting upon stinky code – i.e., a program location affected by multiple smells. Finally, we evaluated if Organic could be used by developers to identify design problems in practice. For this evaluation, we used a method from the Semiotic Engineering theory. This method enabled us to evaluate what are the tool issues that may hinder the identification of design problems in stinky code.

Our study revealed that only 36.36% of the developers found more design problems when explicitly reasoning about multiple smells as compared to single smells. Moreover, 63.63% of the developers reported much lesser false positives when using the first approach as compared to the latter. The second study, in its turn, showed that most developers may be unable to identify design problems in stinky code without proper tool support.

Our experiences, in particular the second study, helped us to refine the features of Organic for better supporting developers in reflecting upon stinkier code. For example, analyses of stinky code scattered in class hierarchies or packages is often difficult, time consuming, and requires proper visualization support. Moreover, without effective support, it remains time-consuming to discard stinky program locations that do not represent design problems.

The two studies presented in this chapter were published in a paper at the *Journal of the Brazilian Computer Society* (JBCS) (Oizumi *et al.* 2018).

3.1 Introduction

The identification of design problems in the source code is not a trivial task (Ciupke 1999, Trifu and Marinescu 2005). Developers usually need to find hints, as code smells, in the source code that can lead to a design problem. A code smell is a structure in the source code that may provide developers with a partial indication about the manifestation of a design problem (Fowler 1999). A classical example of code smell is the *God Class*, which occurs when a class is long and complex, centralizing a considerable amount of intelligence of the system. However, the occurrence of a single smell in isolation in a program often does not represent a design problem (Macia *et al.* 2012b, Oizumi *et al.* 2016). A design problem is a design characteristic that negatively impacts maintainability (Trifu and Marinescu 2005). Recent studies reveal that design problems are much more often located in stinkier program locations (i.e., a class, a hierarchy or a package) affected by multiple smells (Abbes *et al.* 2011, Macia *et al.* 2012b, Yamashita and Moonen 2013, Yamashita *et al.* 2015, Oizumi *et al.* 2016). For instance, a *Fat Interface* (Martin and Martin 2006) is a design problem that often manifests as multiple smells in a program, affecting various classes that implement, extend, and use the interface in a program (Oizumi *et al.* 2016).

The stinkier a program location is, more likely it contains a design problem (Oizumi *et al.* 2016, Oizumi *et al.* 2015). In fact, developers tend to focus on refactoring program locations with a high density of code smells, and ignore those locations affected by a single smell (Cedrim *et al.* 2016, Cedrim *et* *al.* 2017). However, there is limited understanding if developers can effectively identify design problems in stinkier code, i.e. program locations affected by multiple smells. Indeed, existing techniques tend to focus on the detection and visualization of each single smell (Emden and Moonen 2002, Ratzinger, Fischer and Gall 2005, Murphy-Hill and Black 2010, Wettel and Lanza 2008). They do not offer a summarized view of inter-related smells affecting a program location (Oizumi *et al.* 2016). Moreover, previous studies focus on simply analyzing the correlation between design problems and code smells (Oizumi *et al.* 2016, Macia *et al.* 2012). They have not investigated if and how developers are indeed effective in the task of finding design problems in stinkier code.

Therefore, we do not know whether the analysis of multiple smells actually provides better precision for the identification of design problems. Developers may struggle to make a meaning out of inter-related smells affecting the same program location. Additionally, the analysis of stinkier code may require proper tool support due to its analytic complexity. However, there is limited knowledge on what are the requirements for a tool that supports the analysis of stinkier code. This is important because developers may not be able to identify design problems due to tool support issues. To address these matters, we defined three goals for our research: (1) provide proper support for the analysis of stinkier code, (2) assess to what extent developers are able to identify design problems in stinkier code, and (3) identify tool issues that may hinder the identification of design problems.

To achieve our first goal, we designed and implemented Organic – a tool supporting the analysis of stinky code. We used findings from previous studies (Macia *et al.* 2012b, Oizumi *et al.* 2016, Macia *et al.* 2012, Macia *et al.* 2012a, Oizumi *et al.* 2014a, Oizumi *et al.* 2017) as a start point for defining the requirements of Organic. In a nutshell, Organic supports the analysis of multiple forms of stinkier code, provides detailed information about code smells, supports the analysis of dependencies between stinky elements, provides a visualization for stinkier code, provides historical information about stinkier code, and allows developers to specify the thresholds that should be considered when identifying stinkier code. In the context of Organic, the threshold defines the minimum number of smells that a program location should have to be considered stinkier. Those features will be presented in detail in Section 3.3.

For achieving the second goal, we applied a mixed-method approach to analyze if and how developers can effectively find design problems when reflecting upon stinky code. This study comprised both quantitative and qualitative analyses. For the quantitative analysis, we compared the precision of the developers with a baseline, i.e., situations where only single smells were given to them. As we want to assess if multiple smells can help developers to reveal more design problems than single smells, we divided the developers into two groups. In the first group, we asked them to identify design problems through the analysis of stinky program locations. In the second group, we asked them to identify design problems with the analysis of single smells. After that, we inverted the groups, and we asked them to repeat the identification of design problems in a second system. In each identification task, we used the group that identified design problems with single smells as the control group. Thus, we could use the control group to measure if the analysis of stinkier program locations can improve the precision of design problem identification.

In the qualitative analysis, we performed a systematic evaluation through: the careful observation of participants during the study execution and the application of a follow-up questionnaire. The objective of this analysis was to identify the main barriers of reflecting upon multiple smells along the task of identifying design problems. The outcomes of this analysis helped us to better understand ways to improve support for the identification of design problems in stinky code.

By triangulating the results of both analyses, we noticed that 36.36% of the developers found more design problems when explicitly reasoning about multiple smells. We found that the understanding of complex stinky code helped to confirm the occurrence of non-trivial design problems, such as *Scattered Feature* (Garcia *et al.* 2009). Furthermore, we found that 63.63% of the developers reported much less false positives when analyzing multiple smells than when analyzing single smells. Thus, developers that considered stinky program locations, instead of isolated smelly code, could identify design problems with higher precision. However, this study also showed that developers need better support to analyze stinky program locations to reveal design problems. We observed that the analysis of stinky code may be difficult and time consuming. For instance, a prioritization and filtering approach is required so that developers do not waste time analyzing many stinky program locations not related to design problems.

Finally, to achieve our third goal, we evaluated Organic with the Communicability Evaluation Method (CEM) (de Souza *et al.* 2009). CEM is a method from the Semiotic Engineering theory, which is intended to reveal ruptures of communication when a user interacts a system, i.e., in our case when the developer interacts with the Organic tool. This method enabled us to identify issues in the Organic tool that may hinder the identification of design problems.

By conducting the communicability evaluation of Organic, we observed three major issues. First, although the tool detects stinkier program locations, it often fails to provide a concise message that facilitates the reasoning about the possible design problem (affecting the stinky program location). Second, the terms used in the tool are not adequate to certain software developers. Third, Organic uses ambiguous static symbols for representing different types of information. During the evaluation, we noted that, the aforementioned issues may often hinder the identification of design problems in stinky code.

3.2 Contextualization

This section, which is organized into two subsections, provides background information to support the understanding of this chapter. Section 3.2.1 outlines basic concepts. Section 3.2.2 brings up an illustrative example of analyzing stinky code to identify design problems.

3.2.1 Basic Concepts

Design Problem. A design problem is a characteristic in the software design that leads to negative impact on maintainability (Trifu and Marinescu 2005). Design problems affect program locations such as packages, interfaces, hierarchies, classes, and other structures that are relevant for the design of the system (Bass *et al.* 2003). Examples of design problems include *Scattered Feature* (Garcia *et al.* 2009) and *Fat Interface* (Martin and Martin 2006). The description of the eight types of design problems considered in our study is presented in Table 3.1. We opted by selecting these design problems since: (i) they are often considered as critical in the systems (Oizumi *et al.* 2016) chosen in our study, and (ii) other studies haven shown the relation between such design problems and code smells (Macia *et al.* 2012b, Oizumi *et al.* 2016, Macia *et al.* 2012, Macia *et al.* 2012a, Macia 2013).

Smelly Code. Code smell is a recurring micro structure in the source code that may indicate the manifestation of a design problem (Fowler 1999). A design problem can manifest itself in a program by affecting multiple source code locations. Each of these locations are called here *smelly code*. Thus, the developers can analyze the smelly code to identify a design problem. There are several types of code smell, which may affect a method, a class or a hierarchy.

Table 3.1: Description of design problems

Name	Description
Fat Interface	Interface of a design component that offers only a general, ambiguous entry-point that provides non-cohesive services, thereby complicating the clients' logic.
Unwanted Dependency	Dependency that violates an intended design rule.
Feature Overload	Design components that fulfill too many responsibilities (i.e., concerns).
Cyclic Dependency	Two or more design components that directly or indirectly depend on each other.
Delegating Abstraction	An abstraction that exists only for passing messages from one abstraction to another.
Scattered Feature	Multiple components that are responsible for realizing a crosscutting features.
Overused Interface	Interface that is overloaded with many clients accessing it. That is, an interface with too many clients.
Unused Abstraction	Design abstraction that is either unreachable or never used in the system.

In this chapter, we used nine types of code smell, which are: *God Class*, *Brain Method*, *Data Class*, *Dispersed Coupling*, *Feature Envy*, *Intensive Coupling*, *Refused Bequest*, *Shotgun Surgery*, and *Tradition Breaker*. These types of smell were considered in this study as they occur in the target systems used in this work. The description of each type of smell is presented in Table 3.2.

Table 3.2: Types of code smell

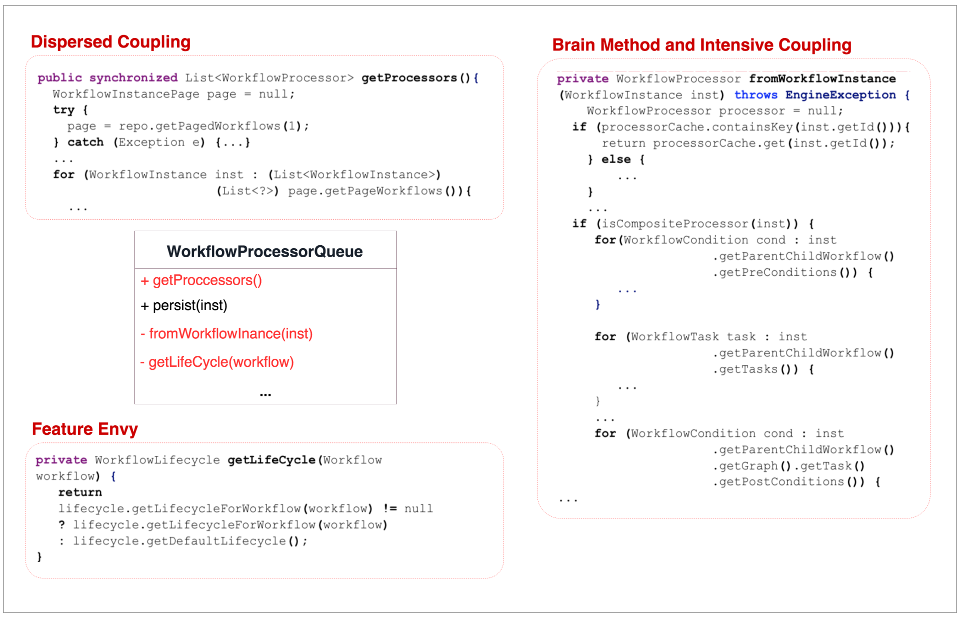
Type	Description
God Class	Long and complex class that centralizes the intelligence of the system
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Dispersed Coupling	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature Envy	Method that calls more methods of a single external class than the internal methods of its own inner class
Intensive Coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	This smell is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior
Tradition Breaker	Subclass that provides a large set of services that are unrelated to services provided by the superclass

Code Smells and Design Problems. Developers can rely on the analysis of code smells to identify design problems (Macia *et al.* 2012, Lanza and Marinescu 2006, Suryanarayana, Samarthyam and Sharma 2014). The use of code smells to identify design problems is possible because some instances of code smells manifest in the source due to the presence of a design problem. Consequently, code smells tend to co-occur in elements affected by design problems (Oizumi *et al.* 2016, Yamashita *et al.* 2015, Macia *et al.* 2012a, Oizumi *et al.* 2015), which make them indicators of design problems. Unfortunately, not all (instance of) code smells are related to a design problem (Macia *et al.* 2012).

Usually, a code smell is related to a design problem when it occurs due to the presence of design problem. For instance, consider *Scattered Feature* (Garcia *et al.* 2009), a design problem that occurs when multiple code elements implement a functionality that should have been implemented by only a few elements. Often, elements that implement the scattered functionality contains code smells such as *God Class*, *Feature Envy*, *Intensive Coupling*, *Divergent Change*, and the like. As the code elements implement a scattered functionality, these elements are likely of implementing at least two functionalities: their predominant functionality and another one, in which the predominant functionality can either the scattered one or not. Either way, the elements implement more than one functionality, which leads them to the appearance of a *God Class*. Additionally, the methods in the class have to communicate with other classes that also implement the scattered functionality. Thus, these methods can contain instances of *Feature Envy*, leading to the appearance of an *Intensive Coupling* smell. Furthermore, every chance in the functionality will impact the elements that implement it; thus, these elements will have the *Shotgun Surgery* and *Divergent Change*. In summary, these code smells could appear in the elements due to the scattered functionality, *i.e.*, due to the Scattered Feature.

Stinky Program Location. Indeed, code smells can be indicators of design problems. In fact, recent studies (Macia *et al.* 2012b, Oizumi *et al.* 2016, Abbes *et al.* 2011, Yamashita *et al.* 2015) suggest that the stinkier a program location is, the more likely it is to be affected by a design problem. Stinky code is the manifestation of multiple code smells in a program location. In this chapter, we are especially interested in stinky code indicated by *smell agglomerations* (Oizumi *et al.* 2016). A smell agglomeration is a group of interrelated code smells affecting the same program location, such as a method, a class, a hierarchy or a package (Oizumi *et al.* 2016). Thus, the agglomeration is determined in the program by the co-occurrence of two or more code smells in the same method, class, hierarchy or package (or component). For code smells that co-occur in the last three cases, we only consider they are part of an agglomeration if they are syntactically related (Oizumi *et al.* 2016). For instance, two classes can be related through structural relationships in the program, such as method calls and inheritance relationships. In this chapter, we considered four categories of agglomeration, which are presented below.

An *intra-method smell agglomeration* consists of multiple code smells that are located in a single method. The minimum number of code smells required to characterize an intra-method smell agglomeration is arbitrarily defined by the developers through a threshold. Figure 3.1 presents an example of intra-method agglomeration extracted from the Apache OODT (Object Oriented Data Technology) system. OODT is a distributed system aimed at supporting the management and integration of processes, data, and metadata (Mattmann *et al.* 2006). The agglomeration of Figure 3.1 occurs in the *fromWorkflowInstance* method, which is implemented by the *WorkflowProcessorQueue* class. This method is affected by two code smells: Brain Method and Intensive Coupling. The smelly method is the source of a Brain Method because *fromWorkflowInstance* performs several operations related to pre-conditions, tasks, and pos-conditions. All these operations make the method difficult to read and, consequently, to maintain. Moreover, this method suffers from *Intensive Coupling* because it is tightly coupled to a few classes, namely *WorkflowInstance, WorkflowProcessor, WorkflowCondition, and WorkflowTask.* These two smells together indicate the method is complicated, addresses multiple responsibilities, and is intensively coupled to a few classes in the system.



```
public synchronized List<WorkflowProcessor> getProcessors(){
  WorkflowInstancePage page = null;
  try {
    page = repo.getPagedWorkflows(1);
  } catch (Exception e) {...}
  ...
  for (WorkflowInstance inst : (List<WorkflowInstance>)
                          (List<?>) page.getPageWorkflows()){
    ...
```

```
private WorkflowLifecycle getLifeCycle(Workflow
workflow) {
   return
   lifecycle.getLifecycleForWorkflow(workflow) != null
   ? lifecycle.getLifecycleForWorkflow(workflow)
   : lifecycle.getDefaultLifecycle();
}
```

```
private WorkflowProcessor fromWorkflowInstance
(WorkflowInstance inst) throws EngineException {
    WorkflowProcessor processor = null;
  if (processorCache.containsKey(inst.getId())){
      return processorCache.get(inst.getId());
    } else {
        ...
    }
    ...
  if (isCompositeProcessor(inst)) {
     for(WorkflowCondition cond : inst
                         .getParentChildWorkflow()
                         .getPreConditions()) {
        ...
     }

     for (WorkflowTask task : inst
                         .getParentChildWorkflow()
                         .getTasks()) {
        ...
     }
     ...
     for (WorkflowCondition cond : inst
                         .getParentChildWorkflow()
                         .getGraph().getTask()
                         .getPostConditions()) {
...
```

Figure 3.1: Intra-method and intra-class agglomerations in the WorkflowProcessorQueue class

An *intra-class smell agglomeration* consists of multiple code smells affecting a single class. Thus, a class C has an agglomeration whenever the number of code smells affecting C is higher than an arbitrary threshold defined by the developer. Figure 3.1 also shows an example of intra-class agglomeration extracted from the OODT system. This agglomeration occurs in the *WorkflowProcessorQueue* class and is composed by four code smells instances: Feature Envy in the *getLifeCycle* method, Dispersed Coupling in the *getProcessors* method, Brain Method and Intensive Coupling in the *fromWorkflowInstance* method. Such a combination of smells suggests the *WorkflowProcessorQueue* class is tied to many other classes in the system, and has more responsibilities than it should.

A *hierarchical agglomeration* follows two conditions. First, all code elements have to be affected by the same type of code smells. Second, these elements have to implement the same interface or inherit from the same code element. Figure 3.2 illustrates an example of hierarchical agglomeration affecting the Apache OODT system. The Versioner class is an abstraction affected by a Fat Interface instance due to the high number of responsibilities implemented into it, which realize different features. Besides that, the Versioner class is inherited by other classes, namely *SingleFileBasicVersioner*, *BasicVersioner*, *DateTimeVersioner*, and *MetadataBasedFileVersioner*. All implementations are affected by Feature Envy, because they have too many dependencies with multiple classes of the system. Thus, all these Feature Envy instances together form an agglomeration that reifies the Fat Interface design problem affecting Versioner.

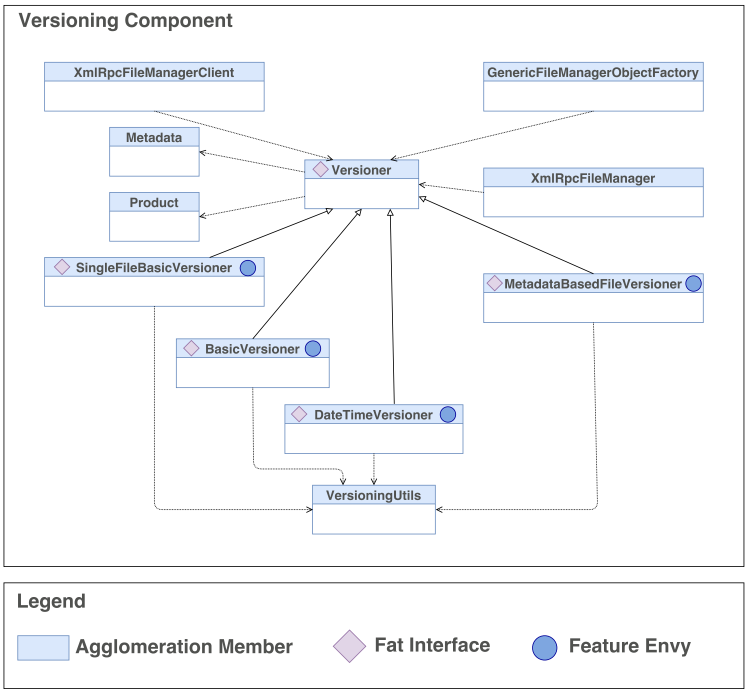


Figure 3.2: Hierarchical agglomeration under the Versioner class

An *intra-component smell agglomeration* occurs inside of a single design component. This agglomeration comprises multiple code smells affecting different code elements that are located within the same component. The minimum number of code smells required to characterize an intra-component smell agglomeration is arbitrarily defined by the developer. Note that, for characterizing this type of smell agglomeration, all code elements have to (i) be affected by the same type of code smell and (ii) be connected by method calls or type references. Section 3.2.2 presents a detailed example involving an intra-component smell agglomeration.

The concept of smell agglomeration is not limited to the categories presented above. There are other categories that we did not consider in this study. An example is the Concern-overload category (Oizumi *et al.* 2016), which is provided by the Organic tool – as described in Section 3.3. In this chapter, our focus was to analyze the identification of design problems in stinky program locations. Therefore, we considered only categories that represent common program locations, such as methods and classes.

3.2.2
Identifying Design Problem in Stinky Code

As explained in previous section, the identification of design problems can be based on code smells. For instance, let us consider the example illustrated in Figure 3.3. This figure presents some classes that belong to the *Workflow Manager* subsystems – a subsystem of the Apache OODT (Object Oriented Data Technology) system (Mattmann *et al.* 2006). It is responsible for description, execution, and monitoring of workflows. Suppose that a developer is in charge of identifying design problems in the Workflow Manager subsystem. She can rely on the analysis of code smells to spot program locations that may contain a design problem. If she is analyzing the repository package, she will notice that this package contains several code smells as indicated by a smell agglomeration. This agglomeration is formed by 4 instances of the *Feature Envy* smell. As illustrated by Figure 3.3, each of the *Feature Envy* occurrences affects a different class. In this case, 3 classes implement the *WorkflowRepository* interface. When the developer analyze these classes based on the *Feature Envy* smell, she will realize that these classes contain the smell because one of their methods is more interested in other classes than in its own hosting class. This happens because these methods are forced to implement a method that was defined in the *WorkflowRepository* interface. That is, the smells in the agglomeration are indicating that (the corresponding method in) the interface may contain a design problem. In fact, this "forced implementation" becomes a problem because these methods are implementing a feature that should not have been implemented in their hosting classes. That happens because of the fact that the *WorkflowRepository* interface processes multiple services; thus, any class that implements this interface needs to handle more services than it actually should have.

In this example, the developer knows that the code smells in the agglomeration have the same type (*Feature Envy*). Also, she knows that 3 classes affected by the code smells implement the same interface, as reified in a *hierarchical*

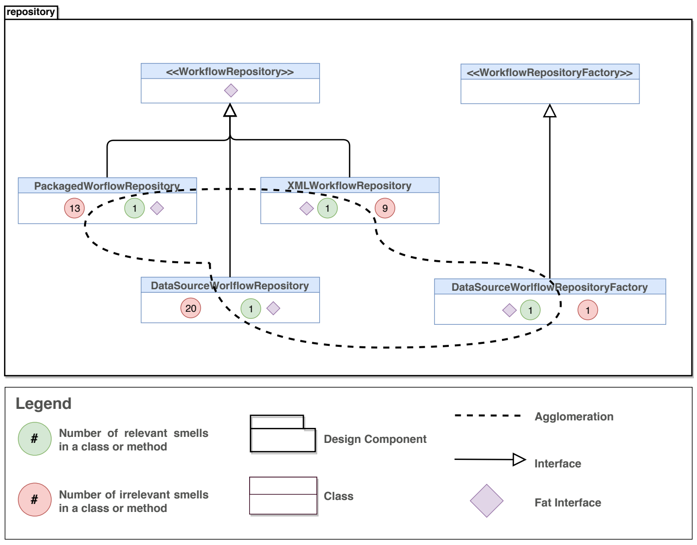

repository
<<WorkflowRepository>>
<<WorkflowRepositoryFactory>>
PackagedWorflowRepository
13
1
XMLWorkflowRepository
1
9
DataSourceWorlflowRepository
20
1
DataSourceWorlflowRepositoryFactory
1
1
Legend
Number of relevant smells in a class or method
Number of irrelevant smells in a class or method
Design Component
Class
Agglomeration
Interface
Fat Interface

Figure 3.3: Example of agglomeration in the Workflow system

agglomeration. This interface, in its turn, seems to provide non-cohesive services. Thus, the developer can infer that a design problem, called *Fat Interface*, is affecting the *WorkflowRepository* interface. On the other hand, if she did not reflect upon the code smell agglomeration, it would be harder to her to identify the same design problem. One of the reasons is the number of code smells spread over the 6 classes and 2 interfaces within the package. Although the package contains only 8 classes (Figure 3.3 only shows some of them), it has more than 50 code smells, many of which are irrelevant for the identification of a *Fat Interface*. Thus, she has to analyze many smelly code snippets in order to discard, postpone or further consider them in the identification of design problems.

Let us assume that the developer only reasons about each code smell in isolation to identify the design problem, i.e., without taking into consideration smell relationships in an agglomeration. Thus, she can choose to analyze the *DataSourceWorkflowRepository* class first because the class contains the highest number of smells in the package. Analyzing the 21 instances of code smells in the class, the developer will notice that the class has smells related to high coupling with other classes (*Intensive Coupling* and *Dispersed Coupling*), low cohesion (*Feature Envy*), and overload of responsibilities (*God Class*). However, all these smells may indicate different problems. Thus, she has to extend the analysis to other classes in order to gather more information that can potentially indicate a design problem. Unfortunately, the other classes also have different instances of code smells, and these instances may not be related to any design problem. Therefore, the developer can face difficulties to find the relevant code smells that can help him to identify a design problem. Thus, the analysis of stinky program locations, as revealed by agglomerations, seems to be a better strategy. However, there is little empirical understanding about this phenomenon. Existing studies are limited to investigating only if there is a correlation of design problems with stinky code.

3.3 Organic: A Tool for the Analysis of Stinky Code

In this section, we present the Organic tool[1]. Organic is a plug-in developed for the Eclipse IDE. The essential objective of Organic is to enable its users to identify and reason about design problems. To fulfill its role, Organic detects, and groups code smells into agglomerations of code smells. The detection of smells is performed with the conventional detection strategies proposed by Marinescu (Marinescu, 2004). Each conventional detection strategy is a heuristic that detects code elements that possibly suffer from a particular type of code smell. The heuristic of a detection strategy is based on a set of metrics and thresholds, which are combined into logical expressions (Marinescu, 2004). After the detection of code smells, Organic explores different forms of relationship between smells in order to search for smell agglomerations. The smell agglomerations are identified through information extracted from different artifacts of the analyzed software. Finally, for each agglomeration, Organic extracts information that may be helpful for the identification of design problems.

We developed and evolved Organic's features based on findings from related work (Macia *et al.* 2012b, Oizumi *et al.* 2016, Macia *et al.* 2012, Macia *et al.* 2012a, Oizumi *et al.* 2014a, Oizumi *et al.* 2017). We used their findings as a start point for defining a preliminary set of requirements for supporting the identification of design problems in stinky code. Next, we present the requirements along with the corresponding features of Organic. The requirements are made explicit in the title of each paragraph below, followed by the description of Organic features that implement the corresponding requirement.

Supporting Multiple Categories of Agglomeration. Our prior work (Oizumi *et al.* 2014a, Oizumi *et al.* 2015) provided evidence that design problems may be reified in the source code by different forms of agglomeration. Therefore, a tool for the analysis of stinky code must support multiple categories of agglomeration. Thus, to support this requirement, Organic pro-

[1]Organic. Available at http://wnoizumi.github.io/organic/.

vides five categories of agglomerations: (i) *intra-method*, (ii) *intra-class*, (iii) *intra-component*, (iv) *hierarchical*, and (v) *concern-overload*.

Before searching for agglomerations, Organic uses the source code model provided by Eclipse – through the *org.eclipse.jdt.core.dom* package – to compute metrics such as Access to Data, Number of Lines of Code, McCabe Complexity, and the like. After that, The metrics are combined into heuristics for the detection of code smells. Organic uses the heuristics defined by the conventional detection strategies of Marinescu (Marinescu, 2004). Below we present an example of detection strategy for Long Method smells:

Long Method = Lines Of Code > VERY HIGH and Cyclomatic Complexity > HIGH

Detection strategy above determines that a Long Method occurs when the method (1) has more lines of code than the number defined by a given threshold (VERY HIGH), and (2) has cyclomatic complexity higher than a given threshold (HIGH).

After detecting all code smells, Organic use different algorithms to search for different categories of agglomeration. Algorithm 1 shows a pseudo-code illustrating the algorithm used by Organic to search for intra-method agglomerations. For each method in the source code, Organic computes the number of smells. When a method has more code smells than a given threshold, Organic considers that there is an intra-method agglomeration.

```
Data: Set M of methods
Result: Set A of intra-method agglomerations
initialization;
for each method m in M do
    if numberOfSmells(m) > THRESHOLD then
        A.add(agglomerationOf(m))
    end
end
```

Algorithm 1: Pseudo-code of the algorithm to search for intra-method agglomerations

The different categories are shown by Organic in the *Agglomerations View*. This is the main view of the tool and it provides features to support the identification of design problems. Figure 3.4 shows a screenshot of the Agglomerations View. As one can observe, this view is separated in two parts: the first part is called Agglomerations, which is shown on the left side; the second part is called Details, which is shown on the right side. The Agglomerations part shows the agglomerations found in one or more projects according to their category. For instance, Figure 3.4 shows two categories of agglomerations detected in a project called *cas-pushpull*.

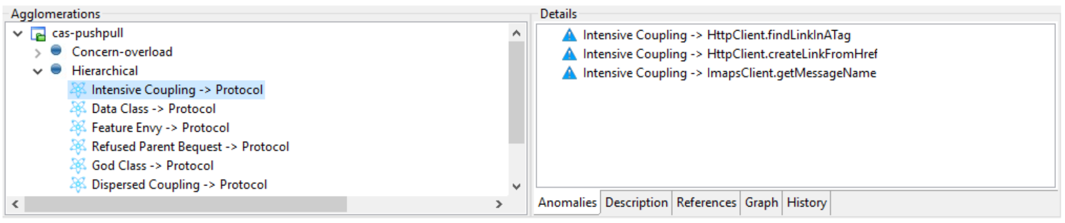

Agglomerations
cas-pushpull
Concern-overload
Hierarchical
Intensive Coupling -> Protocol
Data Class -> Protocol
Feature Envy -> Protocol
Refused Parent Bequest -> Protocol
God Class -> Protocol
Dispersed Coupling -> Protocol
Details
Intensive Coupling -> HttpClient.findLinkInATag
Intensive Coupling -> HttpClient.createLinkFromHref
Intensive Coupling -> ImapsClient.getMessageName
Anomalies
Description
References
Graph
History

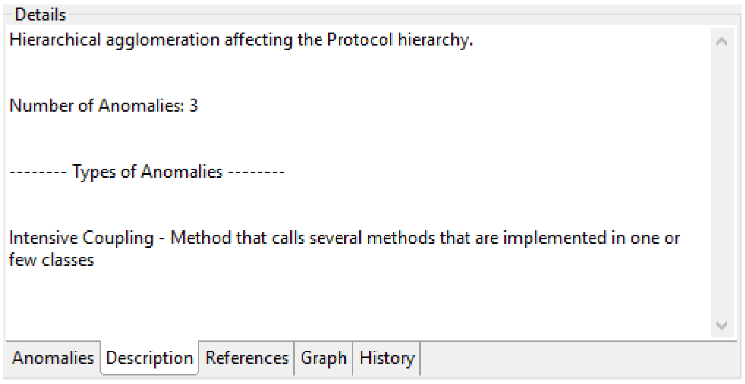
Figure 3.4: Smells that compose an agglomeration

By clicking on an agglomeration category, one more subitem level is expanded. This new level displays all agglomerations in the selected category. For example, in Figure 3.4 all the Hierarchical agglomerations that were found in the *cas-pushpull* project are displayed. Thus, developers can use these Organic features to select the category and/or the specific agglomeration they want to focus.

Providing Detailed Information about Code Smells. Many developers have little to no knowledge about the concept of code smells. In a survey conducted by Yamashita and Moonen (Yamashita and Moonen 2013), for example, only 18% of participants reported a good or strong understanding about code smells. As a result, most developers may fail short in the analysis of stink code when using a tool that do not provide enough information. Hence, to overcome this limitation, Organic provides detailed information about each agglomeration of code smells.

When the user selects an agglomeration, Organic displays all the smells that compose the agglomeration on the right side of the screen (in the Anomalies tab). One can see in Figure 3.4 that the first agglomeration in the *Hierarchical* category contains three *Intensive Coupling* smells. The second tab (Figure 3.5) presents a textual description with information about the agglomeration according to the category. As one can observe in the figure, the description of a *Intra-method* agglomeration displays information about the number of smells that compose the agglomeration with a textual description of each type of smell.

Supporting the Analysis of Surrounding Code Elements. In our previous work (Oizumi *et al.* 2015, Oizumi *et al.* 2016), we observed that a design problem may involve the surrounding code elements of an agglomeration. For instance, we found agglomerations in which one or more surrounding elements


Details

Hierarchical agglomeration affecting the Protocol hierarchy.

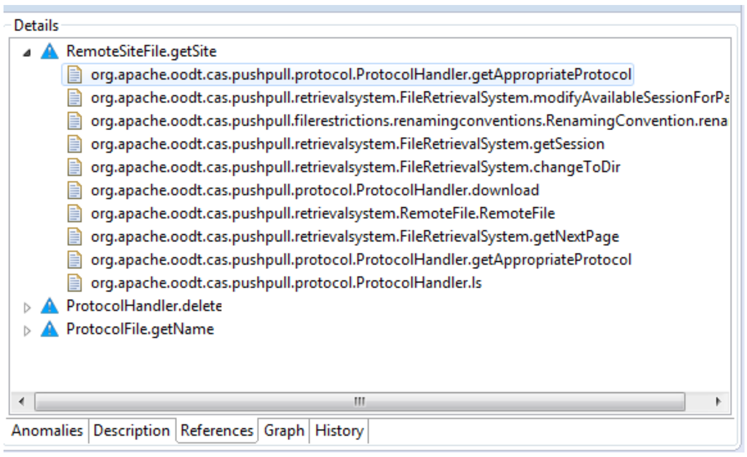
Number of Anomalies: 3

-------- Types of Anomalies --------

Intensive Coupling - Method that calls several methods that are implemented in one or few classes

Figure 3.5: Description of an agglomeration

were the main cause for the design problem manifestation. Thus, to support the analysis of surrounding code elements, Organic provides a tab called References. This tab displays all references involving smelly code elements. For instance, in Figure 3.6 the *getSite* method of the *RemoteSiteFile* class is referenced by 10 other methods. This Organic feature enables developers to reason about the external impact of an agglomeration and further help for the search of a design problem associated with the agglomeration. For example, a high number of references involving agglomerated element(s) and their surrounding elements may suggest the occurrence of a scattered functionality and/or an overly coupled component.


Details

RemoteSiteFile.getSite
- org.apache.oodt.cas.pushpull.protocol.ProtocolHandler.getAppropriateProtocol
- org.apache.oodt.cas.pushpull.retrievalsystem.FileRetrievalSystem.modifyAvailableSessionForPa
- org.apache.oodt.cas.pushpull.filerestrictions.renamingconventions.RenamingConvention.rena
- org.apache.oodt.cas.pushpull.retrievalsystem.FileRetrievalSystem.getSession
- org.apache.oodt.cas.pushpull.retrievalsystem.FileRetrievalSystem.changeToDir
- org.apache.oodt.cas.pushpull.protocol.ProtocolHandler.download
- org.apache.oodt.cas.pushpull.retrievalsystem.RemoteFile.RemoteFile
- org.apache.oodt.cas.pushpull.retrievalsystem.FileRetrievalSystem.getNextPage
- org.apache.oodt.cas.pushpull.protocol.ProtocolHandler.getAppropriateProtocol
- org.apache.oodt.cas.pushpull.protocol.ProtocolHandler.ls

ProtocolHandler.delete

ProtocolFile.getName

Figure 3.6: References of an agglomeration

Providing a Visual Representation of Stinkier Code. Based on findings from our first (mixed-method) study (Section 3.4), we also incorporated a graph-based view into Organic. This view is displayed in the fourth tab. Figure 3.7 shows an example of the graph-based visualization for a selected agglomeration. The visualization is not intended to provide a dependency graph of the agglomeration's code elements. Instead, the objective is to provide an abstract representation of an agglomeration, an overview of the composition of the agglomeration, and to help the analysis and understanding of the agglomeration. Figure 3.7 illustrates the graphic representation of an *hierarchical* agglomeration. In this graph, the blue nodes represent smelly classes, while the red arrows represent inheritance relationships. Figure 3.8 illustrates an *intra-component* agglomeration graph. In the graph, a rectangle with a red outline represents the affected component while the nodes represented within the component are the smelly classes. In the same way, the *concern-overload* agglomeration graph (Figure 3.9) also illustrates the component and smelly classes. In this graph, features (i.e., concerns) are shown by hovering over the nodes representing the classes. The *intra-class* agglomeration graph (Figure 3.10) shows a blue node that represents the agglomerated class, while the gray nodes represent the smelly methods of the class. Finally, the *intra-method* agglomeration graph (Figure 3.11) shows a blue node representing the agglomerated method and red nodes to represent the name of smells affecting the method.

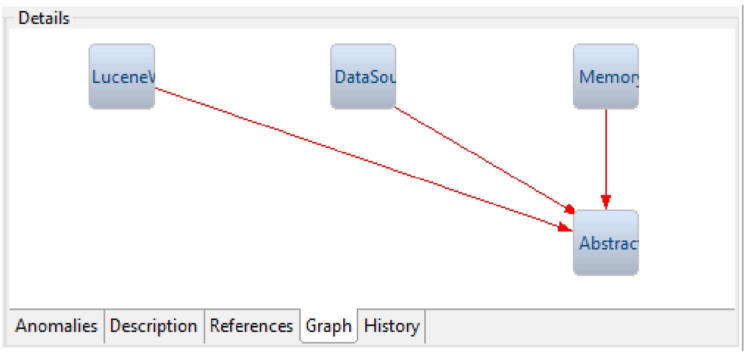
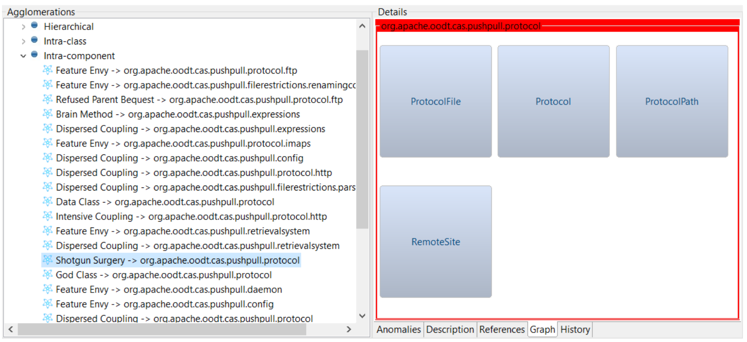


Figure 3.7: Graph representation of a hierarchical agglomeration


Agglomerations

- Hierarchical
- Intra-class
- Intra-component
  - Feature Envy -> org.apache.oodt.cas.pushpull.protocol.ftp
  - Feature Envy -> org.apache.oodt.cas.pushpull.filerestrictions.renamingcc
  - Refused Parent Bequest -> org.apache.oodt.cas.pushpull.protocol.ftp
  - Brain Method -> org.apache.oodt.cas.pushpull.expressions
  - Dispersed Coupling -> org.apache.oodt.cas.pushpull.expressions
  - Feature Envy -> org.apache.oodt.cas.pushpull.protocol.imaps
  - Dispersed Coupling -> org.apache.oodt.cas.pushpull.config
  - Dispersed Coupling -> org.apache.oodt.cas.pushpull.protocol.http
  - Dispersed Coupling -> org.apache.oodt.cas.pushpull.filerestrictions.pars
  - Data Class -> org.apache.oodt.cas.pushpull.protocol
  - Intensive Coupling -> org.apache.oodt.cas.pushpull.protocol.http
  - Feature Envy -> org.apache.oodt.cas.pushpull.retrievalsystem
  - Dispersed Coupling -> org.apache.oodt.cas.pushpull.retrievalsystem
  - Shotgun Surgery -> org.apache.oodt.cas.pushpull.protocol
  - God Class -> org.apache.oodt.cas.pushpull.protocol
  - Feature Envy -> org.apache.oodt.cas.pushpull.daemon
  - Feature Envy -> org.apache.oodt.cas.pushpull.config
  - Dispersed Coupling -> org.apache.oodt.cas.pushpull.protocol

Figure 3.8: Graph representation of an intra-component agglomeration

Providing Historical Information. During the analysis of an agglomeration, developers may benefit from information about the evolution of an agglomeration across the source code history (Oizumi *et al.* 2014b). Therefore, the fifth tab of Organic displays historical information about the selected agglomeration. By historical information we mean information about the agglomeration in previous versions of the software. Thus, this historical information

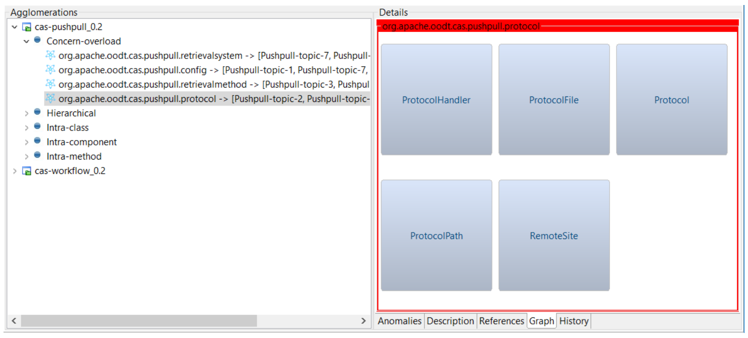

Agglomerations
cas-pushpull_0.2
Concern-overload
org.apache.oodt.cas.pushpull.retrievalsystem -> [Pushpull-topic-7, Pushpull-
org.apache.oodt.cas.pushpull.config -> [Pushpull-topic-1, Pushpull-topic-7,
org.apache.oodt.cas.pushpull.retrievalmethod -> [Pushpull-topic-3, Pushpul
org.apache.oodt.cas.pushpull.protocol -> [Pushpull-topic-2, Pushpull-topic-
Hierarchical
Intra-class
Intra-component
Intra-method
cas-workflow_0.2
Details
org.apache.oodt.cas.pushpull.protocol
ProtocolHandler
ProtocolFile
Protocol
ProtocolPath
RemoteSite
Anomalies
Description
References
Graph
History

Figure 3.9: Graph representation of a concern-overload agglomeration

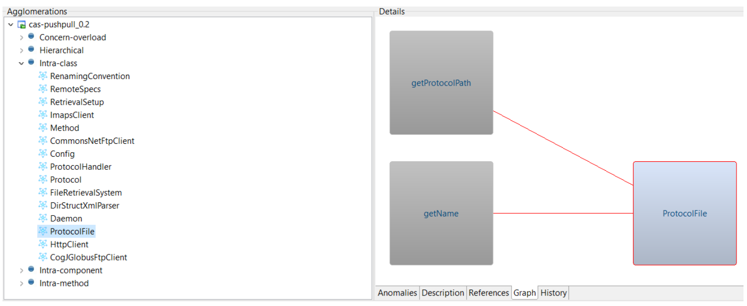


Figure 3.10: Graph representation of an intra-class agglomeration

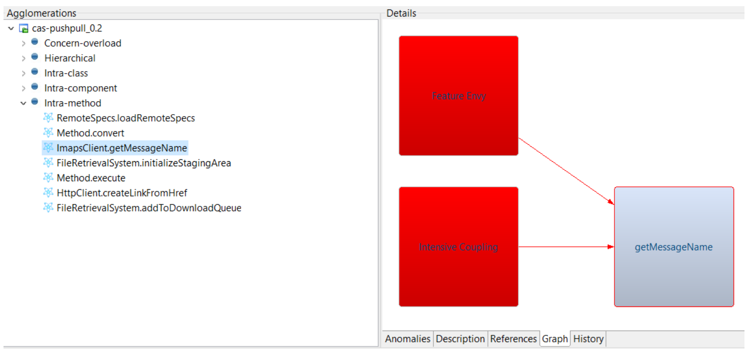
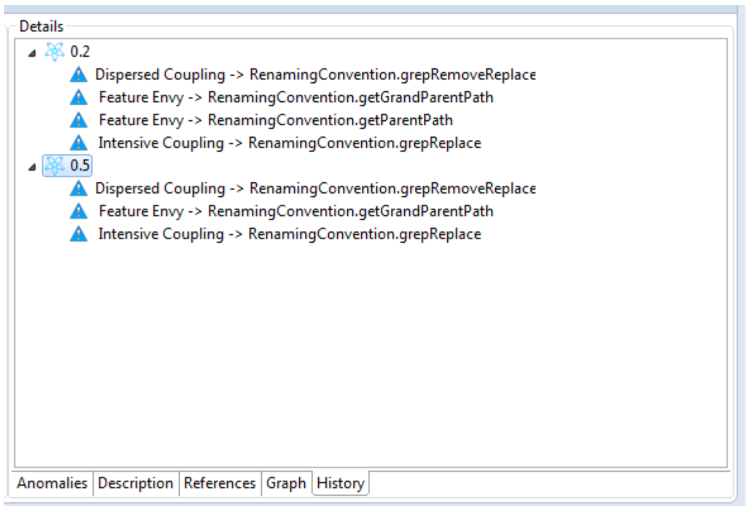


Figure 3.11: Graph representation of an intra-method agglomeration

is organized by versions. For example, in the Figure 3.12, information about the agglomeration is displayed in 2 previous versions: “0.2” and “0.5”. Each version shows the code smells that were members of the agglomeration.

The objective of this tab is to assist the analysis of the evolution of each agglomeration throughout the different versions of the system. This tab shows the history of code smells that progressively composed the agglomeration along the versions of the software. As one can be observe in Figure 3.12, in version 0.2, the agglomeration was composed by four smells (1 *Dispersed Coupling*, 2 *Feature Envy*, and 1 *Intensive Coupling*). On the other hand, the same agglomeration was composed of three smells (1 *Dispersed Coupling*, 1 *Feature Envy*, and 1 *Intensive Coupling*) in version 0.5. Using this feature, developers



Details
- 0.2
  - Dispersed Coupling -> RenamingConvention.grepRemoveReplace
  - Feature Envy -> RenamingConvention.getGrandParentPath
  - Feature Envy -> RenamingConvention.getParentPath
  - Intensive Coupling -> RenamingConvention.grepReplace
- 0.5
  - Dispersed Coupling -> RenamingConvention.grepRemoveReplace
  - Feature Envy -> RenamingConvention.getGrandParentPath
  - Intensive Coupling -> RenamingConvention.grepReplace

Figure 3.12: Historical information of an agglomeration

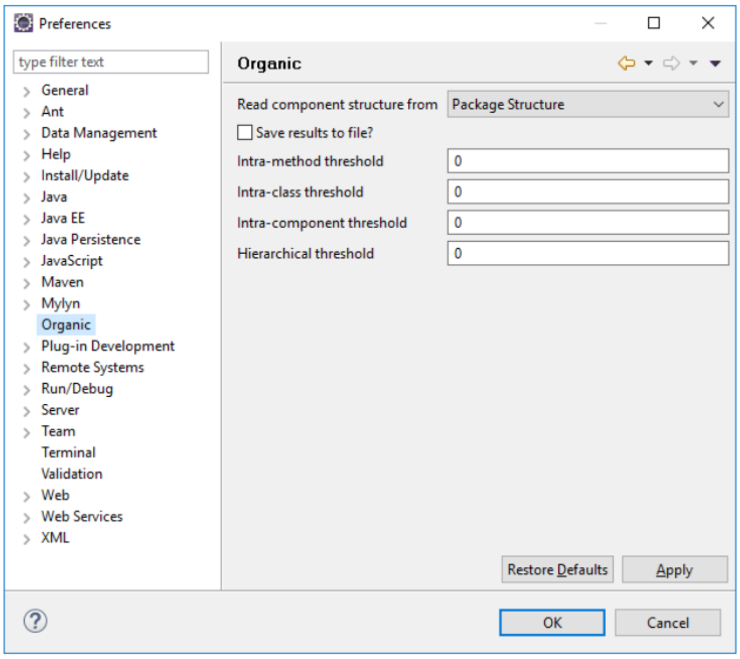
can identify agglomerations that are growing or shrinking along the system's evolution. The analysis of this phenomenon can help developers identify certain design problems. For example, a developer can check: (i) if the number of smelly clients of a specific interface (all taking part in the agglomeration) is growing (or not) along the project history, and (ii) if the agglomeration started from a smell affecting an interface in the program. These observations will help the developer to confirm if the design problem is located in that interface.

Allowing Flexible Thresholds. Similarly to conventional detection strategies for code smells (Marinescu, 2004), the detection of agglomerations also requires flexible thresholds. Therefore, to satisfy this requirement, Organic has a configuration screen (Figure 3.13) that can be accessed through the *Window* → *Preferences* menu. The purpose of this screen is to allow users to define the threshold for each agglomeration category. The threshold defines the minimum number of code smell that should be in a program location before being considered as an agglomeration. For example, if we configure the intra-method category with threshold 2, Organic will only find agglomerations in methods that contain 3 or more code smells.

3.4 Study I: Quasi-Experiment

This work is intended to address three main goals, which are: (1) provide proper support for the analysis of stinkier code, (2) assess to what extent developers are able to identify design problems in stinkier code, and (3) identify tool issues that may hinder the identification of design problems.

In the previous section, we proposed the Organic tool, attending to our first



Organic

Read component structure from: Package Structure
Save results to file? ☐
Intra-method threshold: 0
Intra-class threshold: 0
Intra-component threshold: 0
Hierarchical threshold: 0

Figure 3.13: Configuration of the tool

goal. Section Section 3.5 presents a study that addresses our third goal. Thus, aiming at achieving our second goal, in this section we present a *quasi-experiment* with professional software developers. *Quasi-experiment* is an empirical interventional study in which the subjects are not randomly assigned to certain conditions (Shadish, Cook and Campbell 2001). In this study, we investigated whether the use of smell agglomerations improves the precision of developers in identifying design problems.

3.4.1 Study Design

A previous study suggests that code smell agglomerations are related to occurrences of design problems (Oizumi *et al.* 2016). This study only analyzed the correlation between agglomerated smelly elements and code elements affected by design problems. However, such study did not show evidence that developers indeed identify design problems when exploring information about agglomerations. Thus, there is a need to investigate whether developers can, by themselves, identify design problems when exploring smell agglomerations. In order to address this matter, we defined two research questions. The first one is presented as follow:

RQ1. Does the use of agglomerations improve the precision of developers in identifying design problems?

Research question RQ1 allows us to analyze whether code smell agglomerations help developers to identify design problems. To answer this question, we conducted a *quasi-experiment* with 11 professional developers. In this *quasi-experiment*, we measured the precision of developers using agglomerations to identify design problems. Precision in our context is measured based on the percentage of true positives indicated by the developers – i.e., the percentage of correctly identified design problems (against a ground truth, as explained later). We have used precision since it is an important aspect of the identification task.

Through the precise identification of design problems, developers are able to optimize their work by solving problems that really impact design. On the other hand, the lack of precision would lead software development teams to spend time and budget with irrelevant refactoring tasks. Refactoring is a transformation used for improving the structural quality of a system while preserving its observable behavior (Fowler 1999). In companies adept of code review practices (McIntosh *et al.* 2014), the lack of precision in identifying design problems can lead developers to waste time on refactorings that do not contribute to improving software maintainability, or even refactorings that are harmful to the software design (Cedrim *et al.* 2017). The precise identification of design problems is also important in open source projects. For instance, the contributions of eventual collaborators are often rejected by core developers due to the presence of design problems (Oliveira, Valente and Terra 2016). Therefore, in this case, lack of precision could lead core developers to reject relevant contributions due to "false design problems".

Someone could wonder why we have not measured recall. Although we agree to the relevance of measuring recall, we did not measure it. The reason is because the analyzed systems have a high number of design problems unfeasible to be identified during the *quasi-experiment*, which should not last (in total) for more than 90 minutes. Therefore, the consideration of recall would not be feasible in a *quasi-experiment* as the developers have limited time to search for some design problems only. Together with the system's original developers, we created a ground truth of design problems with more than 150 instances of design problems. Hence, it would be impracticable for participants to find all the design problems in the system due to the time constraints in the study

(45 minutes). Consequently, they were expected to reach a low recall value. Therefore, we focused on the precision.

In order to measure whether precision improved or not, we compared the participants using agglomerations with a control group. The control group comprises of participants identifying design problems with a flat list of smells, i.e., code smells presented individually without showing their connection with other smells in the program. In the comparison, we used a ground truth to confirm or refute each design problem indicated by participants. Then, we compared the number of false positives and true positives produced with the code smell agglomerations against the number of false and true positives produced by the control group. In the context of this study, a false positive occurs when a participant reports a design problem that is not confirmed by our ground truth analysis. On the other hand, a true positive occurs when the design problem is confirmed during the ground truth analysis.

Someone could assume that developers would often benefit from the use of agglomerations in their quest for design problems. However, we do not have evidence about such benefit. Hence, we need to address RQ1 to verify if developers can correctly identify more design problems using smell agglomerations. Regardless of RQ1 results, another question involves the understanding of how to better support developers in exploring smell agglomerations. The success of developers on identifying design problems through agglomerations may strongly depend on additional support for this task. Even though a previous study (Oizumi *et al.* 2016) has shown the strong relation between design problems and code smells within an agglomeration, we do not know whether and how the identification of design problems with agglomerations can be improved with additional support. The following question addresses this matter:

RQ2. How can the identification of design problems with code smell agglomerations be improved?

We conducted a qualitative analysis to address RQ2. This analysis was based on the observation of participants during the *quasi-experiment* and a follow-up questionnaire. As reported in Section 3.4.2, the combination of quantitative and qualitative analyses helped us to draw more well grounded conclusions. Thus, RQ1 can inform us if developers become (or not) more precise on identifying design problems when they use agglomeration, while RQ2 can provide a complementary perspective to explain why developers succedded or struggled to precisely identify design problems. For instance, RQ2 can reveal the benefits and barriers associated with the use of smell agglomerations.

3.4.1.1
Experimental Procedure

We had to define a set of requirements in order to answer our research questions. Thus, we opted for conducting a *quasi-experiment* (Shadish, Cook and Campbell 2001). A *quasi-experiment* is an empirical study in which the units or groups are not assigned to conditions randomly. This allowed us to assign each participant to different treatments during the experimental steps. The experimental procedure was conducted individually with each participant. They had to perform the *quasi-experiment* in two steps with four tasks in each one. Both steps comprise the same set of tasks the only difference between them was regarding the treatment, *i.e.*, usage of agglomerations or non-agglomerated smells.

As explained before, we compare developers using agglomeration with a control group using non-agglomerated smells. Thus, we divided the participants into two groups. The first group would identify design problems using agglomerations in the first step. After that, they would identify design problems using non-agglomerated smells in the second step. The second group of participants would make the identification inversely: using the non-agglomerated smells in the first step and, then, using the agglomerations in the second step. Thus, in each step, we have two groups of participants: a group using agglomerations and a control group.

As each participant identifies design problems twice (first and second step), we had to select two software projects. Thus, each participant could identify design problems using a different project in both steps. Another reason for providing two software projects is to avoid bias with the learning curve. For example, supposing that the participant uses the same project in both steps. She could find more problems in the second step than in the first step. That could happen because she can identify in the second step the same problems that she identified in the first step, plus other design problems identified only in the second step. This increase in the number of design problems found in the second step would not be due to the use of agglomerations, but rather due to the knowledge acquired by the participant.

There are four possible combinations with the participants based on the distribution between steps and software projects. Therefore, all participants were divided into four mutually exclusive arranges. Table 3.3 presents the cross design for the four arranges. The agglomeration group represents the group of participants that identified design problems using the agglomerations, and the control group comprises the participants that identified design problems using the list of non-agglomerated code smells.

Table 3.3: Combinations of groups, projects and steps

	Step 1		**Step 2**	
Arrange	**Group**	**Project**	**Group**	**Project**
1	Agglomeration	Project 1	Control	Project 2
2	Agglomeration	Project 2	Control	Project 1
3	Control	Project 1	Agglomeration	Project 2
4	Control	Project 2	Agglomeration	Project 1

The study was composed by a set of six activities distributed into three phases, as represented in Figure 3.14 described as follows.

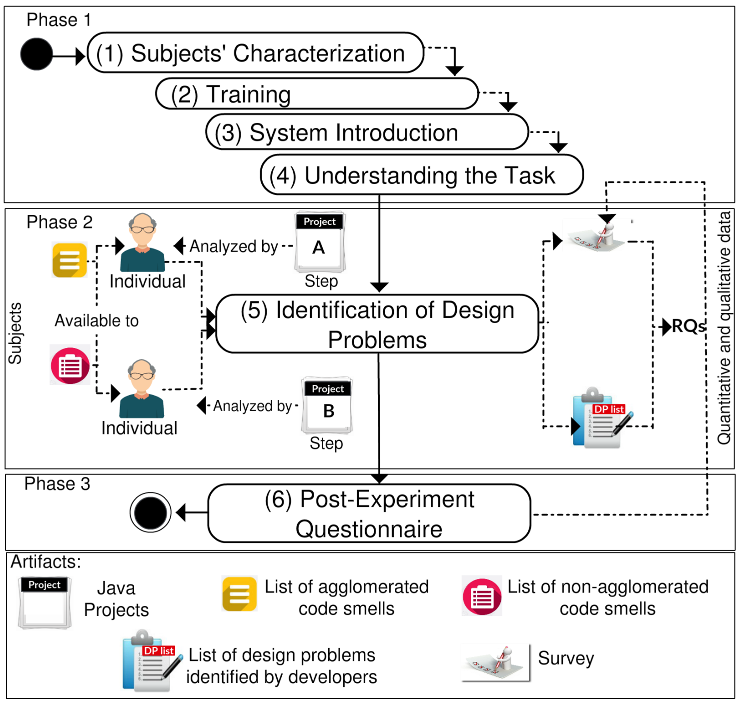

Phase 1
(1) Subjects' Characterization
(2) Training
(3) System Introduction
(4) Understanding the Task
Phase 2
Subjects
Individual
Analyzed by
Project
A
Step
Available to
(5) Identification of Design Problems
Individual
Analyzed by
Project
B
Step
DP list
RQs
Quantitative and qualitative data
Phase 3
(6) Post-Experiment Questionnaire
Artifacts:
Project
Java Projects
List of agglomerated code smells
List of non-agglomerated code smells
DP list
List of design problems identified by developers
Survey

Figure 3.14: The experimental design

Activity 1: Apply the questionnaire for subjects' characterization. The subjects' characterization questionnaire is composed of questions to characterize each participant, including academic degree, professional experience with Java programming, background on code smells, and Eclipse IDE.

Activity 2: Training Session. After defining the order of execution of each step, the next step was to provide a training session for the participants. The main objective of the training session was to level the participant at the same background required to understand and properly execute the experimental tasks. Thus, they received training about basic concepts and terminologies. This training was given only once for each participant before the first steps of the *quasi-experiment*. The training consisted of a 15-minute presentation that covered the following topics: software design, code smells, and design problems. The training session took approximately 15 minutes, and the participants could make any question throughout it.

After the training, subjects received some artifacts that could be used during the study. They received a list with a brief description of the types of design problems presented in the training session. They also received a list with the description of basic principles of object-oriented programming and software design. They received a document containing: (i) a brief description of both project systems, and (ii) a very high-level description of their design blueprint. We gave these documents because when they have to conduct perfective maintenance tasks, they need to have some minimal information about the systems to be maintained. The design blueprint represented the high-level design in the view of the project managers, but it was not detailed enough to support the identification of design problems. As it often occurs in practice, the analysis of the source code is inevitably required to identify a design problem.

Activity 3: System Introduction. We asked the participants to read the document containing the description of the project in which they would identify design problems. They had 20 minutes to read the description and the design blueprint of the system. Thus, they could start the identification with a certain level of familiarity with the software project.

Activity 4: Understanding the Task. In this activity, we explained how the participant could use the Organic tool to collect either the list of agglomerations or the list of (non-agglomerated) code smells. As the Organic tool was developed as an Eclipse plug-in, we explained each one of the sections displayed in the Eclipse IDE and that was related to the Organic tool. This activity lasted approximately 10 minutes.

Activity 5: Identification of Design Problems. In this activity, the participant had 45 minutes to identify design problems in the project. We emphasized to the participant the importance of achieving the key goal of finding design problems. For each identified design problem, the participant was asked to provide the following information: (i) short description of the problem, (ii) possible consequences caused by the problem, (iii) classes, methods or packages realizing the design problem in the source code, and (iv) the category(s) of agglomerations – described in Section 3.2.1 – that helped him to identify the design problems. If the participant was identifying design problems as part of the control group, she needed to provide almost the same information; the difference was that instead of providing the agglomeration (and its category), she needed to provide the code smells that she used to identify the design problem. For conducting this task, participants were instructed to use only the information provided by Organic in the current phase. This means that, neither the control group had access to the list of agglomerations, nor the agglomeration group had access to the list of non-agglomerated smells. This was guaranteed by providing different versions of Organic for each group – that is, one version for agglomerations and another version for non-agglomerated code smells. Nevertheless, both the project source code and the information provided by Organic (agglomerated or non-agglomerated smells) could be freely explored and analyzed during the design problem identification.

Activity 6: Follow-up Questionnaire. In this activity, the participant received a feedback form. This form provides a list of questions, which enables the participant to expose her opinion on the identification of design problems. More details about this activity are provided in Section 3.4.1.4.

After the sixth activity had been completed, we asked the same participant to repeat all tasks in the second phase.

3.4.1.2
Software Projects and Participant Selection

In order to conduct the *quasi-experiment* as explained in the previous section, we selected two software systems in which developers had to identify design problems. We selected two programs that represent components of the Apache OODT project (Mattmann *et al.* 2006): *Push Pull* and *Workflow Manager*. We selected subsystems of the OODT project since it is a large heterogeneous system; then, we could choose subsystems based on their diversity. Also, the Apache OODT project has a well-defined set of design problems previously identified by developers who actually implemented the systems (Oizumi *et al.* 2016); thus, avoiding the introduction of false positive design problems in the ground truth. In addition, the OODT project was developed for NASA, used in other studies (Macia *et al.* 2012a, Macia *et al.* 2012, Macia *et al.* 2012b, Macia 2013, Oizumi *et al.* 2016) and with a global community involved in its development. Table 3.4 presents the characteristics of each project. The columns of this table are organized as follows. The second column shows the project size in Source Lines of Code (SLOC), the third column

Table 3.4: Characteristics of software projects

Project	**Size (SLOC)**	**# Classes**	**# Agglomerations**
Pushpull	11213	133	49
Workflow	18505	150	111

presents the number of classes, and the fourth column contains the number of agglomerations in the project. A brief description of the project systems is presented as follow:

- Push Pull: it is the OODT component responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area.
- Workflow Manager: it is a component that is part of the OODT client-server system. It is responsible for describing, executing, and monitoring workflows.

After choosing the projects, our next step was to recruit developers for the study. Thus, we sent a characterization questionnaire for a group of developers of our network. Their answers were analyzed to determine which of them were eligible to participate in the study based on the following requirements:

- Four years or more of experience with software development and maintenance. We have chosen four years because this is the average time used by multiple companies to classify a developer as experienced.
- No previous knowledge about Push Pull and Workflow Manager.
- At least basic knowledge about code smells.
- At least intermediary knowledge on Java programming and Eclipse IDE.

Table 3.5: Knowledge classification

Classification	**Description**
None	I have never heard about it
Minimum	I have heard about it, but I do not use it
Basic	I have a general understanding, but almost never use it
Intermediary	I have a good understanding, and use basic features sometimes
Advanced	I have a deep understanding, and often use advanced features
Expert	I am a specialist in this topic, and I use many features almost every day

We defined the knowledge in each topic based on a scale composed of six levels: *none*, *minimum*, *basic*, *intermediary*, *advanced*, and *expert*. Table 3.5

presents the description of such classification. We included in the questionnaire a description of each level, allowing the subjects to have a similar interpretation of the answers. Table 3.6 summarizes the characteristics of each selected developer.

Table 3.6: Characterization of the participants

ID	Experience in years	Education Level	Knowledge		
			Java	Code Smells	Eclipse
P1	5	PhD	Advanced	Advanced	Advanced
P2	6	Graduate	Advanced	Basic	Advanced
P3	8	Master	Advanced	Intermediary	Advanced
P4	4	Graduate	Intermediary	Basic	Basic
P5	5	Master	Advanced	Intermediary	Intermediary
P6	5	Graduate	Intermediary	Intermediary	Intermediary
P7	12	Graduate	Expert	Advanced	Expert
P8	5	Graduate	Advanced	Advanced	Advanced
P9	10	Graduate	Intermediary	Intermediary	Intermediary
P10	4	PhD	Advanced	Intermediary	Advanced
P11	5	PhD	Advanced	Intermediary	Advanced

Table 3.7: Precision

ID	Agglomeration Group			Control Group		
	TP	FP	Precision	TP	FP	Precision
1	2	1	66.67%	1	1	50%
2	0	3	0%	1	4	20%
3	3	2	60%	1	4	20%
4	2	0	100%	1	3	25%
5	4	0	100%	3	1	75%
6	1	0	100%	1	0	100%
7	1	1	50%	1	1	50%
8	3	0	100%	3	0	100%
9	0	1	0%	0	6	0%
10	0	0	-	1	1	50%
11	0	1	0%	0	0	-
All	**16**	**9**	**64%**	**13**	**21**	**38.24%**

3.4.1.3
Quantitative Analysis Procedure

In order to answer research question RQ1, we asked the study participants to analyze two systems with the aim of identifying design problems as described above. For each system, we analyzed the precision of participants regarding the identification of design problems. The precision of participants was measured based on *true positives* (TP) and *false positives* (FP). In this context, a true positive is a candidate of design problem, as indicated by the participant, that was confirmed by a ground truth analysis. On the other hand, a false positive is a candidate of design problem that was not confirmed in the ground truth analysis. Thus, the precision is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP} \tag{3-1}$$

We had to validate the identified design problems as true positive or false positive for each one of the analyzed systems. However, we could not argue that a design problem was correct or not since we were not involved with the design of each system. Thus, we relied on the knowledge of the systems' original designers and developers to help us in validating the design problems. We certified they were the people who had the deepest knowledge of the design of the investigated projects. We highlight these designers and developers were not subjects of this study.

We performed two steps to incrementally develop the ground truth. First, we asked original OODT designers and developers to provide us a list of design problems affecting the systems. They listed the problems and explained the relevance of each one through a questionnaire. They also described which code elements were contributing to the realization of each design problem. Second, we identified some design problems using a suite of design recovery tools (Garcia *et al.* 2013). We asked developers of the systems to validate and combine our additional design problems with their list. The procedure for the additional identification was the following: (i) an initial list of design problems was identified using a method presented by Macia and colleagues (Macia *et al.* 2012a), (ii) the developers had to confirm, refute or expand the list, (iii) the developers provided a brief explanation of the relevance of each design problem, and (iv) when we suspected there was still inaccuracies in the list of design problems, we discussed with them. In the end, we had the ground truth of design problems validated by the original designers and developers.

3.4.1.4
Qualitative Analysis Procedure

Our first research question aim to investigate the precision of developers in the identification of design problems with agglomerations. Answer for such question will indicate whether developers benefit or not of using agglomerations. However, answering this questions is not enough for revealing the reasons why agglomerations may benefit developers. Moreover, we will not know how to improve the use of agglomerations to identify design problems. Therefore, we conducted a qualitative analysis to investigate what should be improved from the perspective of professional software developers. Besides identifying possible improvements, this analysis also helped us to understand what are the main strengths of exploring agglomerations for design problem identification.

As described in Section 3.4.1.1, we asked the participants to provide us feedback about the identification of design problems. They answered a follow-up questionnaire, and we use their answers to conduct a qualitative analysis. The objective of the questionnaire was to gather participant's opinion regarding (i) the (dis)advantages of using the agglomerations or code smells to identify design problems, (ii) whether the provided information could be easily understood, (iii) which types of information were fundamental to identify design problems, (iv) what she believes that should be done to improve the identification of design problems, (v) what she thought about the use of the code smells for the identification of design problems, (vi) how the visualization mechanism provided by the Organic tool affected her performance, and (vii) which types of code smell and categories of agglomeration were the most useful for identifying design problems. The results of this questionnaire helped us to answer research question RQ2.

By applying the questionnaire, we were able to gather the opinion of developers regarding the use of code smell agglomerations. However, as reported by Easterbrook and colleagues (Easterbrook *et al.* 2008), what is reported in the questionnaire may not be what actually happens in practice. Therefore, to obtain more reliable results, we also observed the participants of our study during the identification of design problems. This observation was performed during the study and also in analyzes after it, through video and audio recorded during the *quasi-experiement*. This analysis allowed us to look at code smell agglomerations from the standpoint of professional software developers. It is important to note that the observation of participants during the *quasi-experiment* does not replace nor invalidate the questionnaire responses. In fact, the combination of observations and responses helped us to obtain a deeper understanding and interpretation on the results observed in the study.

3.4.2
Results and Analysis

The results of this study are organized in two sub-sections. Section 3.4.2.1 presents the results of our quantitative analysis regarding research question RQ1. Section 3.4.2.2 provides the results of our qualitative analysis to answer research question RQ2.

3.4.2.1
Do Agglomerations Improve Precision?

As described in Section 3.4.1.3, we conducted a quantitative analysis to answer our first research question: *Does the use of agglomerations improve the precision of developers in identifying design problems?*. Table 3.7 presents the precision results for each participant (rows). The first column (*ID*) shows the identification number of each participant. The second column (*Agglomeration Group*) presents the true positives (*TP*), false positives (*FP*), and precision for the participants when they were provided with agglomerations to identify design problems. Similarly, the third column (*Control Group*) presents the true positives (*TP*), false positives (*FP*), and precision for the participants in the control group, i.e., when they were provided with non-agglomerated smells.

Agglomeration led to a slight increase of true positives. We can see in Table 3.7 that the developers identified a few more design problems (TPs) when they were in the agglomeration group (16 TP design problems) than when they were in the control group (13 TP design problems). As far as the per-subject analysis is concerned, 4 developers (light gray rows) identified more true positives when they used agglomerations than when they used the list of code smells in the control group. The use of agglomerations outperformed the use of smells in these 4 cases. On the other hand, 2 participants (2, 10) did not identify any true positive using the agglomerations, but they identified a true positive each in the control group. The other participants (6, 7, 8, 9 and 11) identified the same number of true positives (5 TP design problems) regardless the group.

Upon qualitative analysis, we were able to reveal the main reason why the 4 developers in the light gray rows identified more true positive design problems in the agglomeration group than in the control group. As illustrated in the example in the Figure 3.3 (Section 3.2.2), these 4 participants systematically used each agglomeration's smell as an indicator of the presence of a design problem. They analyzed each one of the code smells as a complementary symptom of the presence of a design problem, which gave them increasing confidence to confirm the occurrence of the design problem. Surprisingly, we noticed the same behavior for the participant 8 even when she was in the control group. She was capable of agglomerating the code smells on her own, starting from the individual smells given in the flat list. Then, she used such agglomerations to identify design problems in the control group. This may be the reason why she reached a precision value of 100% in both groups.

Agglomerations help developers to avoid false positives. In general, developers identified less false positives when they used agglomerations (9 FP design problems) than when they used the list of code smells (21 FP design problems). As presented in our qualitative analysis (Section 3.4.2.2), with the exception of participant 11, who analyzed several irrelevant agglomerations – i.e., agglomerations that do not reveal any design problem – all others identified either less or equal number of false positives when they were in the agglomeration group than when they were in the control group. When we analyze the control group, we can notice that more than half of the identified design problems are false positives (61,76%) while the agglomeration group identified only 36% of false positives.

After observing how developers identify design problems in the control group, we noticed that they did not go further with the analysis of the elements. Usually, a developer needs to analyze other classes in order to gather more information that can potentially indicate a design problem as discussed in Section 3.2.2. When the participants used the agglomerations, they analyzed multiple elements because they analyzed each code smell within the agglomeration even when the smells were in different elements. This behavior did not happen when participants were in the control group. In most of the cases, the participants in the control group analyzed only one code smell, which increased the likelihood of reporting false positives. Then, they reported a design problem in the class due to the presence of the code smell. However, some code smells are not related to any design problem; thus, the developer can report a false positive if she mistakenly considers a code smell that is not related to a design problem. That explains why developers in the control group found so many false positives. As developers tend to look at all agglomeration' smells before reporting a design problem, the likelihood of reporting a false positive decreases, even when there is a code smell that is a false positive by itself.

Agglomerations improve the precision. Even though we cannot claim a statistical significance in our results due to the sample size of this study, we can notice that developers achieve a higher precision (64%) when they use agglomerations than when they use code smells (38,24%). Therefore, this result suggests that agglomerations may improve the precision of developers in identifying design problems, answering our first research question. However, someone could expect that *all* developers using agglomerations would significantly outperform the control group. As a matter of fact, we noticed some factors that explain, at least partially, why developers did not find much more design problems when they were in the agglomeration group than when they were in the control group. These factors are presented next, and they are useful to discover improvements for the identification of design problem with the analysis of stinky program locations.

3.4.2.2
How to Improve Design Problem Identification?

This section presents the answer for our second research question: *How can the identification of design problems with code smell agglomerations be improved?* We conducted a qualitative analysis to answer this question. As described in Section 3.4.1.4, this analysis was based on the observation of participants during the identification of design problems as well as the analysis of responses to our follow-up questionnaire.

Where to start from? As discussed in the previous section, the participants identified few more true positives using agglomerations. Someone could expect that *all* developers using agglomerations would significantly outperform the control group. However, we observed that participants spent much more time analyzing the agglomerations than analyzing the smells in the control group. That happened because they analyzed each code smell in the stinky program location as previously explained in Section 3.4.2.1. Furthermore, sometimes the participants analyzed agglomerations that were not related to any design problem, which is a key factor that explains the almost same number of true positives between both groups.

Unfortunately, almost all the participants analyzed irrelevant agglomerations. Participants 6, 9, 10, and 11 were the ones that suffered the most from the analysis of irrelevant agglomerations. Since these four participants faced this issue, they suggested in our follow-up questionnaire that the Organic tool should provide means to prioritize and filter (or, at least, rank) potentially relevant agglomerations. This feature would help to further reduce the time spent with the analysis of irrelevant stinky code. Thus, this issue also helps us to explain why they fell short in identifying additional design problems through the analysis of agglomerations.

Need for prioritizing and filtering agglomerations. The aforementioned need for agglomeration prioritization and filtering shows that the time and effort required to identify design problems is a key factor for developers; thus, prioritization should be taken into consideration. Such a need is reinforced by recent studies which explored the prioritization of code smells (Arcoverde *et* *al.* 2013, Vidal, Marcos and Díaz-Pace 2016, Vidal *et al.* 2016).

Based on our qualitative analysis, we noticed that existing criteria for prioritization should select agglomerations that are cohesive. We consider an agglomeration to be cohesive whenever all its code smells are related to the same design problem. If there is one code smell that is not related to the design problem: (i) in the best case, the developer will spend time analyzing a code smell that is useless to identify the design problem, or (ii) in the worst case, such smell may direct the developer away from the design problem. This fact suggests that developers need accurate approaches to find cohesive agglomerations and to discard the less cohesive ones. However, prioritization algorithms based on existing criteria are unable to do this as far we are concerned. Consequently, the prioritization of stinky program locations still poses as a challenging research topic. Therefore, after this study, we decided to not incorporate existing prioritization criteria into Organic. Before including any prioritization feature into Organic, we intend to propose and evaluate improvements for the existing prioritization criteria.

Stinky code analysis is challenging. Besides the prioritization issue, participants also suffered to analyze large agglomerations. As reported in Section 3.4.2.1, this problem was even worse for agglomerations affecting larger program scopes, i.e., agglomerations crosscutting implementation packages or class hierarchies. We noticed that a large agglomeration requires that developers reason about a wide range of scattered code smells. As they tend to use each code smell as a symptom of design problem, they have difficulties to correlate the multiple symptoms of an agglomeration. This is a challenging task because the higher the number of code elements involved in an agglomeration, the greater is the volume of code that must be analyzed. Consequently, developers will have more code to analyze, which increases the complexity of the analysis.

Need for proper visualization mechanisms. In order to alleviate the analysis of stinky code, some participants suggested the adoption of better visualization mechanisms. For instance, participant number 8 suggested the visualization of agglomerations through a graph-based representation (Herman, Melancon and Marshall 2000). She mentioned that such visualization would provide an abstract and general view of agglomerations. The main advantage of this form of visualization is that the more abstract a representation is, the less details will be displayed for analysis. Consequently, the developers would not be overloaded with details. At the same time, an abstract representation like the graph-based visualization would help developers to see the full extent of an agglomeration. After providing an abstract view, a visualization mechanism could allow developers to progressively explore the agglomeration details such as the types of smells, location of stinky code and relationships among smells. In order to address this matter, we incorporated a graph-based visualization mechanism into Organic.

Identification of the design problem type. The difficulty in analyzing agglomerations also raised the need for recommendations on which types of design problem each smell agglomeration is more likely to indicate. These recommendations would reduce the effort required to decide whether the elements are affected by a specific design problem. For example, the agglomeration of Figure 3.3 occurs in classes of the same hierarchy that are implementing the *WorkflowRepository* interface. All smelly elements of this stinky program location manifest the same type of smell, which is the *Feature Envy*. The occurrence of multiple Feature Envies in a single hierarchy suggests that there is a problem, in a root abstract class or a root interface, which is spreading through all the subclasses of the hierarchy. Therefore, to help developers to decide whether there is a problem or not, the Organic tool could suggest the analysis of this hierarchical agglomeration trying to identify problems like Ambiguous Interface (Garcia *et al.* 2009) and Fat Interface (Martin and Martin 2006), for example.

Suggestions of design problem types can help developers to focus their attention is specific characteristics of the suggested design problems. However, this kind of recommendation algorithm requires multiple empirical studies to understand how and when each form of agglomeration may consistently represent specific types of design problem. This is indeed a challenging research topic to be addressed in the future and, therefore, we are unable to provide this recommendation feature in the Organic tool.

3.4.3
Threats to Validity

This section presents some threats that could limit the validity of our main findings. For each threat, we present the actions taken to mitigate its impact on the research results.

The first two threats to validity are related to the number of participants in the study and to the convenience approach used to find participants. We have selected a sample of 11 participants, which may not be enough to achieve conclusive results. However, instead of drawing conclusions based on the quantitative results, we complemented our analysis with a qualitative analysis. In addition, we defined a set of requirements to selecting participants suitable for this study. Also, we conducted training sessions with all participants. Such sections aimed to resolve any gaps in the participants' knowledge and any terminology conflicts, allowing us to increase our confidence in the results.

The third threat is related to possible misunderstandings during the study. As we asked developers to conduct a specific software engineering task and to answer a questionnaire, they could have conducted the study different from what we asked. To mitigate this threat, we assisted the participants during the entire study, and we make sure of helping them to understand the experimental tasks and the questionnaire. We highlighted that our help was limited to only clarify the study in order to avoid some bias on our results.

Next threat concerns the ground truth used to confirm or to refute the design problems reported by participants. Since our ground truth was built manually, it is possible that some design problems are missing in the ground truth, which would impact the precision measure. To mitigate this threat, we built the ground truth with the help of original OODT designers and developers. Moreover, we relied on a suite of design recovery tools to identify possible design problems that were not reported by original designers and developers of OODT.

There is another threat that is related to the amount of information we asked participants to provide for each design problem reported. Providing all information during the experiment may slow down the participants and, as a consequence, some participants may report fewer design problems than they would be able to do during the 45-minute time frame. We mitigated this threat by asking the same amount of information for both the agglomeration group and the control group.

Finally, there are two threats concerning the selected projects. The first one is about the difficulty of the participants in understanding the source code used in the experimental tasks. This difficulty appears due to the complexity of the source code and time constraints to complete each task. The second threat is related to one software project could be easier to identify design problem than the other. We minimized the first threat by running a pilot study to define a experimental time reasonable to perform the tasks. To minimize the second threat, we selected projects with similar size, complexity, and number of known design problems. We also have trained all participants about each project. In addition, the cross design of our experiment allowed different combinations of project and technique. Finally, our results suggest no variation in difficulty for identifying design problems in the two projects.

3.5 Study II: Communicability Evaluation of Organic

In the previous section, we presented a *quasi-experiment*, which allowed us to gather quantitative and qualitative results regarding design problem identification. In addition, we collected the opinion of developers regarding the use of code smell agglomerations. Nevertheless, the previous study did not evaluate Organic, which is the tool proposed in this chapter for helping in the identification of design problems. Thus, we did not have shreds of evidence that indicate to what extent Organic affect the users during the identification of design problems. In order to understand these effects, we conducted a qualitative evaluation of the Organic tool. We opted for proposing and evaluating Organic because, to the extent of our knowledge, it is the only tool that meets the requirements (Section 3.3) for helping developers in the analysis of stinkier code.

To evaluate Organic, we applied the Communicability Evaluation Method (CEM) (Prates *et al.* 2000). CEM is a qualitative evaluation method developed to capture communicability issues, which are problems that appear due to poor communication between users and a system, usually when users interact with a system. An example of communicability issue is when the user mistakenly believes that she performed a certain task on the system successfully. Another example is when the user does not understand the answers provided by the system. In our case, we are interested in communicability issues that happen when developers interact with the Organic tool. We have to investigate these communicability issues because they may hinder the identification of design problems when developers use the Organic tool.

CEM has been widely used in HCI (Human-Computer Interaction) research to evaluate the communicability of software systems. This method is based on the theory of Semiotic Engineering (de Souza *et al.* 2009) and is intended to find ruptures of communication when a user interacts with a system. Thus, in order to use CEM, we have to characterize the system and users in the context of our study. As explained in Section 3.3, developers use the Organic tool to identify design problems. Hence, in the context of this study, the system is the Organic tool and the user is the software developer that uses Organic to identify design problems in stinky code. Therefore, the objective of this study is to use the CEM to find communicability issues in the Organic tool.

In the subsection below we present details about this study. Section 3.5.1 presents the study design. Section 3.5.2 contains an overview of the evaluation procedure followed in this study. Section 3.5.3 presents the results. Finally, Section 3.5.4 outlines the threats to validity of this study.

3.5.1 Study Design

We defined our third research question to evaluate the Organic tool as follows:

RQ3. Which are the communicability issues of Organic that hinder the identification of design problems?

To answer RQ3, we followed the procedure defined by the CEM. For being a method focused on user experience, CEM allowed us to look at the Organic tool from the standpoint of potential users, which are professional software developers. In this way, we can observe the aspects of the tool that affect the identification of design problems as if we were the users ourselves. Moreover, such observation was not fully accomplished by our previous study (Section 3.4), since the primary goal there was to evaluate precision of developers when using the technique (code smell agglomerations) rather than the tool itself.

CEM requires the participation of potential users of the system under evaluation. Therefore, similarly to the previous study (Section 3.4.1.2), we selected participants for this study according to the following requirements:

- Minimum of 4 years experience with software development
- Intermediary knowledge about software design
- Advanced knowledge about the Java programming language
- Basic knowledge about the Eclipse development environment
- Basic knowledge about code smells

Requirements above are justified by the fact that Organic is part of a complex domain, which is the identification of design problems. Therefore, participants that have a minimum knowledge about basic concepts have more chance of revealing communicability problems when using the tool. This happens since the influence of the domain complexity is mitigated by the experience and knowledge of participants. Based on the aforementioned requirements, we selected 3 participants for this study. On Table 3.8, we summarize the profile of each participant, presenting their experience with Software Design, Java

Programming Language, Eclipse IDE, and Code Smells. Participant 1 is a developer with vast experience both in academic field and software industry. He has advanced knowledge about software design and code smells. Participant 2 is a professor from the computing field with 4 years of experience in the industry. He has intermediary knowledge about software design and basic knowledge about code smells. Finally, participant 3 had moderate experience in the industry, and he was also undergoing his postgraduate studies at the time of this evaluation. He has advanced knowledge about software design and intermediary knowledge about code smells. Table 3.8 summarizes the profile of all participants.

Table 3.8: Profile of selected participants

Participant	**Software Design**	**Java Programming Language**	**Eclipse IDE**	**Code Smells**
1	Advanced	Advanced	Advanced	Advanced
2	Intermediary	Advanced	Advanced	Basic
3	Advanced	Advanced	Basic	Intermediary

3.5.1.1
Test Scenario

After selecting the participants, the definition of a test scenario is the next CEM procedure. Since Organic is designed to be applied to a single task, which is the identification of design problems, our test scenario is composed of one task as well. The task consists in:

> Using the Organic tool to search for design problems in the source code of a given software project.

In the context of this study, Apache OODT (Mattmann *et al.* 2006) is the selected software project. We selected Apache OODT due to the same reason explained at Section 3.4.1.2. As defined by the CEM, this task was designed to last at most 30 minutes. During the execution of this task, for each design problem found, the participant should give the following information:

– Brief description of the problem.
– Classes and methods participating in the design problem.
– Tool resources that were useful to identify the problem.

During the identification task, besides using the Organic tool, the participants could consult three documents: (1) Apache OODT documentation, (2) a reference document about basic concepts (design problems, code smells, and the like), and (3) manual of the Organic tool. We provided these documents to help users to gather an understanding of the system. Consequently, they were able to focus on the task instead of wasting time trying to understand, for example, the system and basic concepts. This initial preparation was not part of the time frame of 30 minutes.

3.5.1.2
Environment and Infrastructure

To perform this study, we used an individual room equipped with a computer containing the following hardware configuration: 8GB of RAM, CPU Intel Core i5 2.7GHz, GPU GeForce GT 740M, and built-in microphone in the notebook. In addition, we used following softwares: Organic tool, Windows 10 Operational System, Java Development Kit 1.7, Eclipse Luna IDE, Rabbit Eclipse Plugin, and Screen record tool Active Presenter. Eclipse Luna was chosen as the Organic tool only works with this version at the moment. The Rabbit plugin registers information regarding the time spent using the resources of Eclipse (e.g., files, perspectives, views, etc.). This information is useful to the analysis and interpretation of videos recorded using the Active Presenter.

3.5.1.3
Post-study Interview

Using the interview pre-test, we collected data regarding the participant's profile. The questions of the interview post-test were developed individually for each participant. Thus, the evaluators, based on their observation, could explore the participants' answers. Additionally, the following questions were asked to all participants:

– What were the main difficulties to perform the task?

– What were the most useful tool resources?

3.5.2
Data Analysis and Evaluation Procedure

In order to conduct our evaluation following the CEM, we collected the following data: (1) video from the computer screen with audio from the microphone, (2) report collected with the Rabbit Plugin, (3) annotations done during the execution of the tests, and (4) answers given during the interview.

After data collection, we followed three main steps, which are defined by the CEM (de Souza *et al.* 2009). They are: (1) tagging, (2) interpretation, and (3) semiotic profile. On tagging, the researcher analyzes the recording of the task being performed, after that, she identifies the evidence of communicability failures. To each of these failures, she associates with one of the 13 tags defined by CEM (de Souza *et al.* 2009). On the interpretation, the researcher works with the tagged data, trying to identify the main communicability issues. The researcher then analyzes and organizes the collected evidence, according to some perspectives. Finally, in the semiotic profiling step, an in-depth characterization of metacomunication is achieved. The idea of these steps is to achieve higher levels of abstraction in our analysis and interpretation of how the developers receive communication from the Organic tool (de Souza *et al.* 2009).

Tagging. In this step, we analyzed and tagged the communicability failures that occurred during the interaction between software developers and the Organic tool. A communicability failure is the result of a communicability issue. In our case, if the Organic tool contains a communicability issue, this issue will lead to a communicability failure observed when the user interacts with the tool. Thus, tagging was made according to what happened when the communicability failures were observed. To perform this step, we observed each participant during the task of identifying design problems, taking notes of possible communicability failures. After that, we analyzed the video and audio recorded during the task registering communicability issues according to 13 tags defined by the CEM (de Souza *et al.* 2009). Whenever necessary, we consulted the Rabbit reports to confirm or to change the tags. For a detailed description of the tagging step, we refer to (de Souza *et al.* 2009, Prates *et al.* 2000). Table 3.9 presents a brief description of the tags that occurred in this study.

Interpretation. In this step, we analyzed the tagged material aiming to identify the main communicability issues in the Organic tool. Based on the CEM (de Souza *et al.* 2009), we analyzed and organized collected evidence based on three perspectives:

- Frequency and context of each communicability failure.
- Recurrent sequences of communicability failures.
- Identification of communicability issues that have caused the observed failures.

The analysis of frequency and context of communicability failures was helpful

Table 3.9: Description of CEM tags that occurred in this study

Classification	Tag	Description
Complete Failure	I give up	The developer is unable to identify design problems with Organic either because he does not know how to or because he does not have enough time, or will, or patience for it.
Partial Failure	I can do otherwise	The developer manages to identify design problems in a way that is not optimal. For example, without using the most functionalities provided by Organic.
Temporary Failure - Communicate	What now?	The developer searches for a clue of what to do next and not searching for a specific functionality that will help in the identification of design problems.
Temporary Failure - Understand the Rules	What is this?	The developer seems to be exploring the tool to gain more (or some) understanding of what a specific functionality achieves.
	Help!	The developer deliberately calls a help function, using menus, dragging question marks, or asking for help.
	Why doesn't it?	The developer expects some sort of outcome from Organic, but does not achieve it. She steps through the path, again and again, to check that it is not working.

to discover the most frequent failures in the communication between software developers and the Organic tool. Identifying recurrent sequences of failures helped us to discover the origins of communicability issues in Organic. Finally, the identification of communicability issues in the Organic tool is the main objective of this study, as defined by our main research question. To identify communicability issues, we classified tags as Complete, Partial, or Temporary Failures (first column of Table 3.9), following theoretical tag categorizations from Semiotic Engineering (de Souza *et al.* 2009).

Complete failures occur when the developers is unable to identify any design problem with Organic and do not try again. Partial failures occur when the developer gives up from using Organic's functionalities before identifying any design problem and tries to achieve this in another way. Finally, temporary failures occur when the developer temporarily interrupts the identification of design problems with Organic due to some communicability issue. According to the CEM, there are three types of temporary failures: (1) trying to communicate, (2) trying to fix an error, and (3) trying to understand the rules.

Semiotic Profiling. In this last step, we conducted an analysis to understand the communication between software developers and the Organic tool. After executing all steps defined by the CEM, we were able to look at Organic as if we were the users ourselves. This helped us to acquire a deeper understanding of Organic's communicability issues. In addition, looking from the perspective of potential users, we were able to identify the requirements for a tool that supports the identification of design problems in stinky code.

The steps defined by the CEM were fundamental for achieving the goal of this study: discover communicability issues in Organic that hinder the identification of design problems. The first step (Tagging) provided a guideline for the identification of communicability failures that may occur when a user interacts with Organic. After that, with the Interpretation step, CEM provided us with a systematic method for the analysis and classification of communicability failures. This classification was fundamental for organizing the data collected during this study. Finally, in the last step, we analyzed the data collected in previous steps to consolidate our results. During this step, we identified the main communicability issues of Organic based on the recurrent failures observed during the interaction of software developers with Organic. Next, we present the results and our interpretation of this study.

3.5.3

Results and Interpretation

Table 3.10 presents the frequency of occurrence of each tag (rows) by participant (columns). Also, the total frequency, considering all participants, is summarized in the last column. Table 3.11 presents the frequency of communicability failures, categorized by the type of failure. We did not observe any recurring pattern of failures among the participant. Nevertheless, as seen on Table 3.11, all participants suffered from the "I give up" failure, which is a complete failure. For all of them, the complete failure occurred after successive temporary failures. As exposed in Table 3.11 most of the temporary failures were of type 3 - that occur when the developer is trying to understand the communication rules of Organic. This sequence of failures indicates that developers tried to identify design problems with Organic. However, due to successive failures, the developers gave up on the task.

Table 3.10: Frequency of occurrence total and by participant

Tag	Participant			Total
	P1	P2	P3	
I give up	1	1	1	3
Go another way	1	0	1	2
And now?	0	1	0	1
What is this?	1	4	2	7
Help!	2	8	2	12
Why doesn't it?	4	0	0	4

Table 3.11: Frequency of occurrence categorized by type of failure

Type of Failure	**Participant**			**Total**
	P1	**P2**	**P3**	
Complete Failures	1	1	1	3
Partial Failures	1	0	1	2
Temporary Failures - Type 1 *Communicate*	0	1	0	1
Temporary Failures - Type 2 *Fix an Error*	0	0	0	0
Temporary Failures - Type 3 *Understand the Rules*	7	12	14	23

3.5.3.1 Communicability Issues of Organic

Answering research question RQ3, we observed three main communicability issues in the Organic tool, which are: (1) lack of a precise message, (2) inadequate terminology, and (3) ambiguity in static signs. Next, we present details about each of them.

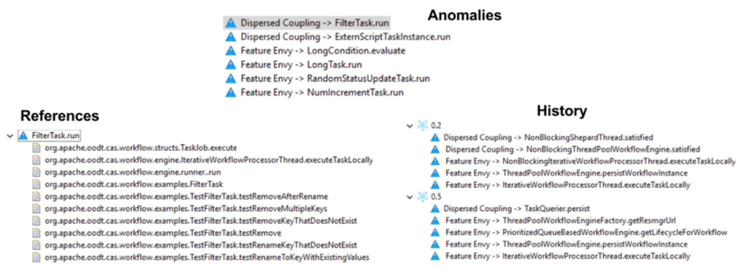
Lack of a Precise Message. Although the tool identifies and groups the symptoms that are interrelated (the agglomerated code smells), it does not provide a message that facilitates concise reasoning about the possible design problem. Hence, the developer needs, by himself, to explore and synthesize all the information needed to analyze a design problem. The tool gives the necessary information, but the analysis of those information requires a significant effort from the developer. Besides being a communicability issue, it also has relation with the domain complexity in which Organic is designed for. We list next some changes in Organic that can contribute for building and delivering a more precise message.

Following our findings from the previous study (Section 3.4.2.2), for this study, we incorporated a graph-based view into Organic. However, we observed that developers did not use the graph-based view of Organic. In the post-study interviews, we noticed that this happened because the graph-based view did not prove to be useful for identifying design problems. Nevertheless, we believe that, after some improvements, this type of view can indeed contribute for transmitting a more concise message about each agglomeration. For that, it is required a better integration of this view with other information. For example, instead of using a separated tab, the description of code smells could be provided in the graph nodes. Moreover, the graph could show the relationships between the agglomeration external classes. This information could be provided on demand, when required by the developer. We believe that such improvements would help developers to conduct an integrated, smoother analysis of an agglomeration.

Inadequate Terminology. The terms used in the tool are not adequate to the target public, which are common software developers. Terms such as "anomalies" and "agglomeration", for instance, are unknown by most developers. Two participants (P2 and P3) who have the least knowledge about software engineering mentioned the fact that the concepts embedded in the tool were too hard to understand. The participant (P1) that did not have difficulties with the terms was a postgraduate student, acting on the software developer field.

To improve the communicability of Organic, participants suggested maximizing the use of terms from popular books like the books of Fowler (Fowler 1999) and Martin (Martin 2008), which are widely known in the software development community. In addition, as observed in the *quasi-experiment* (Section 3.4), for being a complex domain, developers would benefit from interactive help content. This kind of aid should be available, at least, in the first interaction between the developer and Organic.

Ambiguity in Static Signs. The last problem of communicability occurs due to the inadequate use of static symbols. As presented in Figure 3.15, different types of information are presented with the same static symbols. This mixture confused all the participants, leading to situations in which, for instance, the participant believes that he is interacting with the tab "Anomalies," when in fact he was interacting with the tab "References." This is the least harmful communicability issue, but also affects the identification of design problems.



```
Dispersed Coupling -> FilterTask.run
Dispersed Coupling -> ExternScriptTaskInstance.run
Feature Envy -> LongCondition.evaluate
Feature Envy -> LongTask.run
Feature Envy -> RandomStatusUpdateTask.run
Feature Envy -> NumIncrementTask.run
```

```
FilterTask.run
    org.apache.oodt.cas.workflow.structs.TaskJob.execute
    org.apache.oodt.cas.workflow.engine.IterativeWorkflowProcessorThread.executeTaskLocally
    org.apache.oodt.cas.workflow.engine.runner..run
    org.apache.oodt.cas.workflow.examples.FilterTask
    org.apache.oodt.cas.workflow.examples.TestFilterTask.testRemoveAfterRename
    org.apache.oodt.cas.workflow.examples.TestFilterTask.testRemoveMultipleKeys
    org.apache.oodt.cas.workflow.examples.TestFilterTask.testRemoveKeyThatDoesNotExist
    org.apache.oodt.cas.workflow.examples.TestFilterTask.testRemove
    org.apache.oodt.cas.workflow.examples.TestFilterTask.testRenameKeyThatDoesNotExist
    org.apache.oodt.cas.workflow.examples.TestFilterTask.testRenameToKeyWithExistingValues
```

```
0.2
    Dispersed Coupling -> NonBlockingShepardThread.satisfied
    Dispersed Coupling -> NonBlockingThreadPoolWorkflowEngine.satisfied
    Feature Envy -> NonBlockingIterativeWorkflowProcessorThread.executeTaskLocally
    Feature Envy -> ThreadPoolWorkflowEngine.persistWorkflowInstance
    Feature Envy -> IterativeWorkflowProcessorThread.executeTaskLocally
0.5
    Dispersed Coupling -> TaskQuerier.persist
    Feature Envy -> ThreadPoolWorkflowEngineFactory.getResmgrUrl
    Feature Envy -> PrioritizedQueueBasedWorkflowEngine.getLifecycleForWorkflow
    Feature Envy -> ThreadPoolWorkflowEngine.persistWorkflowInstance
    Feature Envy -> IterativeWorkflowProcessorThread.executeTaskLocally
```

Figure 3.15: Example of ambiguity in the static symbols

The direct solution for resolving this ambiguity consists on the use of different static symbols for each type of information. Also, the improvement on the graph visualization - mentioned to solve the first communicability issue - would be an excellent alternative to solve this problem. The graph would integrate the different information in a single view, removing existing ambiguities.

3.5.3.2
Communicability Strengths of Organic

Despite presenting some communicability issues, Organic also has its strengths revealed in this study. In fact, there was no previous study evaluating Organic. Therefore, looking from the perspective of Semiotic Engineering, besides identifying communicability issues, we also identified some communicability strengths of Organic. This provided us with evidence on what is working well in Organic. It is important to note that the strengths presented here were not reported by participants. Instead, they are the result of our own observations.

Next, we present the main strengths found in Organic during this study:

Multiple Analyses of Stinky Code. The identification of a design problem in stinky code requires multiple complementary analyses. Organic provides the opportunity to analyze stinky code based on different agglomeration categories (Section 3.2.1). As reported by Oizumi and colleagues (Oizumi *et al.* 2016), each agglomeration category provides a different perspective to the analysis of source code. In addition, Organic provides multiple information about each agglomeration: (1) list of code smells, (2) description of code smells, (3) dependencies of the agglomeration with external classes, (4) a high level visualization, and (5) information about the agglomeration across different versions of the source code. We observed, in this study and in the *quasi-experiment* (Section 3.4), that developers were able to identify design problems when they managed to explore and synthesize the multiple information provided by Organic for each agglomeration.

Integration with IDE. The analysis of stinky code requires the developer to constantly navigate from Organic to the source code, and vice-versa. This is necessary because most code smells can only be fully understood in the source code. Moreover, the analysis of source code is required to verify if a code smell is a false positive. As an Eclipse plugin, Organic promotes a smooth integration of its views with the source code. In addition, the developer can open the source code affected by a code smell with a double-click in the code smell. Without this integration, the developer would have to constantly shift between programs to analyze stinky code. For example, without this resource, the analysis of an agglomeration of 5 code smells would require, at least, 9 shifts between programs.

Information about Code Smells. Most developers benefit from reading information about code smells during the analysis of stinky code. We observed in this study that, during the analysis of an agglomeration, even experienced developers usually consult the definition of each smell. As explained in Section 3.3, Organic provides this information in a tab called “Description”. We noted that developers used the “Description” tab (back and forth) as a guide for analyzing agglomerated code smells. Providing this information is important because even experienced developers do not know or remember the definition of all code smells. Thus, without this resource, they would spend more time analyzing an agglomeration and searching for information about code smells via external resources.

3.5.4 Threats to Validity

This section presents threats that could impact the validity of this study. For each threat, we present the actions we took to mitigate its impact on the study.

First, one could claim it would be beneficial to have more participants in the study in order to achieve representative results. However, according to Yin (Yin 2015), qualitative research is, by nature, particularistic. Thus, the analysis and understanding of qualitative results requires the study of specific situations and people, complemented by considering specific contextual conditions. We selected three software professionals, which are representative individuals of our target population. Thus, we consider that this threat was properly mitigated. However, we plan to perform other studies in the future in order to analyze the behavior of professionals with other types of background and using future versions of Organic.

The second threat is related to possible misunderstandings during the study. To mitigate this threat, we prepared the participants before the study, explaining how the study would proceed. Moreover, in order to comply with the recommendations of CEM, two researchers assisted the participants during the entire study.

Finally, there is a threat related to the complexity of the system used in this study. We mitigated this threat by selecting system from a widely known computer science domain – Apache OODT is a middleware system that provides infrastructure services, such as file management and networking communication. In addition, as presented in Section 3.5.1.1, we provided participants with the required documentation of Apache OODT.

3.6 Concluding Remarks

In this chapter, we presented Organic – a tool supporting the analysis of stinky code. Organic is a tool to help developers to identify design problems through the analysis of code smells in the source code. Organic supports the analysis of multiple forms of stinkier code, provides detailed information about code smells, supports the analysis of dependencies involving stinky code, provides a graph-based visualization for stinkier code, and it provides historical information about stinkier code. These features were designed and implemented based on findings from previous studies about the relation between design problems and code smells. We believe these features can provide the basic support to support developers on identifying design problems.

In addition to proposing Organic, we also conducted two studies to assess if developers are effective in revealing design problems when they reason about agglomerated code smells, and to identify tool issues that may hinder the identification of design problems. These studies were important because they revealed that: (i) developers find more design problems (and report less false positives) with agglomerations than with a flat list of smells, and (ii) developers sometimes may not be able to identify design problems either because they cannot properly reason about multiple code smells or because limited support tool is hindering the identification. Thus, we address these two aspects not explored by previous studies.

In the first study, we conducted a multi-method study with 11 developers. We asked participants to identify design problems in stinkier program locations. After that, we compared their results with the results of when they analyzed a flat list of single code smells to identify design problems. Our analysis showed that developers found more design problems when they reasoned about stinkier code (i.e., agglomerated smells). In addition, we noticed that, when developers were aware of multiple smells in a program location, they reported less false positives. Therefore, our results suggest that reasoning about stinkier code may improve the precision of developers in identifying design problems. Based on the qualitative analysis, we observed that developers indeed tend to have higher confidence to identify the occurrence of non-trivial design problems when using information about multiple smells. That happens because developers usually analyze all smells before reporting a design problem. Consequently, the likelihood of reporting a false positive decreases.

Additionally to these results, this first study also helped us to identify opportunities to improve the tool support for developers. For instance, we observed that developers need to prioritize stinkier program locations that are most likely to indicate a design problem. This need should be addressed because the analysis of stinky code is difficult and time-consuming. Furthermore, a system may contain several stinkier locations, which choosing which location to analyze can be a cumbersome task for developers. Thus, developers should focus on those locations that are most likely to embody a design problem. In addition, we also noticed that developers need proper visualization mechanisms to support the analyses of stinky code scattered across wider program locations, such as hierarchies or packages.

In the second study, we evaluated Organic with the Communicability Evaluation Method (CEM) (de Souza *et al.* 2009). This method enabled us to identify communicability issues in the Organic tool that may hinder the identification of design problems. For example, we observed that, although detecting stinkier program locations, Organic does not provide a message containing concise reasoning about the possible design problem occurring in the stinkier code. As a result, the developer may struggle to make a meaning out of multiple smells.

The second study also revealed some strengths of Organic. For instance, we observed that Organic provides useful information about code smells and about dependencies. Such information was considered useful by most participants. We also observed that Organic promotes a smooth integration of its views with the source code. This characteristic is important because most code smells can only be understood through source code analysis.

In a nutshell, we conclude that both studies encourage the analysis of stinky code to identify design problems. However, there are issues that should be addressed before developers can more effectively explore multiple code smells in a time-effective manner. As discussed above, there is a need to provide mechanisms for better prioritizing and visualizing stinkier code. As a future work, we plan to improve these mechanisms in the Organic tool and evaluate their impact on developers' effectiveness and efficacy.

4
Filtering and Ranking Design-Related Agglomerations of Code Smells

Code smells are symptoms in the source code that could help to identify design problems. However, developers may feel discouraged to analyze multiple smells if they are not able to focus their attention on a specific context of interest. Unfortunately, current techniques fall short in assisting developers to prioritize and filter smelly locations that are likely to indicate design problems. Furthermore, in Chapter 3, we found evidence that developers often have trouble analyzing interconnected smells that contribute together to realize a design problem. As a result, one of our conclusions in Chapter 3 indicates that this difficulty can be alleviated by filtering and prioritizing refactoring candidates.

Therefore, to deal with these issues, this chapter presents and evaluates a suite of five criteria for ranking groups of code smells as indicators of design problems in evolving systems. These criteria were implemented in a tool called *JSpIRIT*. In a first experiment, we have assessed the criteria in the context of 23 versions of 4 systems and analyzed their effectiveness for revealing design problem locations. In addition, we conducted a second experiment for analyzing similarities between the prioritization provided by developers and the prioritization provided by our best performing criterion. The results provide evidence that one of the proposed criteria helped to correctly prioritize more than 80 code locations of design problems, alleviating tedious manual inspection of the source code vis-a-vis with the design.

We published all the results of this study in a paper at the *Science of Computer Programming* (SCICO) journal.

For a reader who went through Chapters 2 and 3, you can skip Section 4.2.1 as it show definitions already presented there.

4.1 Introduction

Software systems usually suffer from design problems introduced either during development or along their evolution. Several systems have been restructured with high costs or even discontinued due to the constant occurrence of design problems (Bass *et al.* 2003, Kazman *et al.* 2015, Wong *et al.* 2011). Many design problems occur when one or more components of a system are violating design principles or rules (Rosik *et al.* 2008). By component, we mean a software entity that encompasses a set of related functions and often hides the complexity of their implementation. A component might be realized by one or multiple classes in the source code. For instance, in some Java systems, there is a direct mapping between components and packages; while in other systems each component may be realized by classes scattered in different packages.

The violations of principles or rules negatively affect the maintainability and other quality attributes of a system (Terra and Valente 2009, Oizumi *et al.* 2016). Typical examples of design problems are Fat Interface and Unwanted Dependency between components (Sousa *et al.* 2017). The former violates the principle of separation of concerns (Bass *et al.* 2003), while the latter violates a dependency rule in the system's design (Bass *et al.* 2003). Both of them often negatively affect software maintainability and performance (Garcia *et al.* 2009b).

Unfortunately, the identification of code locations in a system that likely indicate design problems is time-consuming and cumbersome for several reasons. This task generally requires the analysis of both design documentation and the realization of design decisions in source code. It is hard for a developer to effectively explore these two types of information together in order to uncover possible design problems. Even when design information is available, it is often not detailed enough to help developers to reveal design problems (Oizumi *et al.* 2016, Garcia *et al.* 2009b, Oizumi *et al.* 2015). Thus, developers need to resort to hints in the source code for the presence of design problems. An occurrence of certain types of well-known code anomalies (i.e., *code smells*) (Fowler 1999) may provide helpful, albeit partial, hints of the location of design problems in a system (Oizumi *et al.* 2016). Classical examples of code smell types often related to design problems are Long Method, God Class and Feature Envy (Oizumi *et al.* 2015, Fowler 1999).

Even for systems of modest size, developers might need to analyze hundreds of smells and infer their likelihood of indicating a design problem. As those systems evolve, the number of smells tends to grow across system versions, thereby further obscuring the location of design problems in a program (Chatzigeorgiou and Manakos 2014, Tufano *et al.* 2015). In such situations, developers do not know where to focus, i.e., which few groups of code smells can be used as a starting point for locating design problems in the source code. We argue that a practical strategy is to prioritize groups of code smells according to their criticality to the system design, that is, their ability to point out one or more design problems. Unfortunately, to the best of our knowledge, existing techniques do not support developers in automatically prioritizing code smells to reveal design problems, thereby discouraging them from locating such symptoms in their programs.

In this context, we defined the concept of code-smell agglomeration in prior work (Oizumi *et al.* 2016, Oizumi *et al.* 2015) and explored such a concept in Chapter 3. An agglomeration is a group of inter-related code smells (e.g., code smells occurring in the same inheritance tree) that likely indicate together the presence of a design problem (Section 4.2). This group of smells can be composed by instances of the same or different kinds of smells. However, we did not address the key challenge of filtering and prioritizing smell agglomerations that indicate design problems. In fact, many smell agglomerations are not related to design problems (Oizumi *et al.* 2016, Oizumi *et al.* 2015). Therefore, in a previous work (Vidal *et al.* 2016) we proposed three ranking criteria for supporting the prioritization of smell agglomerations. The goal was to assist developers in: (i) finding agglomerations that more likely indicate design problems, and (ii) for each design problem, identifying its full extent in the source code by inspecting the group of smells comprising a top-ranked agglomeration, while discarding irrelevant smells. In order to rank smell agglomerations according to their impact on the design, the proposed criteria consider both the system implementation and design information that may be available. Although the criteria were useful, an initial evaluation revealed some limitations for spotting design problems.

In this chapter, we extend the initial set of criteria (Vidal *et al.* 2016) by proposing two additional ranking criteria, which are original contributions of this work. The first criterion is based on how relevant an agglomeration is regarding its type. (Section 4.3.1). An agglomeration type defines the search strategy used to group predetermined smells. An example of search strategy is to look for code smells of the same type, which occur across classes of the same inheritance tree. The second criterion is based on the relation between agglomerations and modifiability scenarios (Section 4.3.3). A modifiability scenario represents a change-related property that a system must support by means of planned feature changes in the system. Thus, agglomerations related to changes could indicate critical modifiability properties of the design.

Given these new criteria, we repeated the initial experiment in (Vidal *et al.* 2016) and assessed how each of our five criteria helps developers to locate symptoms of design problems in the source code while keeping aside irrelevant (agglomerated and non-agglomerated) code smells. Our study considered 23 versions of 4 Java systems (Section 4.4). Furthermore, one of the systems had a large size (1400 classes and 1800 code smells), to test the scalability of our approach. Our results (Section 4.5) show that the use of one of the criteria, the so called agglomeration flood, can consistently and accurately indicate several design problems. The criterion helped to correctly locate more than 80 design problems in our top-7 rankings of the 4 systems, alleviating tedious manual inspections of the source code vis-a-vis with the design. Moreover, this prioritization criterion would have also helped developers to discard at least 500 code smells having no relation to design problems in the analyzed systems. In particular, we observed that the prioritization of agglomerations based on their type usually presents mixed results regarding the identification of design problems. As for the criterion based on scenarios, we found that when the scenarios capture unforeseen changes, then the criterion can prioritize agglomerations related to design problems.

As another contribution, to provide a deeper understanding of the results observed in our case-studies in the first study, we conducted a second experiment in which we analyzed how software engineering students rank code agglomerations without tool assistance (Section 4.6). Finally, we reflect upon these findings and present the concluding remarks (Section 4.7).

4.2 Agglomerations as Pointers to Design Problems

In the context of our work, we focus on design problems (Bass *et al.* 2003) that represent violations of design principles or rules (Garcia *et al.* 2009b, Perry and Wolf 1992). We mainly target design problems affecting the modular decomposition of a system into components and their interfaces, i.e., maintainability-related problems (Garcia *et al.* 2009b). Table 4.1 summarizes all the design problems considered in this work.

Table 4.1: Design problems considered in this chapter

Name	Description
Ambiguous Interface	Interface that offers only a single, general entry-point but provides two or more services
Feature Overload	Component that is responsible for realizing two or more unrelated system features (concerns)
Connector Envy	Component that encompasses extensive interaction-related functionality that should be delegated to a connector
Cyclic Dependency	Components that either directly or indirectly depend on each other to function properly
Scattered Feature	Multiple components responsible for realizing the same high-level feature (concern), with some of them responsible for orthogonal features
Unused Interface	Interface that is never used by external modules
Design Violation	Element or relationship that is in the actual design but is not in the intended design, or vice versa

4.2.1
Formal Definition of Code-smell Agglomerations

Based on previous empirical findings (Oizumi *et al.* 2016, Oizumi *et al.* 2015), our premise is that groups of code smells, so-called *code-smell agglomerations*, are normally associated with several design problems. A code-smell agglomeration is a coherent group of inter-related code smells that contributes to the realization of a design problem.

In Figure 4.1 we present an agglomeration meta-model that supports our definitions. The first meta-model element is *code element*, which corresponds to the most basic unit of description in the system implementation. We consider two *code element types*: *class* and *method*. Constructors are also considered as being methods. Fine-grained program elements, such as code blocks and statements, are examined during the detection of code smells. However, as they are less relevant to the system design, we did not include them in our meta-model.

A *code-smell agglomeration* is composed of a set of two or more *code smells*, where each *code smell* affects a single *code element*. A *code element* may be affected by zero or multiple *code smells*. Each *code smell* may be a member of zero or multiple *agglomerations*. Also, each *code smell* is an occurrence of a *type of smell*. A *type of smell* is associated with a specific *code element type*, which means that occurrences of a *type of smell* exclusively affect either *classes*

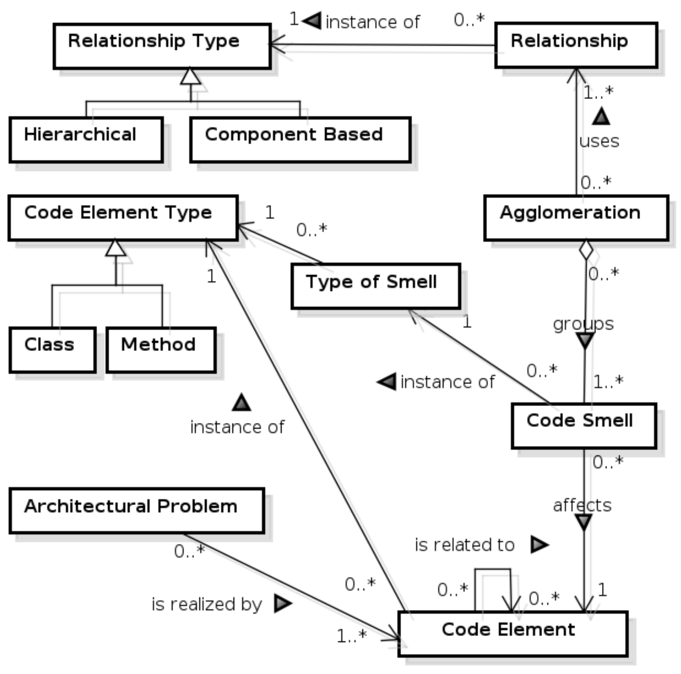


Figure 4.1: Meta model for code-smell agglomerations

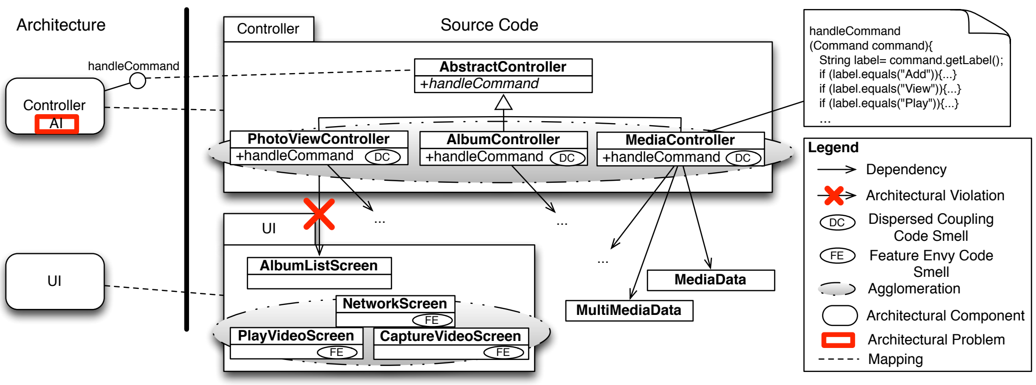
or *methods*.

Code smells are grouped into *agglomerations* based on *relationships* among *code elements*. Two *types of relationship* to group *code smells* are considered. The first one is *common component*, which identifies *code smells* occurring in *code elements* of the same design *component*. The second relationship is *hierarchical*, which identifies code smells occurring in *code elements* of the same *hierarchy*. More details about relationship types are provided in Section 4.2.4. At last, an *design problem* may be reified in the source code by one or multiple *code elements*, which can be *classes* and *methods*.

4.2.2

Illustrative Example

To illustrate the relation between design problems and agglomerations, let us consider the example of Figure 4.2 taken from Mobile Media – a system for managing photos, music, and videos on mobile devices (see Section 4.4.2). The left side of the figure shows a fragment of the component structure of the MobileMedia design. Components *Controller* and *UI* are mapped to separate Java packages in the implementation, each one containing several classes (right side). If a smell detection tool is run over the Mobile Media implementation, the developer would receive a list of more than one hundred code smells. Then, it may not be clear in which code smells she should focus her attention as



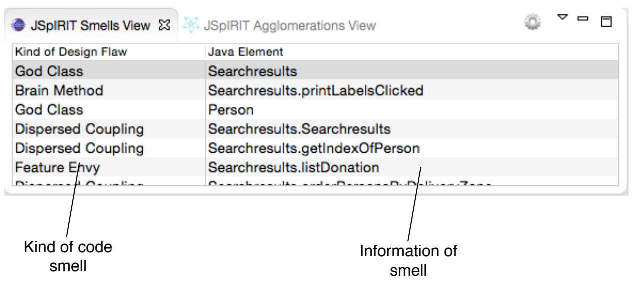
```
handleCommand
(Command command){
   String label= command.getLabel();
   if (label.equals("Add")){...}
   if (label.equals("View")){...}
   if (label.equals("Play")){...}
   ...
```

Figure 4.2: Example of a code-smell agglomeration related to design problems

candidates for revealing design problems. This situation prevents her from performing effective maintenance or refactoring activities.

Design Problems. This example exhibits three design problems. First, the *Controller* component is mainly realized by the class hierarchy rooted in class *AbstractController*, which is responsible for handling different commands through the *handleCommand* method. After a broad look at all the implementations of method *handleCommand* and their callers, the developer realizes, based on her experience, that there is an overload of responsibilities, which leads to two design problems, called Fat Interface and Ambiguous Interface (Garcia *et al.* 2009b). These problems mean that *Controller*, as a design component, provides several non-cohesive services (Fat Interface) that are not properly exposed in its interface *handleCommand* (Ambiguous Interface). Note that the problem is not the controller itself or its object-oriented materialization in terms of an abstract class with many concrete classes, but rather the decision of having only one single controller (at the design level), which can generate ripple effects to other components and their implementations if the controller logic has to be changed. Second, the call from class *PhotoViewController* to class *AlbumListScreen* leads to a usage dependency between packages *Controller* and *UI*, which is not allowed by the component design. This violation is indicated in the design (left side) by the absence of arrows between the two components.

In these examples, a possible way for a developer to identify the design problems is by reasoning about the design documentation and checking candidate problems in the source code. Unfortunately, developers are usually overwhelmed by these tasks because, even with tool support, it is hard to effectively explore all the available information and all code smells to uncover design problems. Thus, the developer needs to turn her attention to the (partial or full)



Kind of Design Flaw	Java Element
God Class	Searchresults
Brain Method	Searchresults.printLabelsClicked
God Class	Person
Dispersed Coupling	Searchresults.Searchresults
Dispersed Coupling	Searchresults.getIndexOfPerson
Feature Envy	Searchresults.listDonation
Dispersed Coupling	Searchresults.orderPersonsByDeliveryZone

Figure 4.3: Code smells detected by *JSpIRIT*

realization of design problems in the source code. Along this line, she might discover that the implementations of *handleCommand* in the subclasses of *AbstractController* are simultaneously affected by the code smell called Dispersed Coupling (DC), which is a method that calls various methods of several classes. That is to say that the subclasses of *AbstractController* generate dependencies on many other classes. Since there are several DC smells within the *Controller* package, this group of smells is considered as an agglomeration. Therefore, this package-level agglomeration is a sign of (potential) design decay (Le *et al.* 2018), which in this case affects the *Controller* component.

However, the developer cannot be fully sure about the design problem exposed by the agglomeration of DC smells, as it could be a false positive. Other agglomerations can be present nearby, as is the case of a group of instances of the smell called Feature Envy (FE) in package *UI*, which corresponds to the *UI* component. FE is a smell representing a class that is more interested in accessing data from other classes (instead of using its data), which often indicates a poor assignment of responsibilities. Things get more complicated for the developer because agglomerations normally vary from a system version to another owing to the creation and fix of code smells. These two factors (false positives and variations over time) motivate our interest in the definition of criteria for prioritizing agglomerations.

4.2.3 Detecting Individual Code Smells

Existing catalogs of code smells define guidelines to identify single smells and to provide tool support for their detection (Fowler 1999, Lanza and Marinescu 2006). In this work, we use the *JSpIRIT*[1] tool for that purpose. *JSpIRIT* is an Eclipse plugin for detecting and ranking code smells according to different criteria (Vidal, Marcos and Díaz Pace 2014b). Figure 4.3 shows the view of *JSpIRIT* that lists the code smells found in a system. Currently, *JSpIRIT*

[1]https://sites.google.com/site/santiagoavidal/projects/jspirit

supports the detection of 10 types of code smells (Vidal, Marcos and Díaz Pace 2014b) following the detection strategies presented in Lanza's catalog (Lanza and Marinescu 2006). Table 4.2 presents a list of all code smell types supported by *JSpIRIT*.

Table 4.2: Types of code smell supported by JSpIRIT

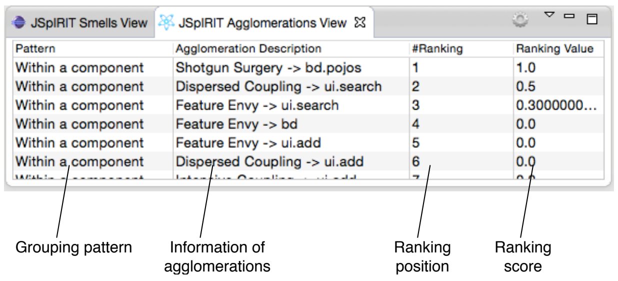
Name	**Description**
Brain Class	Complex class that accumulates intelligence by brain methods
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Disperse Coupling	Method that calls one or few methods of several classes
Feature Envy	Method that calls more methods of a single external class than its own methods
God Class	Long and complex class that centralizes the intelligence of the system
Intensive Coupling	Method that calls several methods that are implemented in one or few external classes
Refused Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	Method called by many methods that are implemented in different classes
Tradition Breaker	Subclass that does not specialize its superclass

4.2.4 Types of Agglomerations

In previous work (Vidal *et al.* 2015), we extended the original version of *JSpIRIT* to support the detection of agglomerations. Figure 4.4 shows a list of agglomerations detected based on the smells detected by *JSpIRIT*. Since our work focuses on design information regarding static code structures, we deal with agglomerations within the scope of design components. For our case-studies (Sections 4.4 and 4.5), we assumed a mapping between a design component and its realization as a Java package in the code. However, developers can flexibly establish other kinds of mappings between components and packages or classes in a program.

We are mainly interested in two particular types of agglomerations:

– **Smells within a component.** This type of agglomeration groups code smells that occur in code elements of the same design component.



Pattern	Agglomeration Description	#Ranking	Ranking Value
Within a component	Shotgun Surgery -> bd.pojos	1	1.0
Within a component	Dispersed Coupling -> ui.search	2	0.5
Within a component	Feature Envy -> ui.search	3	0.3000000...
Within a component	Feature Envy -> bd	4	0.0
Within a component	Feature Envy -> ui.add	5	0.0
Within a component	Dispersed Coupling -> ui.add	6	0.0

Figure 4.4: Code-smell agglomerations detected by *JSpIRIT*

Specifically, we look for one single component with: (i) code smells occurring in code elements that are syntactically related, or (ii) code elements - of the same design component - infected by the same type of code smell. Two code elements are syntactically related if at least one of them references the other one. Figure 4.2 showed an example of this kind of agglomeration where different classes in package *UI* are affected by the Feature Envy (FE) smell.

– **Smells in a hierarchy.** This type of agglomeration groups code smells that occur across the same inheritance tree involving one or more components. We only consider hierarchies exhibiting the same type of code smell. The rationale is that a recurring introduction of the same type of smell in different elements might represent a problem related to the design imposed by the root class or root interface. An example of this agglomeration is the *AbstractController* hierarchy in Figure 4.2 whose subclasses are affected by Dispersed Coupling (DC) smells.

A more complete description of the agglomerations above can be found in previous work (Oizumi *et al.* 2015). Certainly, other types of agglomerations are possible, as we previously reported (Oizumi *et al.* 2016), but they are related to other design problems not addressed in this chapter.

4.3 Prioritization Approach

Since the agglomerations are not enough to completely reveal relevant design problems, additional mechanisms are needed. Along this line, we present five ranking criteria to prioritize agglomerations through scoring criteria. We hypothesize that one or more of these criteria are useful for prioritizing agglomerations with high chances of spotting design problems. In this way, a criterion can be seen as a function:

$$criterion_A(agglomeration_B) = score_{A,B}$$

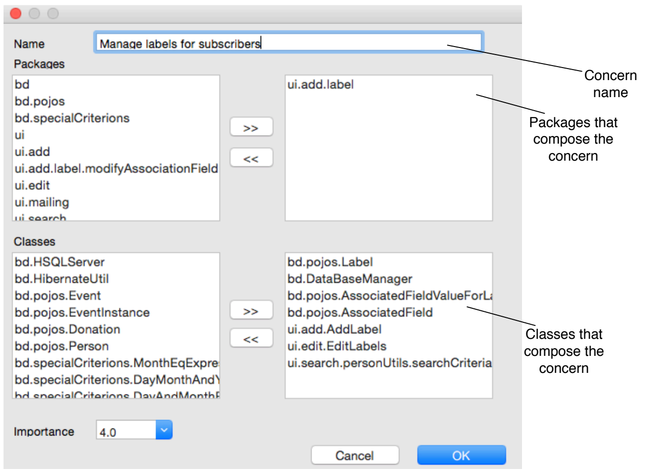
where the score for an agglomeration B given by a criterion A is a value between 0 and 1. The score value indicates how critical the agglomeration is for the system design (0=not critical, 1=very critical).

Details about the five criteria are organized as follows. Section 4.3.1 presents a criterion based on the relevance of each agglomeration type. Section 4.3.2 introduces a criterion for prioritization based on design concerns. Section 4.3.3 shows details about a criterion based on modifiability scenarios. Section 4.3.4 presents a criterion that uses history information for prioritizing agglomerations. Finally, Section 4.3.5 introduces a criterion that explores information about the evolution of agglomerations along different versions of the source code.

All the 5 criteria were implemented in *JSpIRIT*. Furthermore, developers and researchers can add new prioritization strategies or modify the current ones in the tool, as explained in our previous work (Vidal *et al.* 2015).

4.3.1 Agglomeration Relevance

This criterion specifies how relevant a type of agglomeration is for the design of a system. This criterion was initially suggested in (Oizumi *et al.* 2014a), and it relies solely on information about code-smell agglomerations. We conjecture that some types of agglomeration may indicate more design problems than other types of agglomeration. In this way, a developer must choose a relevance value using a [0..1] continuous scale for each type of agglomeration. In this context, 0 means that the agglomeration type is not relevant to the system design and 1 means that agglomeration is very relevant for finding design problems in the system (values in the middle are also allowed). For example, a developer could think that agglomerations affecting classes of a component are more relevant than agglomerations focused on a hierarchy of classes. If so, she can assign a higher relevance value (e.g., 1.0) to the Smells within a component agglomerations and a lower value (e.g., 0.2) to the Smells in a hierarchy agglomerations. In our first experiment, we investigated different configurations of this criterion and prioritized different types of agglomeration in each configuration (Section 4.4.3).


Concern name
Packages that compose the concern
Classes that compose the concern
>>
<<
Cancel
OK

Name: Manage labels for subscribers

Packages

bd	ui.add.label
bd.pojos	
bd.specialCriterions	
ui	
ui.add	
ui.add.label.modifyAssociationField	
ui.edit	
ui.mailing	
ui.search	

Classes

bd.HSQLServer	bd.pojos.Label
bd.HibernateUtil	bd.DataBaseManager
bd.pojos.Event	bd.pojos.AssociatedFieldValueForL
bd.pojos.EventInstance	bd.pojos.AssociatedField
bd.pojos.Donation	ui.add.AddLabel
bd.pojos.Person	ui.edit.EditLabels
bd.specialCriterions.MonthEqExpre	ui.search.personUtils.searchCriteria
bd.specialCriterions.DayMonthAnd	
bd.specialCriterions.DayAndMonthF	

Importance: 4.0

Figure 4.5: Wizard to provide concern mappings in *JSpIRIT*

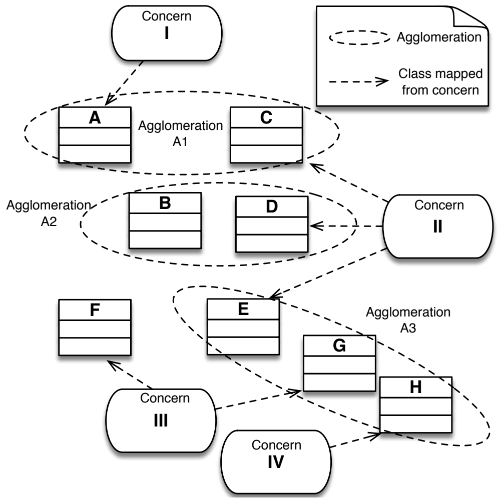


Figure 4.6: Example of concerns mapped to classes

4.3.2 Design Concerns

This criterion analyzes the relationship between an agglomeration and a design concern. A design concern (i.e., feature) is some important part of the problem (or domain) that developers aim at treating in a modular way (Sant'Anna *et al.* 2007), such as graphical user interface (GUI), exception handling, or persistence, among others. For example, in Figure 4.2, the subclasses of *AbstractController* (along with other system classes) address a concern called *PhotoLabelManagement*. This criterion was adapted from (Guimaraes, Garcia and Cai 2014) where it is used for ranking single code smells. The rationale behind this criterion is that an agglomeration that realizes several concerns could be an indicator of a design problem.

The *JSpIRIT* tool offers a simple interface to specify concerns (Figure 4.5). Specifically, the developer must provide a concern name and select the system packages and classes the concern maps to. To compute the ranking score of an agglomeration, we count the number of concerns involved in that agglomeration. A concern is involved in an agglomeration if the agglomeration is located in a class mapped by the concern. At last, we normalize the values to obtain scores between [0..1]. To do so, the highest number of concerns affecting a single agglomeration is used. For example, let us consider the example in Figure 4.6, in which there are three agglomerations (A1, A2, A3) and four concerns (I, II, III, and IV). Agglomeration A1 is related to two concerns (I and II), A2 with one concern (II) and A3 with three concerns (II, III, and IV). Thus, the highest number of concerns per agglomeration is 3, and the agglomeration scores will be $2/3 = 0.66$ for A1, $1/3 = 0.33$ for A2, and $3/3 = 1.0$ for A3.

Certainly, the specific mappings of concerns to program elements affect the results of this criterion. Furthermore, as the implementation evolves, the mappings might need to be adjusted. Existing feature-location tools, such as Mallet (McCallum 2002) and XScan (Nguyen *et al.* 2011), can be used here to derive concern mappings automatically and with high accuracy according to our experience (Oizumi *et al.* 2016).

4.3.3
Modifiability Scenarios

This criterion analyzes the relationship of agglomerations with modifiability scenarios. A modifiability scenario describes a change-related property that is desirable in a system (Bass *et al.* 2003). That is, scenarios describe specific kinds of changes that the system must support. In terms of the design, a scenario affects certain design components that are key for fulfilling the scenario. This criterion was adapted from (Vidal, Marcos and Díaz Pace 2014b), and its rationale is that the agglomerations related to modifiability scenarios can be more critical because they directly affect modifiability properties of the system design.

Scenarios can be considered as particular types of design concerns related to modifiability because, like in the case of concerns, each scenario is mapped to different packages and classes of a system using *JSpIRIT*. Moreover, the developer can give different importance values to the scenarios, by assigning values between 0 and 1 to each scenario to model the criticality of its satisfaction.

To compute the score of an agglomeration, we analyze if the main classes of the code smell (within the agglomeration) are affected by one or more scenarios. We consider "main classes" to be those classes in which the smells are mainly located. For a coupling-related code smell, the main class is the one that causes or that concentrates the anomalous relationships. For example, the main class of a FE smell is the class where the feature-envy method is declared. As another example, the main class of an Intensive Coupling (IC) is the class containing the method that is tied to many other methods. A code smell is affected by a scenario when the scenario is mapped to the main class of such smell.

The agglomeration ranking score is computed as the sum of the importance values of the scenarios that affect each smell being part of the agglomeration (in case there is any). If a code smell is affected by more than one scenario, we only consider the scenario with the highest importance. Once this sum is calculated for all the agglomerations, the values are normalized. For example, let us consider the definition of 2 modifiability scenarios, S1, and S2 with importance values of 0.7 and 0.9 respectively. S1 is mapped to classes Foo and Foo2 while S2 is mapped to Foo and Foo3. Given an agglomeration A1 composed of two code smells whose main classes are Foo and Foo2, its score will be computed as $0.9 + 0.7 = 1.6$. Then, this value will be normalized such that the largest value is 1.

4.3.4
History of Changes

This criterion analyzes the stability of the classes in which the code smells (of an agglomeration) are located. The stability assesses how often the main class of a smell was modified during the lifetime of the system. By looking at the "stability" of the smells within an agglomeration, we want to check whether the agglomeration is in a component or class hierarchy that is usually modified. Our assumption is that agglomerations appearing in classes that changed often should have a higher score, and thus, might hint design problems. Note that this notion of stability relies not only on the actual design information (e.g. an agglomeration affecting a particular component) but also on information from the history of class changes.

To calculate the stability of an agglomeration (and also, its ranking score) we use the LENOM (Latest Evolution of Number of Methods) metric (Girba, Ducasse and Lanza 2004). We previously used this metric to rank single code smells (Vidal, Marcos and Díaz Pace 2014a). LENOM identifies the classes that experienced most changes in the last versions of the system. The classes that most frequently changed are identified by weighting the delta in the method count (NOM) of a class between two adjacent versions. More formally:

$$LENOM_{j..k}(C) = \sum_{i=j+1}^{k} \mid NOM_i(C) - NOM_{i-1}(C) \mid * 2^{i-k}$$

where $1 \leq j < k \leq n$ being j the first version of the system analyzed, k the last version analyzed and n the total number of versions of the system.

Once the LENOM values for each main class of the code smells are obtained, the criterion computes the score of the containing agglomeration by averaging the LENOM values. For example, given an agglomeration A1 that is composed of three Brain Method (BM) smells: Foo.a(), Foo.b(), and Foo2.c(), and knowing that $LENOM(Foo) = 0.8$ and $LENOM(Foo2) = 0.5$, the score of A1 will be $\frac{0.8+0.8+0.5}{3} = 0.7$. A score close to 1.0 means that the classes composing the agglomeration change often during the system history. In contrast, a score of 0.0 means that those classes did not change since their initial implementation.

4.3.5 Agglomeration Flood

This criterion makes an analogy of the agglomeration with a flooding problem in the system history. If the impact of the flood is given by the number of smells contained in an agglomeration, our assumption is that a growing flood is more critical than a flood that seems stable (i.e., the agglomerated smells do not change) or that is shrinking (i.e., the agglomeration has progressively less smells). Along this line, we analyze the behavior of the agglomerations across system versions and compute a variation rate in terms of the number of code smells that compose each agglomeration. This criterion concentrates on the "volume" of smells over time, by combining history-based and design information.

To calculate the ranking score of an agglomeration, we consider pairs of adjacent versions and determine the percentage of variation in the number of code smells that constitute the agglomeration. This percentage will be positive or negative, depending on whether the smells increased or decreased. For example, given an agglomeration A1 with 3 code smells in version v1, 5 smells in v2, and 4 smells in v3, the corresponding variation rates are $\frac{5*100}{3} - 100 = 66.6\%$ from v1 to v2, and $\frac{4*100}{5} - 100 = -20\%$ from v2 to v3 (by definition, agglomerations always have at least two smells). Then, all the percentages of variation (for the same agglomeration) are averaged. In our example, this value becomes $\frac{66.6-20}{2} = 23.3\%$. Once averages for all the agglomeration are obtained, we normalize these values to produce scores in the range [0..1].

4.4

Study Settings

This section describes the research question and hypothesis of our first experiment. We also describe the target applications used in our empirical evaluation, as well as the procedures for data collection and analysis.

4.4.1

Research Question and Hypothesis

To investigate the effectiveness of the scoring criteria on the prioritization of design problems, we defined the following research question (**RQ1**): *Does the use of a scoring criterion assist developers to prioritize smell agglomerations that indicate design problems?* RQ1 is analyzed for each of our 5 scoring criteria in Section 4.5.

We derived the corresponding hypothesis for this research question: ($H1_0$) *the use of a scoring criterion does not assist developers to prioritize critical agglomerations.* We consider that a scoring criterion is effective to assist developers if at least half of the prioritized agglomerations are related to design problems. The reasoning is that developers would give up in inspecting the agglomerations if more than 50% of them are not related to design problems. If the criterion ranks correctly most agglomerations, we can conclude that the criterion enables developers to analyze the most critical agglomerations.

4.4.2

Target Applications

We chose systems for our study that had exhibited several symptoms of design degradation so that we could properly evaluate the effectiveness of the prioritization criteria. The selection of the target systems was performed in two stages. In the first stage, we selected 3 Java applications of a reasonable size (from 10 to 54 KLOC). They were also chosen because either the original developers were available or we could rely on an expert architect, to validate the design problems and code smells being inspected in our analyses. With their assistance, we produced ground truths for the design problems of all the systems. The first application is Mobile Media (MM) (Young 2005), a software product line that provides support for the manipulation of media on mobile devices. The second application is Health Watcher (HW) (Soares *et al.* 2002), a Web-based application that allows citizens to register complaints about health issues in public institutions. Our third application is SubscriberDB (S_{DB}) (Vidal, Marcos and Díaz Pace 2014b), a subsystem of a publishing house that manages data related to the subscribers of its publications, and also supports different queries on the data.

In the second stage, we selected a large system, consisting of 182 KLOC. The goal was to check whether our most effective scoring criteria would also be effective to prioritize design problems in more complex projects. Given this goal, we selected Apache OODT (OODT) (Mattmann *et al.* 2006), a middleware framework aimed at supporting the management and storage of scientific data. A summary of the application characteristics is given in Table 4.3.

Table 4.3: Characteristics of the target applications

Target Application	MM	HW	S_{DB}	OODT
System Type	Software Product Line	Web	Web	Middleware
Programming Language	Java	Java	Java	Java
Architecture Design	MVC	Layers	MVC	Layers
Selected Version	5	8	2.4	10
KLOC	54	49	10	182
#Classes	77	125	151	1424
#Code Smells	260	497	82	1816

4.4.3 Data Collection and Analysis

This section describes the main activities of the study, which are graphically summarized in Figure 4.7. For the sake of reproducibility, the resulting dataset is available in our supplementary material.[2]

[2]https://bit.ly/2E7kOAG

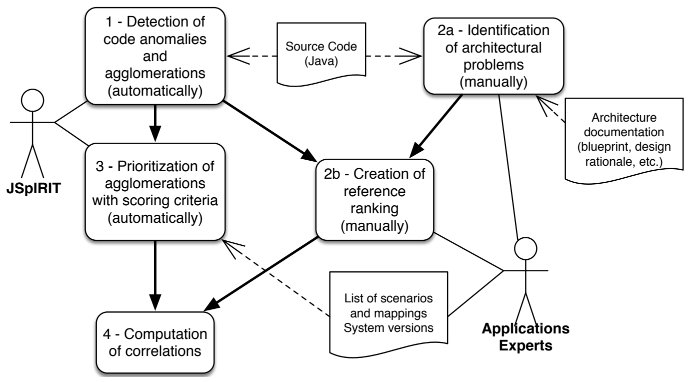
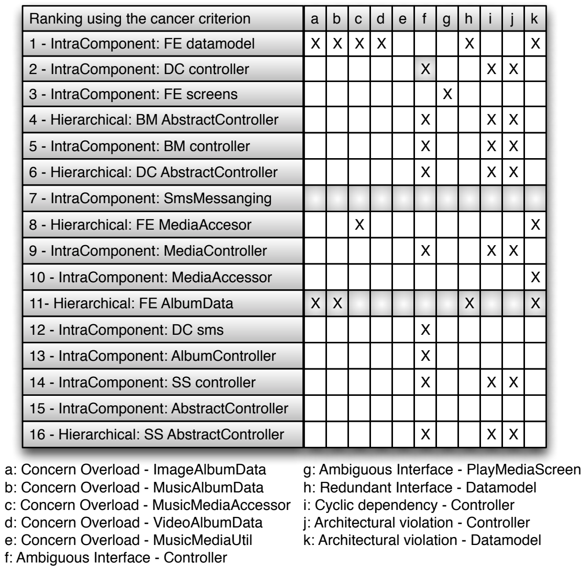

1 - Detection of code anomalies and agglomerations (automatically)
Source Code (Java)
2a - Identification of architectural problems (manually)
Architecture documentation (blueprint, design rationale, etc.)
JSpIRIT
3 - Prioritization of agglomerations with scoring criteria (automatically)
2b - Creation of reference ranking (manually)
List of scenarios and mappings System versions
Applications Experts
4 - Computation of correlations

Figure 4.7: Procedures for data collection

1-Detection of code smells and agglomerations. We used *JSpIRIT* to detect both code smells and agglomerations automatically. After detecting all instances of code smells, *JSpIRIT* proceeds to identify the agglomerations based on the grouping patterns described in Section 4.2.4. *JSpIRIT* presents two different outputs: (i) a list of smell instances, and (ii) groups of inter-related code smells, i.e., the agglomerations, along with their score for a given criterion (Figure 4.4). For the criterion of design concerns, we relied on a list of concerns provided by the original architects of each system. For each concern, they provided a list of packages/classes realizing the concerns, i.e., the concern mappings. A correspondence between design components and Java packages was assumed.

2-Identification of design problems. For HW, MM and S_{DB}, the application developers or the expert architect identified and reported to us the design problems they faced in their projects. Based on a catalog of design problems (Garcia *et al.* 2009b), they reported the existence of 7 types of design problems, namely: Ambiguous Interface, Concern Overload, Connector Envy, Cyclic Dependency, Scattered Feature, Unused Interface, and Design Violations (unwanted dependencies among components). To confirm the presence of design problems, the developers first manually inspected the source code and the design blueprint of each system. Based on their experience with the project, they produced a list of the most critical design problems for each version of the target applications. As a result, using the list of design problems, we produced a reference ranking of the agglomerations detected by *JSpIRIT* that contribute to the most critical design problems for each target application.

To determine if an agglomeration X was linked to a design problem Y, we check whether problem Y mapped to (some of) the main classes hosting the smells of agglomeration X. That is, we looked at intersections between the program elements realizing the design problem and those related to the agglomeration.



Ranking using the cancer criterion	a	b	c	d	e	f	g	h	i	j	k
1 - IntraComponent: FE datamodel	X	X	X	X				X			X
2 - IntraComponent: DC controller						X			X	X	
3 - IntraComponent: FE screens							X				
4 - Hierarchical: BM AbstractController						X			X	X	
5 - IntraComponent: BM controller						X			X	X	
6 - Hierarchical: DC AbstractController						X			X	X	
7 - IntraComponent: SmsMessanging											
8 - Hierarchical: FE MediaAccesor			X								X
9 - IntraComponent: MediaController						X			X	X	
10 - IntraComponent: MediaAccessor											X
11- Hierarchical: FE AlbumData	X	X						X			X
12 - IntraComponent: DC sms						X					
13 - IntraComponent: AlbumController						X					
14 - IntraComponent: SS controller						X			X	X	
15 - IntraComponent: AbstractController											
16 - Hierarchical: SS AbstractController						X			X	X	

a: Concern Overload - ImageAlbumData	g: Ambiguous Interface - PlayMediaScreen
b: Concern Overload - MusicAlbumData	h: Redundant Interface - Datamodel
c: Concern Overload - MusicMediaAccessor	i: Cyclic dependency - Controller
d: Concern Overload - VideoAlbumData	j: Architectural violation - Controller
e: Concern Overload - MusicMediaUtil	k: Architectural violation - Datamodel
f: Ambiguous Interface - Controller	

Figure 4.8: Matrix of ranked agglomerations for MM versus related design problems ('a' to 'k').

Coming back to Figure 4.2, we can see an example of this intersection for the Ambiguous Interface problem, which is realized by the *Controller* package and some of its classes take also part in an agglomeration. The reference ranking of agglomerations was built in such a way that it has in the first positions the agglomerations being related to the highest number of design problems. That is to say, the score of an agglomeration is the number of related design problems. The agglomerations along with their related design problems for each case-study constituted our ground truth. For OODT, we did not produce a ground truth, due to the size and complexity of this application. Our goal with OODT was to evaluate the criteria in a system larger than HWS, MM or S_{DB} and assess the design problems for the best-ranked agglomerations. Along this line, we performed a manual analysis considering the top-12 agglomerations as ranked by the flood criterion (Section 4.3.5).

3-Prioritization of agglomerations with scoring criteria (JSpIRIT). We executed *JSpIRIT* to apply automatically all the scoring strategies (criteria) from Section 4.3, one by one, on the agglomerations detected in activity 1. We configured the relevance values for the agglomeration types. However, we relied on the concerns, scenarios, mappings and system versions provided by the application experts. As a result, the agglomerations were ranked according to their scoring value in decreasing order. We focused on analyzing the top-7 rankings for each system, as those high-priority agglomerations would represent the initial focus of the developer's attention.

As an example, Figure 4.8 shows the ranking of agglomerations for MM as produced by *JSpIRIT* with the flood criterion (rows), and the associated design problems (columns) determined from the ground truth. Note that cell 2f (smells within a component agglomeration based on DC for *Controller* intersecting with Ambiguous Interface in *Controller*) corresponds to the situation of Figure 4.2. Also note that the smells within a component agglomeration in SmsMessaging (row 7) has no association to design problems, even when it is relatively high in the ranking. This is a case of a false positive. which can be due to variations in the number of smells of the agglomerations across system versions, as detected by the flood criterion. In other cases, like the hierarchical agglomeration based on FE for *AlbumData* (row 11), the agglomeration is ranked low in spite of being related to four design problems. This situation can be explained by the fact that the smells of the agglomeration remained almost constant over time.

4-Computation of correlations: For HW, MM and S_{DB}, once a given scoring strategy was applied on the agglomerations, we measured the correlation between the ranking generated by *JSpIRIT* and the reference ranking (from the ground truth). To do so, we applied the Spearman's correlation coefficient for rankings with ties (p) (Ricci *et al.* 2011). This coefficient measures the strength of the association between two rankings. The coefficient can take values between 1 and -1. If p=1, it indicates a perfect association between both rankings. If p=0, it indicates no correlation between the rankings. If p=-1, it indicates a negative association between the rankings. Finally, values between 0.5 and 0.7 are regarded as a good correlation, while values higher than 0.7 are regarded as a strong correlation. As we did not have a ground truth for OODT, we did not use the same correlation strategy. To evaluate OODT, we looked instead at the precision of the criteria for the top-12 ranked agglomerations. With the help of an OODT architect, we analyzed whether each agglomeration (in the top-12 ranking) was related to design problems. Table 4.4 shows the correlation results computed on the three case-studies, plus the precision value for OODT (7 true positives over 12 cases).

4.5
Empirical Evaluation

In this section, we first report on the results of applying each of the five scoring criteria to the three target applications. Then, as previously explained (Section 4.4.2), we also discuss the results of applying the most effective criterion to the fourth software project, OODT, which is the largest one. Table 4.5 shows the number of design problems (reported by architects) and agglomerations

Table 4.4: Correlation results (the value for OODT is a precision and not a correlation value)

Applications	Design concerns	Modifiability scenarios	History of changes	Agglomeration flood
Health Watcher (HW)	0.01	-0.13	0.57	0.62
Mobile Media (MM)	0.53	0.01	0.34	0.77
SubscriberDB (S_{DB})	0.38	0.45	0.71	0.14
SubscriberDB ($S_{DB_{v2}}$)	0.1	0.08	0.51	0.6
Apache OODT	n/a	n/a	n/a	*0.58*

Table 4.5: Design problems and code-smell agglomerations for the 4 applications

	HW	MM	S_{DB}	OODT
#Design problems	61	41	60	n/a
#Agglomerations	11	16	22	431

(identified by *JSpIRIT*) in each system. On the one hand, as suggested in recent studies (Oizumi *et al.* 2016, Oizumi *et al.* 2015), we confirmed that the use of the agglomerations helped to discard hundreds of (non-agglomerated) smells that had no relationship to design problems. On the other hand, up to 60% of the agglomerations had no relationship to design problems, thereby confirming the need for defining and assessing the effectiveness of alternative prioritization criteria. Therefore, in the following subsections, we carefully analyze the correlation results for each scoring criterion and discuss their effectiveness to indicate locations of design problems. We also derive additional insights after inspecting all the prioritized agglomerations.

4.5.1

Does Agglomeration Relevance Help?

For the three applications, we analyzed several settings of this criterion. Specifically, we analyzed the cases in which an agglomeration type has a bigger relevance than another one, and vice-versa (e.g. by setting relevance values to different values in the range [0..1]). Table 4.6 shows the results for the different applications. We could not find any agglomeration relevance setting with a strong correlation for the three applications (correlations higher than 0.4 are considered significant). Moreover, we only found 3 settings with a correlation higher than 0.5 but only for HW. The lack of a strong correlation means

Table 4.6: Correlation results for agglomeration's relevance

Smells within a component	Smells in a hierarchy	Correlation HW	Correlation MM	Correlation S_{DB}	Correlation S_{DB_v2}
0.1	0.3	0.34	-0.02	-0.46	-0.48
0.2	0.3	0.51	0.37	-0.07	0.08
0.3	0.2	-0.02	0.36	0.23	0.34
0.3	0.1	-0.34	0.02	0.46	0.49
0.1	0.2	0.02	-0.36	-0.23	-0.34
0.2	0.1	-0.51	-0.37	0.07	-0.08
0.1	0.2	0.57	0.21	-0.48	-0.37
0.1	0.1	-0.28	-0.43	-0.09	-0.22
0.2	0.1	-0.18	0.21	0.36	0.43

that this criterion is not enough to prioritize agglomerations related to design problems. This happens because there is no specific agglomeration type that dominates the manifestation of design decay in the implementation of the analyzed systems.

4.5.2 Do Design Concerns Help?

The design concerns were provided by the system architects. Our goal was to check whether the scoring criterion (Section 4.3.2) would work with a minimal amount of design information, which is usually part of either the project documentation or the architects' mindset. The architects defined: (i) nine design concerns for HW – their mappings encompass around 100 classes in the program, which cover 74% of the total number of classes), (ii) seven design concerns for MM – their mappings include 65 classes (84% coverage), and (iii) five concerns for S_{DB} – their mappings subsume 45 classes (30% coverage).

After applying this criterion, *JSpIRIT* ranked the agglomerations according to their number of concerns. All the agglomerations were related to at least 3 design concerns. As shown in Table 4.4, only MM had a moderate correlation with this criterion (correlation of 0.53 with p-value = 0.03471); the correlations for HW and S_{DB} turned out low. The correlations were low because the agglomerations with the largest number of design problems were not ranked first. The reason for these low correlations is that, albeit agglomerations were often related to design problems, this criterion gave the highest scores to agglomerations that were not related to design problems. Developers would need to inspect more agglomerations in the ranking to find design problems. The use of this criterion would require more effort than the other criteria to find the first spots of design problems in the source code.

This criterion had the worst results in the HW system. In this system, the first two agglomerations ranked were related to 13 problems. However, the agglomerations ranked third, fourth and fifth were not related to design concerns. In the case of the HW system, 11 agglomerations were found, but only 5 of such agglomerations were related to problems. Therefore, this result does not necessarily represent a negative result because, on average, the developer would need to approximately inspect two agglomerations for finding at least one agglomeration related to design problems. Moreover, we found that certain agglomerations in the ranking tend to concentrate on most of the design problems. For example, in the case of the HW system: two agglomerations were related to 14 problems, two with 13, and one agglomeration with 7 problems.

Concluding, we observed that the successful use of design concerns (as a criterion) depends on the completeness and coverage of the list of design concerns provided by the developers. In fact, MM was the system that had the mappings with the highest coverage (84%) and the highest correlation (0.53). Also, we observed that this criterion worked well in systems (e.g. MM) where most problems were caused by the poor modularization of design concerns.

4.5.3
Do Modifiability Scenarios Help?

To evaluate this criterion, we described the change-related properties of three target application with the help of experts. To perform this task, besides the experts' knowledge, we used all design documentation that was available. For HW, 4 scenarios involving 25 classes were defined, which is 20% of the total number of classes. Regarding MM, 4 scenarios involving 10 classes were defined, resulting in 13% of coverage. Finally, 3 scenarios involving 10 classes were described for S_{DB}, achieving a coverage of 7. A complete list of the modifiability scenarios is provided in our supplementary material [3]. The importance values of the different scenarios were defined by the application experts.

A high correlation of design problems with modifiability scenarios was expected, because design problems are often introduced during source code changes and, modifiability scenarios represent planned changes in the system. However, we obtained poor results with this criterion (Table 4.4), with negative

[3]https://bit.ly/2E7kOAG

or low correlations. The reason is that the agglomerations related to scenarios are not necessarily the agglomerations related to design problems. For example, in the case of HW, 7 (out of 11) agglomerations are related to modifiability scenarios. While the agglomerations related to 14 design problems are ranked third and fourth (i.e., they are related to scenarios), the agglomerations related to 13 and 7 problems are ranked in the last position since they are not related with any scenario (they are tied).

These results seem to contradict our initial intuition. Therefore, we performed further analysis to understand them. In this analysis, we observed that the designs of our target applications were designed, from the outset, for satisfying those modifiability scenarios. Consequently, the modifiability scenarios had low influence on the introduction of design problems, which may explain the poor results presented by this ranking criterion. Following this idea, this ranking criterion may present better results when probed with modifiability scenarios that were not anticipated in the design. This would reveal code-smell agglomerations related to changes not supported by the design.

4.5.4
Does Change History Help?

To compute the rankings using the history criterion (Section 4.3.4), we loaded in *JSpIRIT* previous versions of the analyzed systems. In particular, we analyzed all the versions available for each application, namely: 10 versions of HW, 8 versions of MM and 15 versions of S_{DB}. In this case, we were able to find a moderate correlation for HW (0.57 with p-value = 0.06713) and a strong correlation for S_{DB} (0.71 with p-value = 0.00021). Also, we obtained a positive correlation for MM (Table 4.4). These results mean that the agglomerations located in the classes that changed often during the history represent sources of design problems in the implementation. We observed that the classes realizing agglomerations related to design problems experienced more changes during their history than the agglomeration classes that were not affected by those problems. For instance, in the case of HW, after applying this criterion, the agglomeration ranked first by *JSpIRIT* is related to 7 design problems, while the agglomerations ranked second and third are related to 14 problems. Regarding the agglomerations related to 13 problems, they were tied in the sixth position. Therefore, we observed that the consideration of history information improves the prioritization of design problems as compared to the use of design concerns, presented in the previous sub-section.

However, the use of the change history criterion was also not effective to prioritize design problems in all software projects. Recent studies (Wong *et al.* 2011) have suggested that anomalous source code, whenever it is frequently changed, indicates the presence of major design problems. We found this might be true in certain systems and, therefore, this factor helps to identify design-relevant code smells. However, this is not always the case, as captured by the low correlation (0.34) in MM. In this system, several design-harmful agglomerations were not often touched by changes. Moreover, the success of the history criterion depends on having several versions to be processed.

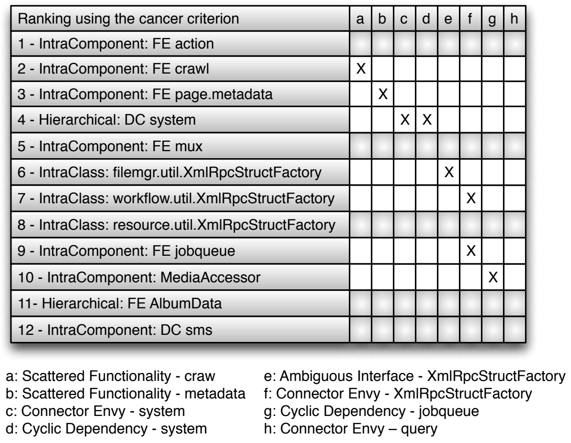
4.5.5

Does Agglomeration Flood Help?

Overall, the use of agglomeration flood was the best-performing criterion in the context of our dataset. As shown in Table 4.4, we obtained strong correlations for HW and MM. At first, we obtained a low correlation in S_{DB}. However, while understanding the reasons for this low correlation, we realized there was an issue to be addressed in the S_{DB} artifacts. When examining the design blueprints provided by the system architects, we found out that the blueprint of S_{DB} was inconsistent with the source code (Guimaraes, Garcia and Cai 2014). By inconsistent, we mean that the blueprint was an "ideal" design model of the application, but it was not faithfully implemented in the source code. In fact, we computed a consistency metric (Guimaraes, Garcia and Cai 2014) for HW, MM, and S_{DB}. We found that the HW blueprint had 89.6% of consistency, the MM blueprint had a 67.9%, and the S_{DB} blueprint had just a 54.5%.

For this reason, with the help of an S_{DB} architect, a new, more realistic blueprint called S_{DB_v2} was created, which had a consistency of 77.3%. In this case, the architect found 11 critical design problems. Then, we re-computed the reference ranking of this application and ran again the scoring criteria using *JSpIRIT*. Then, we observed a significant improvement in the correlation for the flood criterion (0.6 with p-value=0.00315), that is, a moderate correlation. As shown in Table 4.4, the correlation values for the remaining criteria decreased in S_{DB_v2}. In the case of the change history criterion, the correlation was still acceptable with the adjusted blueprint. Nonetheless, this was not the case for the criterion of design concerns that dropped to 0.1 (Table 4.4). These results indicate that the correlation values are sensitive to how the blueprints are defined. Therefore, to get the best results of this criterion, developers also need to rely on blueprints of the implemented design rather than on blueprints of the planned design.

As agglomeration flood was consistently the best-performing criterion in all



Ranking using the cancer criterion	a	b	c	d	e	f	g	h
1 - IntraComponent: FE action								
2 - IntraComponent: FE crawl	X							
3 - IntraComponent: FE page.metadata		X						
4 - Hierarchical: DC system			X	X				
5 - IntraComponent: FE mux								
6 - IntraClass: filemgr.util.XmlRpcStructFactory					X			
7 - IntraClass: workflow.util.XmlRpcStructFactory						X		
8 - IntraClass: resource.util.XmlRpcStructFactory								
9 - IntraComponent: FE jobqueue						X		
10 - IntraComponent: MediaAccessor							X	
11- Hierarchical: FE AlbumData								
12 - IntraComponent: DC sms								

a: Scattered Functionality - craw	e: Ambiguous Interface - XmlRpcStructFactory
b: Scattered Functionality - metadata	f: Connector Envy - XmlRpcStructFactory
c: Connector Envy - system	g: Cyclic Dependency - jobqueue
d: Cyclic Dependency - system	h: Connector Envy – query

Figure 4.9: Matrix of (top-12) ranked agglomerations for OODT versus related design problems (‘a’ to ‘h’)

the three systems, we also applied it to OODT. Our goal was to check whether this criterion would scale well to very large systems, such as OODT. Following the procedures described in Section 4.4.3, we analyzed the top-12 ranked agglomerations in OODT (Figure 4.9). In this analysis, we observed a moderate precision for design problems: 7 out of 12 agglomerations were true indicators of design problems. Also, three of the top-4 agglomerations were related to one or more design problems. Therefore, the results for OODT suggest that the proposed criterion can work well for large software projects in terms of “circumscribing” the search for design problems. For instance, consider the OODT’s *crawl* component. In OODT, *crawl* is responsible for listing the contents of a staging area and submitting products to be persisted by the *file management* component. However, as presented in Figure 4.9, there is an agglomeration indicating the presence of a design problem named Scattered Feature. This design problem takes place when a design concern is scattered across multiple components and some of those components are responsible for other independent concerns. This problem occurs in *crawl* because functionality related to the extraction of metadata from products, which should be done by the *metadata* component, is mixed with the crawling functionality. The Scattered Feature problem may affect aspects like maintainability and reusability. This happens because when the shared concern needs to be changed, all the components that realize it should be updated and tested. Therefore, the metadata concern should be better modularized by the *metadata* component.

The only top-ranked agglomeration unrelated to any design problem is a smell within a component agglomeration, found in the *action* component. In this agglomeration, the number of Feature Envy (FE) smells grew over time. However, this agglomeration turned out to be a false positive when judged by the OODT architect. The rationale is that the *action* component was responsible for parsing command-line options, and thus it was expected to depend on several other classes for this task. The ranking also exposed several FE-related agglomerations, because this smell was more prevalent in the OODT versions than other types of smells. Thus, we consider that this result does not undermine our conclusions.

4.5.6

Overall Conclusion

After analyzing the results of the five scoring criteria the answer to **RQ1** is that: (i) we cannot reject $H1_0$ for the first four criteria since none of them sufficed to rank correctly at least half of the agglomerations across all the systems, and (ii) we can reject $H1_0$ for the fifth criterion, the agglomeration flood criterion, since it ranked correctly more than half of the agglomerations in all the systems, including OODT. The use of the latter would help developers to find most design problems in all the systems with less effort. Developers would still need to inspect each ranked agglomeration and discard the irrelevant ones. However, we found that they could discard more than 500 code smells in their analysis. If developers analyze all these individual code smells, they would need to carefully inspect dozens or hundreds of smells to eventually find a partial source of a single design problem.

Furthermore, even when the detection might lead to some false positives, the automation of the criteria with *JSpIRIT* contributes to significantly reducing mistakes and manual effort of developers. With existing solutions or tools to detect code smells, developers would have to investigate the design information, code smells, and all their relationships, to luckily find key design problems. Alternatively, developers could rely on recent tools for detecting design smells in the code, although their detection capabilities for different design problems remains to be investigated (Azadi, Fontana and Taibi 2019).

When analyzing why the agglomeration flood was consistently the best criterion, we observed an interesting phenomenon affecting most of the design problems: groups of smells flocking together tend to better indicate the presence of design problems, and these groups tend to be increasingly connected with additional new smells when changes are made in the source code over time. This phenomenon is often caused by a bad design decision in early system versions. This finding can be illustrated by the misuse of Controllers in the MVC design of MM. In principle, the architects decided to rely on the use of a single Controller instead of multiple Controllers (Section 4.2.2). There were only three smells as part of the agglomeration affecting Controllers in the first version. These smells were located in the BaseController class. In subsequent versions, this agglomeration was being “expanded” to several code elements, including those located in BaseController subclasses and clients. The newly-introduced smells in the existing agglomeration were all directly caused by the harmful constraint of having only a single Controller. Therefore, inter-related smells in the code of evolving software systems (i.e., flood criterion) tend to be good indicators of design problems.

4.5.7
Threats to Validity

In this section, we present potential threats to the validity of our study and how we tried to mitigate them.

Internal and External Validity. An internal threat is related to the quality of the mappings between design problems and code elements. We used a consistency metric (Guimaraes, Garcia and Cai 2014) to make sure that the design specification reached a minimum quality. Also, for each target application, we validated with system experts all the responsibilities and design components realized by the code elements in the different system versions. A threat related to the criteria was about the mapping of concerns to code and the selection of the system versions. Also, the usage of LENOM as the main metric for the history criterion can introduce bias, because some kinds of changes are insensitive to LENOM. The main threat to external validity is that the applications analyzed were relatively small with few instances of code smells and agglomerations. We mitigated this threat by analyzing the flood criterion in the context of Apache OODT. Unfortunately, performing a complete analysis in larger applications (like OODT) is not always viable because an expert must manually analyze the source code and the blueprints to find the design problems. Another threat is associated with possible errors in the detection of the code smells and agglomerations. There is a possibility that the metric-based detection rules of JSpIRIT might have identified false-positive smells. We mitigated this threat by applying the same metric thresholds proposed by Lanza and Marinescu (Lanza and Marinescu 2006), which had been also used in previous work (Vidal, Marcos and Díaz Pace 2014b, Vidal *et al.* 2018, Guimaraes *et al.* 2018).

Construct and Conclusion Validity. As for construct threats, we can mention possible errors introduced in the generation of the reference ranking. We partially mitigated this imprecision by involving the original architects and developers in the inspection process. For all target applications, architects with previous experience on the detection of design problems and code smells, validated and refined the list of problems. The main threat to conclusion validity refers to the number and characteristics of the target applications. We are aware that a higher number of applications are needed for generalizing our findings. However, the information required to conduct this kind of study can be difficult to obtain. For instance, the activities of identifying and validating design problems are highly dependent on having the original personnel available. To account for this threat, we selected applications with different sizes, purposes, and domains. The applications had different design styles and involved a different set of design problems (with a minor overlapping). Finally, to mitigate the risks associated with the difficulty in obtaining the required information, we selected applications in which we had access to the original developers or expert architects. Also, we selected applications having design information available in documents, in addition to their source code, to validate the implementation with the design information.

4.6
Study with Novice Developers

In the previous section, we reported an empirical evaluation of different prioritization criteria for agglomerations. This evaluation showed the agglomeration flood was consistently the best criterion among the 5 criteria. Nevertheless, we are uncertain whether this prioritization criterion would be useful in practice. In this section, we present an experiment about the prioritization of agglomerations by human subjects. This experiment can help us to understand when and why the agglomeration flood may be useful in practice. The remainder of this section is organized as follows. Section 4.6.1 presents the design of the experiment. Section 4.6.2 gives the results, while Section 4.6.3 discusses the main findings. Finally, Section 4.6.4 presents threats to validity.

4.6.1
Study Settings

To complement the previous empirical evaluation, we conducted an experiment with advanced students of Software Engineering to answer the following research question (**RQ2**): *Does the agglomeration flood criterion rank smell agglomerations in a similar way as subjects do?* RQ2 applies the criterion with the best performance as determined from RQ1.

To achieve our goal, we performed this experiment in the context of a course on Software Evolution and Maintenance taught at UNICEN university. The experiment was run off-line (i.e., in a laboratory under controlled conditions). The subjects were undergraduate students of Systems Engineering in their fourth and fifth years at the university. All students had previous experience with Java and object-oriented programming. Also, all of them had attended the course lectures, a tutorial on code smells, agglomerations, and a laboratory where they practiced how to refactor different kinds of smells. For these reasons, it is possible to assume that their experience in code smells and agglomerations was, in general, similar. Although the experiment was conducted with novice developers, developers with this level of experience are often contributors to open-source projects, including tasks of code and design reviews. In this context, it is reasonable to assume that these kinds of tasks could be performed by novice developers.

The experiment was run over SportsTracker,[4] which is a Java system for recording sport activities. This application is open-source and it has been developed over the last 13 years. The analyzed version of SportsTracker has around 23K lines of Java code and 183 classes. A first run of *JSpIRIT* on SportsTracker reported 99 smells and 13 agglomerations. SportsTracker was selected for the experiment because it is a well-documented mid-size application. This helped the subjects to easily understand the source code and, as a consequence, to focus on the experimental tasks.

A total of 20 students participated in the experiment. Because of time constraints, we randomly selected five agglomerations of those identified by *JSpIRIT* in SportsTracker. Then, we asked each participant to analyze and rank the selected agglomerations. Each participant had to identify possible design problems in the source code related to each agglomeration, and then rank them according to the probability that the agglomeration could generate a problem of critical maintenance. The list of agglomerations instances was presented to each participant in random order. Additionally, we made available the complete source code of the SportsTracker project to the participants. Participants were allowed to spend as much time as they needed to complete the task. At the end of their analysis, we asked each participant to fill out a form, which is available in our complementary material, using a scale ranging from 1 to 5 – being 1 the most critical agglomeration.

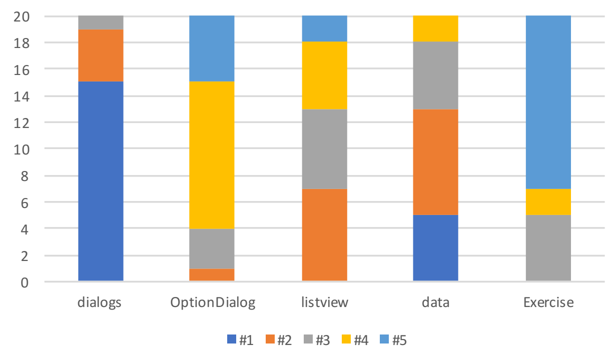


Figure 4.10: Rankings given to each agglomeration by the developers

4.6.2
Results

Figure 4.10 shows the number of participants that gave a specific ranking for each agglomeration. A total of 15 participants indicated the dialogs agglomeration as being the most critical one. Moreover, 19 out of 20 participants ranked this agglomeration in the first or second positions. Similarly, most participants indicated the data agglomeration as being critical: 5 ranked it in the first position and 8 in the second one. Also, the participants showed some agreement regarding which agglomerations were the less critical ones. In the case of Exercise, 15 out of 20 participants ranked this agglomeration in the fourth or fifth positions. Similarly, OptionDialog was ranked fourth or fifth by 16 out of 20 participants. There were mixed opinions for listview: 7 participants ranked it in the second position, and 7 participants ranked it in the fourth or fifth positions (the remaining 6 participants ranked it in the third place).

To measure the level of agreement between participants, we used the Kendall coefficient of concordance (Kendall's W). Kendall coefficient ranges from 0 (no agreement) to 1 (total agreement). After running the test, we obtained W=0.65 with p-value=1.379e-10. This value rises to W=0.739 (p-value=1.278e-09) when we do not consider the listview agglomeration. Thus, we can say that a high level of agreement existed for the remaining agglomerations.

To answer to **RQ2**, we need to compare the participants' rankings with those generated by *JSpIRIT*. When considering only the selected agglomerations, *JSpIRIT* generates the following ranking:

1. dialogs
2. OptionDialog

[4]http://www.saring.de/sportstracker/index.html v 5.7.0

Subject	1	2	3	4	5	6	7	8	9	10	11	12
W	0.55	0.85	0.55	0.75	0.55	0.8	0.85	0.85	0.7	0.65	0.6	0.8

Subject	13	14	15	16	17	18	19	20	Avg.	Std. Dev.
W	0.85	0.85	1	0.65	0.85	0.85	0.95	0.8	0.765	0.13

Table 4.7: Kendall concordance coefficient with JSpIRIT's ranking

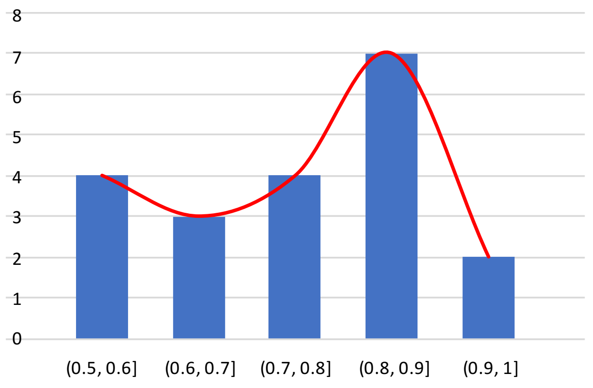


Figure 4.11: Kendall W correlations between developer's ranking and JSpIRIT's ranking

3. data

4. listview

5. Exercise

Table 4.7 lists the Kendall coefficients obtained after the comparison. Since the goal of this study is to empirically test if the flood criterion ranks agglomerations in a similar way as developers, we state the null and alternative hypotheses as follows:

- H_0: the mean of the Kendall coefficients is less or equal than 0.7.
- H_1: the mean of the Kendall coefficients is greater than 0.7.

We choose the threshold of 0.7 for our hypothesis because it is a value that could be interpreted as a strong agreement. To test our hypothesis, we used the Wilcoxon test. After running the test, we can reject H_0 with a one-tailed test with a probability of error (or significance level) $\alpha = 0.05$ and a p-value of 0.007126. This means that there is enough statistical evidence to claim that the mean of the Kendall coefficients is greater than 0.7. This finding is partially explained by the fact that 13 out of 20 W values are in the range 0.7-1 (Figure 4.11). Along this line, we can answer our RQ positively by saying that the agglomeration flood criterion ranks the agglomerations in a similar way as the subjects do.

4.6.3
Discussion

As presented in the previous section, most participants of the study ranked the agglomerations similarly to the ranking provided by *JSpIRIT* using the flood criterion. Therefore, we have evidence that ranking agglomerations with the flood criterion may help developers to focus the analysis on agglomerations that are more likely to indicate design problems. However, the statistical tests cannot reveal why the rankings of participants are similar to the flood criterion ranking. Thus, to provide a deeper understanding of this matter, we conducted a manual analysis of the five agglomerations considered in this study. In this analysis, we focused on understanding the characteristics that increase the chances of an agglomeration to indicate a design problem. We investigated (i) the information provided by participants to justify their rankings, (ii) the algorithm used in our flood criterion, and (iii) the internal characteristics of each agglomeration (e.g. type of each smell, number of smells, implemented functionalities, etc.).

Growing agglomerations as indicators of Feature Overload. Starting with the highest-ranked agglomeration, we analyzed the *dialogs* component. This component is affected by a *Smells within a component* agglomeration. Some classes from the *dialogs* component, such as class *OverviewDialog*, for example, are affected by smells like Brain Method, Dispersed Coupling, Intensive Coupling, and Feature Envy. These smells indicate that, despite being GUI classes, classes in the *dialogs* component implement functionalities that should be modularized in another component. For example, method *addExerciseTimeSeries* from class *OverviewDialog* – which is affected by the Brain Method and Dispersed Coupling smells – calculates exercise values such as distance, duration, and average speed. This method is implementing a business rule that should not be coupled to the GUI implementation. As a result, the *dialogs* component is overloaded with GUI functionalities and business functionalities. This design problem is known as *Feature Overload*.

The flood criterion was able to prioritize this agglomeration as being a strong indicator of design problems because classes overloaded with two or more concerns tend to aggregate more smells as the system evolves. This happens because multiple functionalities induce the implementation of bigger classes and bigger methods (Brain methods). Moreover, the evolution of components affected by this type of design problem introduces multiple dependencies that would not exist if the component was not overloaded with functionalities.

Those undesired dependencies are often linked to smells like Feature Envy and Intensive Coupling.

Are bigger agglomerations strong indicators of design problems? The *data* agglomeration, which is a *Smells within a component* agglomeration, was highly ranked by several participants of our experiment. However, there is no design problem in the SportsTracker system involving classes of this agglomeration. Thus, after a qualitative analysis, we observed that the size of agglomerations – i.e., the total number of code smells – plays an important role both for manual and automated prioritization of agglomerations. The bigger an agglomeration is, the more impressed a developer may become. As a result, she may think that bigger agglomerations are strong candidates to indicate a severe design problem.

A similar effect may occur with prioritization criteria like the flood criterion. Along with the evolution of a system, bigger agglomerations may aggregate more and more smells. As a consequence, such agglomerations are considered the most relevant ones according to the heuristics defined in the flood criterion (Section 4.3.5). At first, it is acceptable to expect any big agglomeration as being a strong indicator of design problems. Nevertheless, an agglomeration may be overloaded with false positive code smells, which are code smells that do not contribute to the identification of any design problem. Therefore, the prioritization criterion would fail in prioritizing the most relevant agglomerations, since the results of a criterion – such as the agglomeration flood – are highly influenced by the number of false positive smells. This is also important from the perspective of practitioners, as they may lose confidence in a tool that prioritizes agglomerations with many false positives. In fact, according to Sousa et al. (Sousa *et al.* 2018), developers tend to lose confidence in symptoms that, at least once, failed to reveal design problems. Therefore, a reduction of false positives, which is a challenging research problem, must be achieved to increase the practical impact of our prioritization criteria.

4.6.4

Threats to Validity

The first threat to validity is about the sample of participants. We selected a total of 20 students, which may not be enough for generalizing our results. Moreover, the lack of experienced programmers among participants might have also affected the validity of our results. We tried to mitigate this threat by only selecting students with previous experience with Java. Also, we conducted training sections with all participants about fundamental concepts, such as code smells and design problems.

Another threat is related to the particular software project that we used for running the experiment. The tasks may have been hampered by the lack of knowledge of the participants about the project. We tried to mitigate this threat by selecting a small and low complexity system.

Finally, there is a threat related to possible misunderstandings during the study. We mitigated this threat by preparing the participants before the experiment. We explained how the experiment would proceed and we asked them to clarify all doubts before starting with the experiment.

4.7 Concluding Remarks

As far as we are aware of, no previous work supports the prioritization of smell agglomerations to assist developers to focus on a limited set of potential sources of design problems. The prioritization is based on 5 scoring criteria that have the goal of ranking first the agglomerations that likely indicate locations of design problems. To rank agglomerations, the criteria explored different types of information that are typically available in software projects, including (partial) lists of design concerns, (approximate) component structure, and change history. As a proof-of-concept, the scoring criteria were implemented in the *JSpIRIT* tool.

To assess and compare the effectiveness of the prioritization criteria, we conducted a fist study based on the analysis of four systems, one of them with a very large size (OODT). In this study, we found that, although the effectiveness of most criteria depended on the characteristics of each project, the use of the agglomeration flood criterion was consistently effective across all the projects, including OODT. The other criteria did not present a good correlation in certain projects. For instance, in two projects, the use of design concerns alone did not suffice to pinpoint agglomerations related to design problems. We observed that the criterion based on design concerns is effective only in projects where developers can provide complete coverage of the design concerns. In our dataset, this was the case of MM, where the specification of the design concerns covered 84% of the classes in the source code. However, even for this project, it would be useful to also rely on the use of the flood criterion as: (i) it had a strong correlation with design problems in this system, and (ii) it helped to spot a different list of design problems, not detected with the criterion of design concerns. Along this line, another experiment using the flood criterion showed that the ranking provided by the criterion for a given application is similar to the ranking given by a group of developers.

In a second study with subjects, we observed that different types of design problems may require different prioritization criteria. The flood criterion, for example, presented good results for the identification of design problems like the Feature Overload. However, this criterion is unable to capture many other types of design problems. Our analysis also revealed that some agglomerations are overloaded with false positives, which impacts both automated and manual prioritization. Thus, we conjecture that developers need better tool assistance to analyze agglomerations. This can be achieved by (i) improving the underlying techniques used for detecting code smells; and (ii) providing multiple and diverse filtering and prioritization criteria for capturing different types of design problems.

We believe that these findings have practical implications. For instance, the choice among the relevance-based, concern-based, scenarios-based, history-based and flood-based criteria has tradeoffs (Vidal, Marcos and Díaz Pace 2014b). The usage of the criterion of design concerns may be preferred in several cases, even at the cost of being an inferior indicator of design problems, because problems can be spotted already in the first versions when they are usually easier to be dealt with. Further studies could investigate how the use of two or more scoring criteria could be combined to get better indicators of design problems. In fact, in our study, using design concerns and agglomeration flood in conjunction would lead to the identification of almost 90% of all design problems affecting the three first systems: HW, MM, and S_{DB}. Further work can investigate, for example, which combination of criteria tends to be the most effective across a larger sample of projects. For instance, the fact that both the modifiability scenarios and the system history are based on code changes suggests that scenarios affecting previous versions of a system can impact subsequent versions and their agglomerations (which is hinted by the performance of the flood criterion). As regards history-based information, we envision the usage of metrics other than LENOM as an alternative proxy for maintenance effort. We intend to investigate other types of code-smell agglomerations (Oizumi *et al.* 2016) to check if they lead to better correlations.

Finally, we would like to mention that the identification of agglomerations as likely indicators of design problems brings the issue of system refactoring to alleviate those problems. However, even when developers know about a critical agglomeration, envisioning a refactoring solution for it is often not trivial from a design perspective. Along this line, appropriate refactoring tools for design smells still need to be investigated. Anyway, if the practical decision is not to refactor the agglomeration, developers can still perform careful testing of the affected classes, avoid adding more logic to them, and also plan alternative evolution paths for the system.

5 On the Density and Diversity of Design Problem Symptoms in Refactored Classes: A Multi-Case Study

Refactoring is a software development activity that is intended to improve dependability-related attributes such as modifiability and reusability. Despite being an activity that contributes to these attributes, deciding when applying refactoring is far from trivial. In fact, finding which elements should be refactored is not a cut-and-dried task. One of the main reasons is the lack of consensus on which characteristics indicate the presence of design problems.

Chapters 3 and 4 showed that the use of multiple symptoms (so called agglomerations in previous chapters) is an effective strategy for finding DPs. Thus, in this chapter, we evaluated whether the density and diversity of multiple automatically detected symptoms can be used as consistent indicators of the need for refactoring. To achieve our goal, we conducted a multi-case exploratory study involving 6 open source systems and 2 systems from our industry partners. For each system, we identified the classes that were changed through one or more refactorings. After that, we compared refactored and non-refactored classes with respect to the density and diversity of design problem symptoms. We also investigated if the most recurrent combinations of symptoms in refactored classes can be used as strong indicators of design problems.

Our results show that refactored classes usually present higher density and diversity of symptoms than non-refactored classes. However, refactorings that are performed by developers in practice may not be enough for reducing design problems, since the vast majority had little to no impact on the density and diversity of symptoms. Finally, we observed that symptom combinations in refactored classes are similar to the combinations in non-refactored classes. Thus, combinations alone may not be enough for consistently finding DPs. Based on our findings, we elicited an initial set of requirements for automatically recommending refactorings. Such requirements are based both on the results presented in this chapter and on results obtained in Chapters 3 and 4.

The results of this study were published in a series of papers. First, we published a short paper at the negative results track of the *International Conference on Program Comprehension* (ICPC) (Eposhi *et al.* 2019). This paper included only an initial investigation with our industry partners. Next, we published a full paper in the *International Symposium on Software Reliability Engineering* (ISSRE) (Oizumi *et al.* 2019). The ISSRE paper contains all the results presented in this chapter. Finally, we also published a position paper in the Doctoral Symposium track of ISSRE (Oizumi 2019). Such a paper was focused in the requirements for refactoring recommendation techniques (Section 5.5).

5.1
Introduction

As software systems evolve, they can go through changes that can lead to their design problems. Unfortunately, the design problems can lead software systems to the discontinuation or at least either significant maintenance effort or the complete redesign (Godfrey and Lee 2000, Gurp and Bosch 2002, MacCormack, Rusnak and Baldwin 2006, Schach *et al.* 2002). This design problem occurs when stakeholders make decisions that have a negative impact on dependability-related attributes (Li, Avgeriou and Liang 2015, Lim, Taksande and Seaman 2012, Besker, Martini and Bosch 2017). An example of this scenario is when a stakeholder decides to create a common system interface to provide access to different unrelated services. This decision is likely to harm the system maintainability and extensibility (Martin 2002).

Developers constantly have to improve the internal design of software systems to, in the worst case scenario, repair a deteriorated code. For this purpose, they have been relying on one of the most common activities applied during software maintenance and evolution: refactoring (Fowler 1999). Refactoring is a transformation in the source code design without changing the functional behavior of the system (Fowler 1999, Murphy-Hill and Black 2008b, Murphy-Hill, Parnin and Black 2012). A commonly applied refactoring tactic is known as root canal refactoring, which involves a process of exclusively applying refactorings to reduce the design problems (Murphy-Hill and Black 2008b, Murphy-Hill, Parnin and Black 2012). Despite being an activity aimed at improving dependability-related attributes of the system's design, deciding when applying root canal refactoring is not trivial (Bavota *et al.* 2014).

Developers need to know where they should refactor the source code; more specifically, they have to find first what code elements (packages, classes, methods, and the like) need to be refactored to reduce the design problems (Bavota *et al.* 2014). To this end, developers can find and monitor indicators of design problems in the source code, i.e., they need to rely on symptoms of design problems (Sousa *et al.* 2018). Code smell is an example of a symptom. It is a structure in the system implementation that represents a surface indication of design problems (Fowler 1999). An example of code smell is *Long Method*, which indicates a method that is too long to understand (Fowler 1999).

After the design problem symptoms have been found, developers can reduce the design problems by applying efactorings (Murphy-Hill and Black 2008b). Hence, based on results of our previous chapters, one might expect that developers often apply refactoring in code elements that contain either multiple symptom instances (*density*) or different types of symptoms (*diversity*). Unfortunately, there is little information and no much consensus whether the density and diversity of multiple automatically detected symptoms can be consistent indicators of the need for refactoring.

Existing studies are mostly focused in investigating the impact of any refactoring kind in the density of symptoms (Bavota *et al.* 2015), (Cedrim *et al.* 2017). However, none of them investigated the relation of refactorings with the density and diversity of symptoms. Thus, we investigated to what extent the density and diversity of symptoms indicate the need for refactoring. We also investigated whether refactoring impacts the density and diversity of symptoms.

To better understand the density and diversity of design problem symptoms, we conducted a multi-case exploratory study in which we observed the refactorings applied by developers. This study involves eight software system: six open source systems and two systems from our industry partners. For each software system, we found the classes that were changed through one or more refactorings, and then we collected the design problems symptoms in all classes from the system. After that, we compared refactored and non-refactored classes with respect to the density and diversity of symptoms. We also evaluated the impact of refactorings on the density and diversity of these detected symptoms. Finally, we investigated if the most recurrent combinations of symptoms in refactored classes are different from the recurrent combinations in non-refactored classes.

Upon data analysis, we found that refactored classes usually present higher density and diversity of symptoms than other classes. After investigating what happens with the refactored classes, we did not find a consistent reduction in the density or in the diversity of symptoms, leading us to conclude that refactorings cause little to no positive effect in design problem symptoms. In fact, the effects of refactorings are practically nonexistent in the core classes that are constantly modified. We also found that the recurrent combinations of symptoms in refactored classes are similar to the recurrent combinations in non-refactored classes. Thus, the combinations by themselves are not good indicators of design problems. Based on such findings, we elicited an initial set of requirements for automatically recommending refactorings.

5.2 Background

5.2.1 Design Problem

Design problems occurs when stakeholders make decisions that negatively impact dependability-related attributes (Li, Avgeriou and Liang 2015), (Lim, Taksande and Seaman 2012), (Besker, Martini and Bosch 2017). An example of design problems is the so called Fat Interface (Martin and Martin 2006). This form of design problems occurs when a single interface provides multiple and unrelated operations, making it difficult to use and increasing the chance of introducing defects to its clients. Due to the negative impact caused by design problems, software systems have often been discontinued or redesigned when design problems was allowed to persist (MacCormack, Rusnak and Baldwin 2006). Thus, to be able to maintain the system's quality, developers need to identify and to confirm the existence of design problems. Next, we present an example to illustrate how dependability-related attributes may be impacted by design problems.

Figure 5.1 shows a partial view of the OpenPOS system before and after a degraded design has been refactored. OpenPOS is a system that provides sales features. One of the functionalities of OpenPOS comprises the generation of payment slips. In the country where OpenPOS is used, payment slips serve for clients to make payments at any bank. Developers of OpenPOS implemented this feature in the *PaymentSlip* sub-component. To protect system information, this sub-component was strongly dependent from the *Authentication* component.

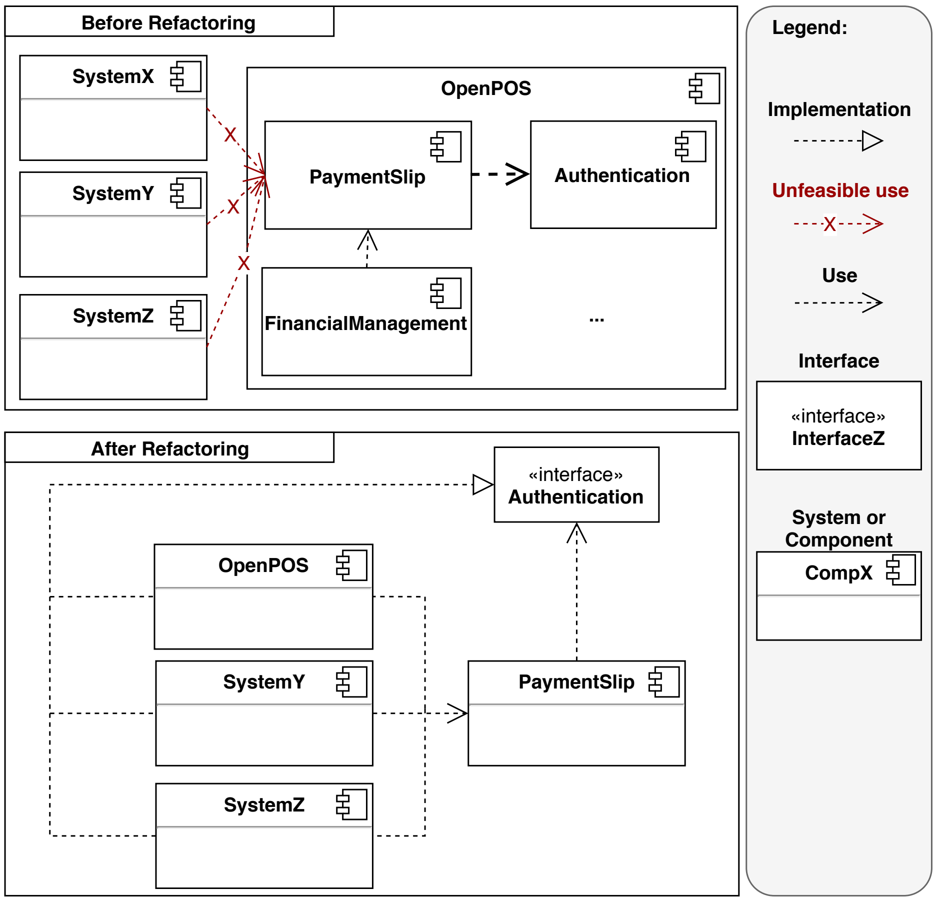


Figure 5.1: Example of design problem impacting reusability

Unfortunately, the strong dependency with the *Authentication* component led to a side effect on the reusability of *PaymentSlip* sub-component. Reusability is a sub-category of maintainability that indicates the degree to which a component can be re-used in two or more systems (ISO-IEC 25010 2011). Since *PaymentSlip* was so coupled to the *Authentication* component, it could not be reused in other systems. In this context, developers have to refactor the *PaymentSlip* sub-component to reduce the coupling with *Authentication* component. Additionally, refactoring this kind of design problems is fundamental to avoid code duplication among systems and rework. In Section 5.2.3, we will explain the design obtained after the refactoring.

5.2.2

Design Problem Symptoms

Sousa et al. (Sousa *et al.* 2018) identified five categories of symptom upon which developers frequently rely to identify design problems. Similarly to other related work (Yamashita *et al.* 2015, Macia *et al.* 2012a, Oizumi *et al.* 2016, Oizumi *et al.* 2018), they observed that developers tend to combine multiple symptoms, taking into account dimensions such as diversity and density to decide if there is a design problem or not. In this work, we selected a sub-set of two symptom categories that can be automatically detected using state-of-the-practice tools, which are the code smells and the principle violations.

Code smell is a surface indicator of possible design problems (Fowler 1999). This symptom category have been extensively investigated by different researchers (e.g., (Lanza and Marinescu 2006, Murphy-Hill and Black 2010, Yamashita and Moonen 2012, Moha *et al.* 2010)). Recent studies (Yamashita *et al.* 2015, Oizumi *et al.* 2016, Oizumi *et al.* 2018, Sousa *et al.* 2018) suggest that combining multiple code smells may improve the precision when identifying design problems. An example of code smell type is the Long Method. This type of smell usually leads to design problems related to modifiability.

In object-oriented systems, design problems usually impact object-oriented design characteristics, such as abstraction, encapsulation, modularity, and hierarchy. Therefore, the second symptom category we used comprises the **principle violations**, which are symptoms that may indicate the violation of common object-oriented principles (Martin and Martin 2006). An example of object-oriented principle is the *Single Responsibility Principle* (SRP). The SRP determines that each class should have a single and well defined responsibility in the system (Martin and Martin 2006). An example of symptom that may be used for finding SRP violations is the *Insuficient Modularization* (Suryanarayana, Samarthyam and Sharma 2014). This symptom occurs in classes that are large and complex, possibly due to the accumulation of responsibilities.

Table 5.1 shows the descriptions for the 17 types of principle violations and 10 types of code smells used in this study. The descriptions are based on the taxonomy of symptoms provided by Sharma and Spinellis (Sharma and Spinellis 2018, Sharma 2020).

5.2.3 Refactoring

Refactoring consists in transforming the source code design without changing the functional behaviour of the system (Fowler 1999). Thus, we consider that refactoring is any design software change that is aimed at improving dependability-related attributes of the system's design.

According to Murphy-Hill and Black (Murphy-Hill and Black 2008b), refactoring can be classified into two tactics, which are floss refactoring and root canal refactoring. On one hand, floss refactoring is characterized by refactoring

Table 5.1: Short description for the symptoms used in this study

Symptom Type	**Description**
Category 1 - Code Smells	
Abstract Function Call From Constructor	A constructor that calls an abstract method
Complex Conditional	A conditional statement that is complex
Complex Method	A method that has high cyclomatic complexity
Empty Catch Block	A catch block of an exception that is empty
Long Identifier	An identifier that is excessively long
Long Method	A method that is too long to understand
Long Parameter List	A method that accepts a long list of parameters
Long Statement	A statement that is excessively long
Magic Number	When an unexplained number is used in an expression
Missing Default	A switch statement that does not contain a default case
Category 2 - Principle Violations	
Broken Hierarchy	A supertype and its subtype that conceptually do not share an "is a" relationship
Broken Modularization	When data and/or methods that should have been into a single abstraction are spread across multiple abstractions
Cyclic Dependent Modularization	When two or more abstractions depend on each other directly or indirectly
Cyclic Hierarchy	A supertype in a hierarchy that depends on any of its subtypes
Deep Hierarchy	An inheritance hierarchy that is excessively deep
Deficient Encapsulation	The accessibility of one or more members of an abstraction is more permissive than actually required
Hub Like Modularization	An abstraction that has dependencies with a large number of other abstractions
Imperative Abstraction	When an operation is turned into a class
Insufficient Modularization	An abstraction that has not been completely decomposed
Missing Hierarchy	When a design segment uses conditional logic instead of polymorphism
Multifaceted Abstraction	An abstraction that has more than one responsibility assigned to it
Multipath Hierarchy	A subtype that inherits both directly as well as indirectly from a supertype
Rebellious Hierarchy	A subtype that rejects the methods provided by its supertype(s)
Unexploited Encapsulation	A client class that uses explicit type checks instead of exploiting the variation in types already encapsulated within a hierarchy
Unnecessary Abstraction	An abstraction that is actually not needed in the system
Unutilized Abstraction	An abstraction that is left unused
Wide Hierarchy	An inheritance hierarchy that is too wide

changes intermingled with other kinds of source code changes, such as adding new features and fixing bugs. The aim of floss refactoring is to keep design quality as a means to achieve other goals. On the other hand, root canal refactoring aims at exclusively reducing design problems. A root canal refactoring consists of only refactoring changes; it is not performed in conjunction with other non-refactoring changes. Thus, in this chapter, our focus is on root canal refactorings as they are explicitly aimed to reduce design problems. Thus, from now on, whenever we talk about refactoring in this chapter, we'll be referring to root canal refactoring.

To illustrate our definition of refactoring, let's return to the example of Figure 5.1. As previously discussed, multiple different systems of the same company began to require a payment slip feature. Therefore, developers were asked to remove the reusability design problems by refactoring the *PaymentSlip* sub-component. The refactoring consisted of introducing an interface for authentication. This way, each system that needs to use the *PaymentSlip* component must specify an authentication component that meets the interface specifications required by *PaymentSlip*. After refactoring the *PaymentSlip* sub-component, besides fixing the design problems, it is expected the removal of symptoms such as the *Hub-Like Modularization* (Table 5.1).

5.3 Study Design

5.3.1 Goal and Research Questions

Several studies (e.g., (Murphy-Hill and Black 2008a, Le *et al.* 2018, Xiao *et al.* 2016)) have proposed and evaluated techniques for the detection of design problems. Nevertheless, in practice, most of them are not applied by developers. One of the issues of existing techniques is the high amount of false positives (Oizumi *et al.* 2018, Macia *et al.* 2012b), which may lead developers to have little confidence in the presented symptoms. Another problem is that most techniques are based on a single category of symptom. However, according to the literature, developers may combine multiple and diverse symptoms for confirming the existence of a design problems. In this sense, there are techniques that work with multiple symptoms (Macia *et al.* 2012a, Oizumi *et al.* 2018). Still, unlike what was observed in the study of Sousa et al. (Sousa *et al.* 2018), existing techniques only combine symptoms of the same category (e.g., code smells). In addition, their efficiency to reveal classes impacted by design problems has not been exhaustively validated. Finally, there is little evidence on the impact of refactoring on symptoms such as principle violations. Thus, in this chapter, *we aim at evaluating the relation of refactorings with the occurrence of multiple and diverse design problems symptoms.* To achieve our goal, we defined the following research questions:

RQ1. Are the density and diversity of design problem symptoms in (root canal) refactored classes different from the density and diversity in other classes?

With RQ1, we aim at understanding if the design problem symptoms are denser and more diverse in refactored classes when compared to other classes. As refactorings should be applied to classes impacted by design problems (Murphy-Hill and Black 2008b), we need to know if such classes, before being refactored, present higher density and diversity of symptoms than most regular classes. Answering this question will be helpful for evaluating whether combining multiple and diverse symptoms is indeed an effective strategy for identifying and confirming the existence of design problems.

RQ2. Do classes modified by refactorings present design improvement in the medium term?

With RQ2, we want to observe whether the expected improvements of refactorings impact the density and diversity of symptoms. This question will help us to understand if symptoms disappear, decrease, or increase in the medium term after the application of refactorings. In this context, we consider medium term as being the next release after refactoring. With this research question we will also be able to better understand if the refactorings performed in practice have been effective, according to the measurement provided by the investigated symptom categories.

RQ3. Are the combinations of symptoms in (root canal) refactored classes different from the combinations in other classes?

The aim of RQ3 is to investigate whether combinations of symptoms can be used to differentiate refactored classes from other classes. In addition, this research question will help us to understand which combinations of code smells and principle violations are often refactored by the developers of our target systems. Based on the findings, it may be possible to prioritize degraded classes based on the combinations of symptoms that developers refactor more

Table 5.2: Characteristics of target systems

Name	**Platform**	**Domain**	**Size (LOC)**	**# of Commits**	**Releases**
Partners' Systems					
OpenPOS	.Net/C#	Enterprise	97,000	3,318	67, 68
UniNFe	.Net/C#	Enterprise	492,000	2,373	345, 362
Open Source Systems					
Achilles	Java	Tool	83,124	1,188	1.0-beta, 3.0.0, 5.1.0
Ant	Java	Tool	137,314	13,331	15_ 141, 163_ 170, 180_
Derby	Java	Database	1,760,766	8,135	10.3.2.1, 10.5.3.0, 10.7.1.1
Elasticsearch	Java	Engine	578,561	23,597	1.2.2, 1.5.0, 2.3.0
MPAndroidChart	Android/Java	Library	23,060	1,737	1.0.1, 2.1.0, 2.2.4
Tomcat	Java	Middleware	668,720	18,068	7.0.0-RC1, 7.0.8, 7.0.35, 7.0.57, 7.0.67, 8.5.9

often.

To answer our research questions, we conducted a case study involving multiple and diverse software systems. We collected and analyzed source code changes due to refactoring , i.e., changes that were exclusively dedicated to fixing design problems. After that, we collected multiple types of code smells and principle violations and conducted our data analysis. Next, we provide details about the target systems and about our procedures for data collection and analysis.

5.3.2 Target Systems

Table 5.2 shows the target systems of this chapter. Columns two to five of Table 5.2 show respectively the: platform, the system domain, the size in Lines of Code (LOC), and the number of commits. Column six shows the releases of each system in which we collected design problem symptoms.

Open Source Systems. As presented in Table 5.2, we selected six open source systems for this study: Apache Ant, Apache Derby, Apache Tomcat, Achilles, Elasticsearch, and MPAndroidChart. To select these systems, we first selected 50 open source systems in which we applied a set of filtering criteria (Section 5.3.3). We aimed at selecting a set of representative systems from different domains.

Partners' Systems. To make our data sample more heterogeneous, we selected two C# systems from our industry partners. The first system is OpenPOS, a desktop system that provides sales features, such as sales registration and cashier closing. UniNFe is a background service that sends and receives electronic invoices. These projects are suitable for this study because each of them presents more than one hundred classes that were refactored due to design problems. In addition, their refactorings are documented in specific refactoring tasks. Finally, we had full access to their original developers for questions and clarifications.

5.3.3 Data Collection and Analysis

We followed three main steps for data collection and analysis: (a) finding refactorings, (b) collecting information about design problems symptoms, and (c) running data analysis. Next we present details about each step.

Finding refactorings In the first step, we searched for source code changes that were exclusively intended to fix design problems. To achieve this goal, we adopted different procedures for the partners' systems and for the open source systems. To select the open source systems, we started by analyzing a database containing information about 50 open source projects. We created and validated this database in previous studies (Cedrim *et al.* 2017, Bibiano *et al.* 2019) in which we collected information about the projects' history of changes, commit messages, and performed refactorings. We used Refactoring Miner 0.2.0 (Tsantalis *et al.* 2018) to automatically detect refactorings of 11 different types. Due to space constraints, the description of refactoring types are presented in our replication package[1].

Since Refactoring Miner is unable to differentiate root canal from floss refactoring, we identified and filtered the refactorings based on the following filter: (1) the selected refactorings should be occurring in groups of two or more refactorings, and (2) the refactorings withing a group should have been detected in the same commit or in sequential commits. As a result, this filter has helped us to find refactorings that changed multiple source code structures and, therefore, had a greater chance of being refactorings. After filtering, we discarded the systems with less than 10 refactored classes since it would be a very small sample of classes. In this way, we have reduced our sample of systems to the 6 open source systems presented in Table 5.2. We also tried to apply a second filter by considering the commit message associated with each refactoring change. With this filter we would only select the refactorings for which the associated commit message contained any variation of the word refactor (e.g., refactoring, Refactor, etc). However, the resulting sample of refactorings was very small, which meant that the results would not be statistically significant.

[1]http://wnoizumi.github.io/ISSRE2019/

Thus, we decided to use this second filter only as a parameter of comparison for the results obtained with the first filter.

To find refactorings in partners' systems, we asked two original developers of each project to provide us with a list of tasks aimed at refactoring. After that, we conducted an automated search in the issue tracking system to complement the lists of tasks provided by developers. Our automated search was based on a set of keywords that are often associated with design problems (e.g., structure, interface, and duplicate). We have defined those keywords based on the analysis of task descriptions from 50 open source projects from our previously mentioned database. These keywords often occur in the description of tasks that aim at improving dependability-related attributes. After the automated search with the keywords, we, together with the two developers, analyzed the resulting list of tasks. We discarded those that could not be characterized as refactorings.

Collecting information about design problem symptoms As presented in Section 5.2, we collected two categories of design problems symptoms: code smells and principle violations. We used the Designite tool to collect these symptoms (Sharma *et al.* 2016). We selected this tool because it detects the same set of symptoms for both C# and for Java programs, thus, keeping the consistency regarding the detection strategies for both programming language. Detailed descriptions, detection strategies, and thresholds for all types of symptoms are available in our replication package.

As illustrated in Figure 5.2, we collected the symptoms in the last release of the system before refactorings and in the first release after refactorings. The releases presented in the last column of Table 5.2 are the ones that we collected the symptoms. We are aware that this makes refactoring changes to be mixed with other changes. Nevertheless, we have chosen this approach intentionally, since we wanted to evaluate if the possible design improvements caused by refactorings persist in the medium term.

Running data analysis After collecting data about tasks, source code changes and symptoms, we conducted the data analysis to answer our research questions. To answer RQ1, we divided our dataset into two groups: Refactored Classes and Other Classes (or simply, Others). The first one is composed by all classes for which we found one or more refactorings. The second group is composed by all classes in the systems that are not in the former group.

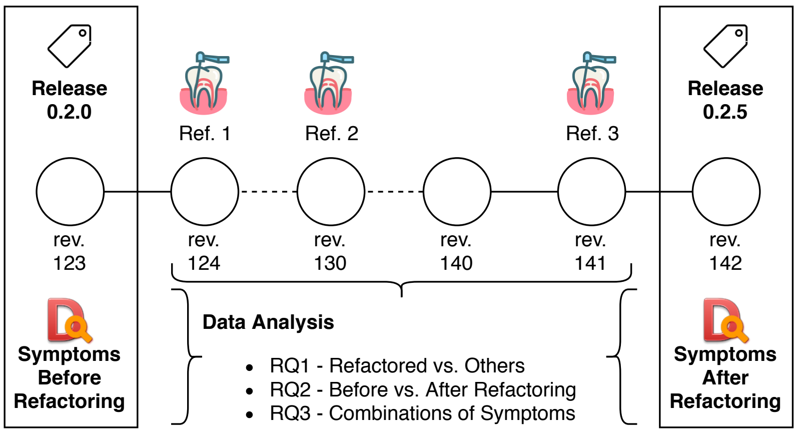

Release 0.2.0
Ref. 1
Ref. 2
Ref. 3
Release 0.2.5
rev. 123
rev. 124
rev. 130
rev. 140
rev. 141
rev. 142
Symptoms Before Refactoring
Data Analysis
RQ1 - Refactored vs. Others
RQ2 - Before vs. After Refactoring
RQ3 - Combinations of Symptoms
Symptoms After Refactoring

Figure 5.2: Collection of symptoms for data analysis

For both groups, we calculated the density and diversity of symptoms. *Density* represents the number of individual instances of symptoms occurring in a class, while *diversity* represents the number of different symptom types occurring in a class. We compared the density of both groups by computing the code smell and principle violation distributions. We compared the diversity of symptoms by calculating the distribution of symptom types quantity. As previously explained, we considered 10 types of code smells and 17 types of principle violations. Based on the collected data about density and diversity, we used the *Mann-Whitney Wilcoxon* statistical test to check whether there was a significant difference in the distributions presented by both groups.

To answer RQ2, we collected and compared the same data used to answer RQ1. The difference here is that we compared the density and diversity of classes collected in releases before and after the execution of refactorings. Hence, for this question, we only considered classes that were changed by refactorings. Additionally, with the help of developers from our industry partners, we conducted further analysis to better understand the obtained results.

Finally, to answer RQ3, we performed a threefold analysis. First, we investigated the number of classes affected by 170 pairwise combinations of code smell types with principle violation types. Second, we generated two rankings for the combinations based on the number of refactored (1st ranking) and non-refactored (2nd ranking) classes affected by each combination. Then, we applied the *Spearman's rank correlation rho* statistical test to compare both rankings. Finally, we evaluated the relevance of symptom combinations for recommending refactorings. Our rationally for combining symptoms from different categories is that they could be stronger indicators of design problems. We also tried to investigate combinations with more than two symptoms. However, those combinations were rare and not observed in more than two systems.

Conversely, combinations with only two symptoms from different categories occurred frequently.

5.4 Results

5.4.1 Density and Diversity as Consistent Indicators

Table 5.3: Mean density of symptoms in refactored classes and in others

Category	Mean Density of Symptoms															
	OpenPOS		UniNFe		Achilles		Ant		Derby		Tomcat		Elasticsearch		MPAndroidChart	
	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others	Ref.	Others
Before Refactoring																
P. Violation	0.806	0.756	2.238	1.134	1.632	1.073	1.802	1.068	1.802	1.068	1.801	1.032	1.632	1.042	1.250	1.138
Code Smell	35.843	4.102	41.642	1.415	16.775	4.578	10.985	1.810	10.985	1.810	23.72	3.731	14.475	6.380	21.62	9.317
After Refactoring																
P. Violation	0.725	0.768	2.404	1.497	1.666	1.043	1.921	1.070	1.921	1.070	1.794	1.020	1.557	1.082	1.625	1.127
Code Smell	35.462	3.765	41.357	1.719	32.055	4.463	12.156	1.840	12.156	1.840	21.339	3.691	24.242	7.440	18.687	8.475

Table 5.3 shows the mean density of symptoms in the classes of both groups (refactored and others). Each line shows, for a symptom category, the mean density of symptoms in refactored classes (Ref.) and in other classes (Others). This information is provided for each system before and after refactorings.

For code smells, we observed a notable difference when comparing refactored classes with others in all target systems. This difference indicates that the density of smells can be used as a strong indicator of design problems. For the most extreme cases (OpenPOS and UniNFe), the density of smells was more than 8 times higher in refactored classes. Analyzing the dataset, we also observed several outliers in the distribution of code smells for Others. Many of these outliers may be classes affected by design problems that were not changed in refactorings.

Principle violations, in general, were denser in refactored classes when compared to other classes. However, for all target systems the observed difference was small. The system that presented the greatest difference regarding principle violations was the UniNFe, where the density of violations was almost two times higher in refactored classes. Nevertheless, in all target systems, the density of principle violations in refactored classes (before refactoring) was higher than in other classes.

We applied the *Mann-Whitney Wilcoxon test* to check whether there was a statistically significant difference in the density distribution. Table 5.4 summarizes the results for all systems. The results related to density, in the context of RQ1, are presented in the second (for principle violation) and fourth

Table 5.4: p-values of the Mann-Whitney Wilcoxon Test for research questions RQ1 and RQ2

System	**Principle Violation**		**Code Smell**	
	Density	**Diversity**	**Density**	**Diversity**
RQ1 - refactored classes and others				
OpenPOS	**0.18**	**0.12**	<0.01	<0.01
UniNFe	<0.01	0.03	<0.01	<0.01
Achilles	<0.01	0.01	<0.01	0.01
Ant	<0.01	<0.01	<0.01	<0.01
Derby	<0.01	<0.01	<0.01	<0.01
Elasticsearch	<0.01	0.01	<0.01	<0.01
MPAndroidChart	**0.76**	**0.52**	<0.01	0.04
Tomcat	<0.01	<0.01	<0.01	<0.01
RQ2 - before and after refactoring				
OpenPOS	0.62	0.68	0.54	0.32
UniNFe	0.98	0.88	0.72	0.68
Achilles	0.72	**<0.01**	0.06	0.06
Ant	0.33	0.30	0.59	0.54
Derby	0.63	0.61	0.96	0.46
Elasticsearch	0.73	**<0.01**	0.10	**<0.01**
MPAndroidChart	0.35	**<0.01**	0.64	0.07
Tomcat	0.96	0.11	0.20	0.34

(for code smell) columns and in lines four to eleven of Table 5.4. A p-value smaller than 0.05, means that the distribution of density of symptoms in refactored classes is different from the distribution of density in other classes. The raw data and the detailed results of this statistical test are available in our replication package.

The tests showed that, for all systems, **the smell density in refactored classes was significantly different from the smell density in other classes**. On the other hand, when we ran the same test for the density of principle violations, we cannot reject the null hypothesis for the OpenPOS and MPAndroidChart systems. Therefore, the density distribution of principle violations in refactored classes may be equal to the density distribution of principle violations in other classes for some systems. For MPAndroidChart, this result is partially explained by the sample size, since we found only 18 refactored classes in this system. The explanation for OpenPOS, is the fact that its refactorings were more focused on fixing modifiability design problems. Such issues generally do not manifest themselves in the form of principle violations, since they do not affect aspects such as abstraction and hierarchy. In any case, to achieve greater generalization, the density of principle violations must be further investigated in the context of other systems.

Table 5.5: Mean diversity of symptoms in refactored classes and in others

Category	**Mean Diversity of Symptoms**															
	OpenPOS		**UniNFe**		**Achilles**		**Ant**		**Derby**		**Tomcat**		**Elasticsearch**		**MPAndroidChart**	
	Ref.	**Others**	**Ref.**	**Others**	**Ref.**	**Others**	**Ref.**	**Others**	**Ref.**	**Others**	**Ref.**	**Others**	**Ref.**	**Others**	**Ref.**	**Others**
Before Refactoring																
P. Violation	0.650	0.701	1.166	0.815	0.500	0.218	1.157	0.540	1.725	0.876	1.113	0.568	1.143	0.828	0.555	0.363
Code Smell	1.581	0.444	3.285	0.190	0.437	0.175	1.931	0.352	3.412	1.247	2.581	0.529	2.151	0.770	1.388	0.411
After Refactoring																
P. Violation	0.600	0.722	1.142	1.076	0.187	0.366	1.280	0.644	1.825	0.974	1.272	0.619	0.459	0.477	1.444	0.765
Code Smell	1.387	0.424	3.523	0.245	0.237	0.280	2.093	0.418	3.700	1.365	2.795	0.569	0.877	0.441	2.555	0.822

Diversity of symptoms is also more significant for code smells. Table 5.5 shows the diversity of symptoms for code smells and for principle violations. This table follows the same organization of Table 5.3, providing the mean diversity of each symptom category. In all systems, the diversity of code smells was significantly higher in refactored classes when compared to other classes in the systems. For refactored classes, the diversity of code smells was more than five times higher in Ant and more than seventeen times higher in UniNFe, for example. Similarly to what we observed regarding the density of symptoms, the statistical tests (3^{rd} and 5^{th} columns of Table 5.4) revealed that, for all target systems, the diversity mean of code smells in refactored classes is different from the diversity mean of code smells in other classes. However, regarding diversity of principle violations, we cannot reject the null hypothesis for the OpenPOS and MPAndroidChart systems. The rationale for explaining the diversity results in these two systems is the same as that used to explain the density results.

When we applied the second filter in the refactorings of open source systems (Section 5.3.3), the density and diversity averages remained similar. The refactored classes continued to present higher density and diversity of symptoms when compared to other classes. This second filtering made the averages of most open source systems more similar to the averages observed in the partner systems. For the Derby system, for example, the mean density of both smells and violations in refactored classes became significantly higher: 2.50 for violations and 29.25 for smells. Therefore, based on our analyses, we conclude that **the density and diversity of symptoms in refactored classes are, indeed, different from the density and diversity of symptoms in other classes.** However, for two out of eight target systems, principle violations did not show significant differences for both density and diversity of symptoms. This indicates that we should, in future work, test our hypotheses on another set of systems to verify if the results converge. In addition, the results observed here indicates that the density and diversity of code smells may be considered a more reliable indicator of design problems, according to the criteria adopted by developers to decide when to conduct refactorings.

5.4.2
Low Reduction of Symptoms After Refactoring

Table 5.3 shows the mean density of symptoms before and after the application of refactorings. It is possible to observe that refactorings caused little impact on all symptom categories. In some cases the mean value increased while in others it decreased after the refactorings. However, for most of them, there was not a significant difference. To confirm whether there is a significant difference between the two groups – before refactoring and after refactoring –, we applied the *Mann-Whitney Wilcoxon* test. In all target systems, the test revealed p-values higher than 0.05 for both code smells and principle violations, indicating that we cannot reject the null hypothesis. Based on such result, we concluded that refactorings applied in the systems of this case study did not reduce the density of any of the investigated symptoms.

Regarding diversity, we carried out a similar analysis. Table 5.5 shows the mean number of different types of code smells and principle violations. In this case, the difference was often marginal for both symptom categories. Moreover, we did not observe similar trends for most systems. The mean diversity for both symptom categories was reduced in some systems but increased in other systems. The statistical test revealed p-values higher than 0.05 for both code smells and principle violations in most systems. The only systems in which we observed a statistically significant difference for principle violations were Achilles, MPAndroidChart, and Elasticsearch. For code smells, only in Elasticsearch the diversity before and after refactoring was statistically different. Thus, it is not possible to state that diversity of any symptom category always reduces or increases after refactorings.

As we could not observe a significant and consistent reduction in the density or in the diversity of symptoms, the answer of our second research question RQ2 is that **refactorings cause little to no impact on symptoms of design problems in the medium term**. This trend was maintained even after applying the second filter (Section 5.3.3) to the refactorings of open source systems.

To better understand why most refactorings did not remove symptoms, we decided to take a close look at refactored classes with the help of our industry partners. We selected and analyzed two sub-sets of refactored classes from OpenPOS and UniNFe: (1) classes with increased density, and (2) classes with decreased density. The former is composed by refactored classes that, after refactorings, presented higher density of symptoms. The latter is composed by

Table 5.6: Classes with increased density and diversity of symptoms

System	Class
OpenPOS	OpenPOS.Data.Abstract.Faturamento.Lancamento.Movimento.NF.NFBase
	OpenPOS.Data.Regra.CFOP.CFOPRegraFiltro
	OpenPOS.Data.Abstract.Cadastro.Item.ItemBase
	OpenPOS.Desktop.Forms.FrenteCaixa.Lancamento.frmVendaCF
UniNFe	UniNFe.Service.TFunctions
	UniNFe.Service.Processar
	UniNFe.Service.TaskAbst
	UniNFe.ConvertTxt.UniNFeW
	UniNFe.Service.TaskConsultarLoteeSocial
	NFSe.Components.SchemaXMLNFSe_TIPLAN

refactored classes that, after refactorings, presented lower density of symptoms. With the first set of refactored classes, we expected to identify and analyze the classes that, even after refactorings, have continued to worsen the design quality. On the other hand, with the second set, we intended to find cases of success in which the refactorings fixed design problems.

Symptoms tend to increase in core classes. Table 5.6 shows the classes of OpenPOS and UniNFe that presented higher density for both symptom categories. Analyzing the classes in which there was an increase in the density of all symptoms, we asked the developers to describe what they remember about the implementation and maintenance of each class. Based on their observations, we noted that many of the refactored classes are also frequently changed in other tasks. In addition, many of the refactored classes that presented higher density after refactorings are considered core classes of the system. That is, they are linked to fundamental functionalities of the system. The *frmVendaCF* class, for example, is a core class that was changed in 306 different commits, while most classes in the OpenPOS system were not changed more than 20 times. Such changes may be often conducted without proper concern for design quality.

As a result, any improvement promoted by refactorings ends up getting lost with the design problems. Thus, even being refactored in three different tasks – for improving modifiability and reusability –, *frmVendaCF* continued to present high density and diversity of symptoms. In fact, analyzing these results from the perspective of the refactoring literature, other studies (Bavota *et al.* 2015), (Cedrim *et al.* 2017) have pointed out that refactoring in general (not just root canal) does not usually remove symptoms such as code smells. We conjecture that this is due to the fact that carrying out design transformations is usually costly. Therefore, developers end up performing refactorings only to avoid increasing design problems in classes that (1) have been poorly designed, or (2) undergo constant modifications related to changing requirements.

The effects of refactorings only persist in classes that are not often modified. In OpenPOS and UniNFe, only the *frmCliente* class from OpenPOS presented a decreased number of symptoms. We observed that the design quality of this class was indeed improved. However, the developers of OpenPOS revealed to us that *frmCliente* is not often modified – it was changed less than 20 times along source code history. Therefore, a natural conclusion is that the low volume of changes allowed this class to maintain a good design. Nevertheless, design problems is critical when impacting the core classes of the system, such as the ones presented in Table 5.6. Thus, developers still need help to effectively identify and refactor the most relevant design problems.

5.4.3
Combinations as Indicators of Design Problems?

In the previous research questions we observed that developers tend to apply refactorings in classes with high density and diversity of symptoms, and that most refactorings present little to no persistent positive effects on the density and diversity of symptoms. Thus, aiming at improving the effectiveness of existing detection techniques for design problems, we investigated which combinations of symptoms are more likely to indicate design problems.

As explained in Section 5.3, we identified the number of classes affected by 170 pairwise combinations of code smell types with principle violation types both for refactored classes and for other classes. Table 5.7 shows the top-10 pairwise combinations of code smells and principle violations in refactored classes. Each line corresponds to a pairwise combination of code smell and principle violation. Column 3 shows the number of refactored classes, considering all target systems, affected by each pairwise combination. The complete rankings of combinations for both groups (refactored and other classes) are available in our replication package.

To observe whether the recurrent combinations can be used as indicators of design problems, we applied the *Spearman's rank correlation rho* test to compare the ranking of combinations in refactored classes and in other classes. With a confidence level of 95% and p-value smaller than 0.00001, we obtained 0.90 as the correlation coefficient. This result indicates a strong correlation between the two rankings, which means that **we cannot use the combinations of symptoms to differentiate degraded classes from other classes**. On the other hand, based on results from the literature (e.g., (Abbes *et al.* 2011)), we believe that combinations of symptoms may still be useful in other contexts, as we will describe in Section 5.5.

Table 5.7: Top-10 combinations of symptoms in refactored classes

Position	Combination	# of Affected Classes
1	Long Statement-Insufficient Modularization	216
2	Complex Method-Insufficient Modularization	212
3	Magic Number-Insufficient Modularization	178
4	Long Statement-Deficient Encapsulation	145
5	Complex Conditional-Insufficient Modularization	135
6	Complex Method-Deficient Encapsulation	126
7	Long Statement-Unutilized Abstraction	111
8	Magic Number-Deficient Encapsulation	109
9	Complex Method-Unutilized Abstraction	102
10	Long Parameter List-Insufficient Modularization	99

5.5 Requirements for Recommending Root Canal Refactorings

Given the results presented in this chapter, we envision a technique for automatically recommending refactorings for degraded classes. This technique should take into consideration our insights on the density, diversity, and combination of symptoms. Therefore, based on such insights, we elicited four main requirements for the recommendation of refactorings. As illustrated by Figure 5.3, requirements involve the activities of symptom collection, filtering of refactoring candidates, prioritization of the most relevant candidates, and summarizing of information about design problems. Below we present the detailed description of each requirement.

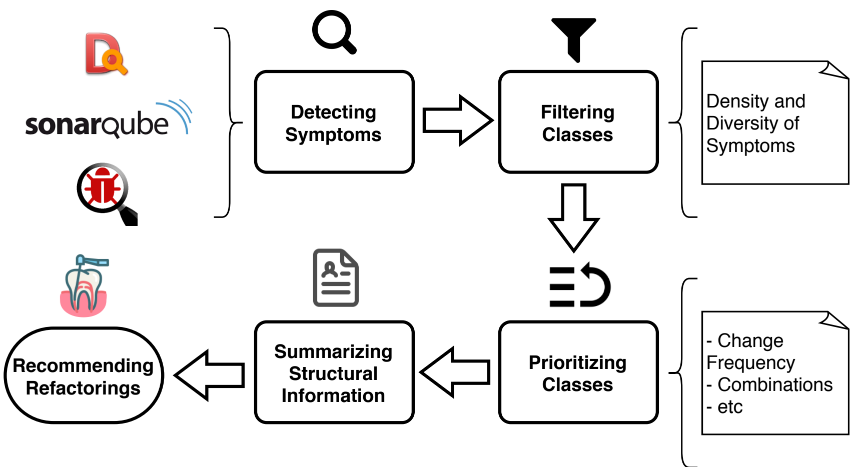


Figure 5.3: Steps taken by a recommender technique based on our proposed requirements

Collecting multiple symptom categories We observed in this study that, according to the theory proposed by Sousa et al. (Sousa *et al.* 2018), we should rely on two or more categories of symptoms to identify degraded classes. Each symptom category will reveal design problem aspects in dependability-related attributes that other symptom categories may not be able to capture. For example, while most code smells investigated in this study are related to modifiability and analyzability, the principle violations are mostly linked to modularity and reusability. Our results showed that high density and diversity in both symptom categories is usually associated with deep design problems that is difficult to fix even after successive refactorings. Therefore, a recommender technique should combine information about both symptom categories to provide precise recommendations. In this study, we used only symptoms provided by the Designite tool. However, recommendation techniques can implement their own detection strategies or can be based on symptoms provided by other tools, such as SonarQube (Campbell and Papapetrou 2013) and SpotBugs (SpotBugs 2019).

Filtering classes Due to several constraints, developers cannot waste time with the analysis of classes that do not need to be refactored. Thus, developers would benefit from a technique that automatically filters and selects only the classes that have the greatest chance of presenting design problems. Our findings revealed that combining density and diversity of symptoms such as code smells and principle violations can be an effective strategy for selecting degraded classes. The filter may consider only one category of symptom or it may combine multiple symptom categories. Some state-of-the-practice tools (e.g., SonarQube) already considers the density of symptoms to filter and to prioritize elements for refactoring. However, our study shows that diversity of symptoms should also be considered as degraded classes tend to present higher diversity of symptoms. The combination of both information could make filtering even more stringent, helping to save time for developers.

Prioritizing classes Even after filtering classes, there will be too many candidates for refactoring in medium- and large-sized systems. Thus, it is fundamental to prioritize the refactoring candidates according to their relevance. Our study provided evidence that the core classes of the system should receive special attention because they are often involved with the main changes made throughout the system evolution. One of the characteristics that differentiate these core classes is the change frequency. Thus, this information can be used as a prioritization criterion for ranking the refactoring candidates. The symptom combinations may also be explored as a criterion to prioritize refactoring candidates. For instance, degraded classes can be prioritized based on the combinations of symptoms that developers, of the team, refactored more often in previous projects and previous tasks. The effectiveness of this idea is supported by findings from the literature (e.g., (Abbes *et al.* 2011)).

Depending on the context, other factors can also be taken into account to prioritize classes for refactoring. For instance, developers may want to prioritize refactoring of degraded classes that will be modified in future tasks. One of the challenges for the implementation of this criterion is the identification of which classes will be modified by future tasks. In our future work, we plan to address this challenge using the strategy adopted by Kim et al. (Kim *et al.* 2013) to locate bugs based on bug reports. We believe that our proposed criteria could be helpful for generating more accurate rankings, since existing prioritization criteria (e.g., (Vidal *et al.* 2016, Vidal *et al.* 2019)) are unable to present consistent results for every system.

Summarizing design information One problem that often hinders the adoption of automated tools is the difficulty in understanding, exploring, and combining different symptoms. Knowing beforehand which combinations of symptoms are most recurrent, the tool can be prepared to explore the characteristics of the most recurrent combinations, providing detailed information about the types of design problems indicated by each combination. In addition, this information may be used by the tool to recommend specific refactoring operations, according to the refactoring operations that are usually associated with each combination. To provide a more readable and easy to understand summary, the recommender technique may apply an approach similar to the one designed by Moreno et al. (Moreno *et al.* 2013). This summarized and readable design information would help developers to reason about each degraded class and to perform more effective refactorings.

Although we have focused our research mostly on maintainability, we believe that these requirements can be applied in the context of other attributes, such as reliability, security, and performance. We leave for future work the evaluation of this technique in the context of maintainability and other dependability-related attributes.

5.6 Threats to Validity

The first threat to validity is regarding our dataset. Analyzing data from eight systems may not be enough for finding generalizable results. We mitigated this threat by selecting systems with different characteristics, developed in different platforms and with different practices. Another threat is related to the method used to find the refactorings. For the systems of our industry partners, we may have missed refactorings that were not remembered by the developers or did not contain the searched keywords. To mitigate this threat, we asked for at least two developers of each system to provide us a list of refactorings. All participating developers have knowledge about refactoring and have worked in the systems since their inception. Still related to the refactorings, it is possible that a task description demonstrates the intention to remove design problems but this does not occur in practice. We mitigate this threat by checking, with the help of developers, whether there was any discrepancy between the task description and the actual changes made.

In the context of open source systems, we may have missed several refactorings after applying multiple filters. This may have influenced the number of outliers observed in the set of non-refactored classes. However, this has helped us to drastically reduce the possibility of false positives. In addition, we discarded systems that had a very low number of refactored classes after the filters were applied.

There is another threat related to the tools used for detecting symptoms. Aspects such as precision and recall may have influenced the results of this study. We mitigate this threat by selecting Designite, the only tool we know that is capable of detecting the investigated symptoms both in C# and in Java systems. Moreover, Designite has been used successfully in other recent studies (Sharma *et al.* 2016, Alenezi and Zarour 2018). Finally, there is a threat regarding reproducibility as we are not allowed to publish detailed information about the refactorings and about the issue tracking system of our industry partners. Thus, to mitigate this threat, we created a replication package containing, among other information, the description of each refactoring identified in their systems.

5.7 Conclusion

In this chapter, we investigated whether symptoms of design problems appear with higher density and diversity in classes changed by refactorings. After that, we investigated if refactorings have a positive impact on the density and diversity of symptoms. We also investigated the combinations of symptoms that often occur in classes that were changed by refactorings. To achieve our goal, we conducted a case study involving two C# systems from our industry partners and six open source Java systems.

Our results indicate that refactorings caused almost no positive impact on the density and diversity of any category of symptom. Nevertheless, we also observed that refactored classes have higher density and diversity of code smells when compared to other classes in the target systems. This result indicates that, despite not being removed by refactorings, some categories of symptom may be indeed strong indicators of design problems. Based on our insights, we proposed a set of four requirements for automatically recommending refactorings. These requirements are related to the tasks of: (1) symptom collection, (2) filtering, (3) prioritization, and (4) summarizing of information about design problems. As future works, we aim at proposing and evaluating a semi-automated technique based on the requirements for recommending refactorings.

6
Recommending Composite Refactorings for Design Problem Removal: Heuristics and Evaluation

Design problems (DPs) occur when quality attributes of a system are negatively impacted. When due attention is not paid to DPs, the source code may also become difficult to change. The previous chapters focused on investigating issues related to the identification of DPs using code smells as indicators. Code smells are recurring structures in the source code that may represent DPs (Section 2.6.1). However, previous chapters did not approach how to support developers in refactoring out DPs that are relevant to their projects and contexts.

There are many catalogs and techniques for supporting the removal of code smells through refactoring recommendations, which usually consist of single refactorings such as a Move Method or an Extract Method. However, single refactorings are often not enough for completely removing certain smell occurrences. Moreover, recent studies show that developers most often apply composite refactorings – *i.e.*, sequences of two or more refactorings – for removing code smells. Despite showing the importance of performing composite refactorings, most studies do not provide information on which composite refactoring patterns are recurrent in practice. In addition, in Chapter 5, we found evidence that refactorings performed in practice are often ineffective for removing DPs. Such a finding is reinforced by recent refactoring studies (e.g., (Bibiano *et al.* 2019), (Cedrim *et al.* 2017)).

In this context, in a previous collaborative study (Sousa *et al.* 2020a), we have identified 35 smell removal patterns that are frequent across multiple open source systems. However, such a study has not explored how the removal patterns could help developers to apply effective composite refactorings. Thus, in this work, we propose a suite of new recommendation heuristics to help developers in applying effective composite refactorings. These heuristics are intended to remove three DP types, namely Complex Component, Scattered Feature, and Feature Overload. The manifestation of such DPs are represented here by the Complex Class, Feature Envy, and God Class smells, respectively.

The justification of why we focused on these DPs and smells is given along this chapter.

After designing the heuristics, we evaluated their effectiveness through a quasi-experiment. This evaluation was conducted with 12 software developers and 9 smelly Java classes. Results indicate that developers considered our heuristics effective or partially effective in more than 93% of the cases. In addition, the evaluation helped us to identify multiple factors that contribute to the acceptance or rejection of the refactoring recommendations. Based on these factors, we defined new guidelines for the effective recommendation of smell-removal composite refactorings. Such guidelines helped us to complement the requirements identified in Chapters 3, 4, and 5.

The study presented in this chapter was published as conference paper at the *Brazilian Symposium on Software Engineering* (SBES) (Oizumi *et al.* 2020). The quality of this study was recognized by the Brazilian software engineering community through the Distinguished Paper Award.

6.1 Introduction

The design of a system is often impacted by design problems due to the code changes that are carried out throughout its evolution (MacCormack, Rusnak and Baldwin 2006). As a result, maintaining the impacted code elements can become increasingly difficult and error-prone. To avoid this and other consequences, developers often need to take actions for identifying and removing design problems.

To identify design problems, developers can rely on automatically detected symptoms such as code smells (Lanza and Marinescu 2006, Fowler 1999). An example of code smell is the God Class, which is a recurrent indicator for the Feature Overload problem. Such a problem occurs when a class is large and has too many responsibilities (Lanza and Marinescu 2006).

Refactoring is a popular technique that is often applied for removing DPs (Fowler 1999). This technique consists of applying micro changes in the source code design without impacting the system's behavior (Fowler 1999). If refactorings are carried out properly, the design quality of the system can be improved, or at least maintained. However, in Chapter 5 we found evidence that refactorings performed in practice are not always effective in removing DPs.

Existing catalogs and recommendation techniques (*e.g.*, (Fowler 1999, Tsantalis, Chaikalis and Chatzigeorgiou 2018)) usually recommend only single refactorings for removing DPs. However, besides not being enough for DP removal, single refactorings are often linked to the introduction of new DPs (Cedrim *et al.* 2017). Moreover, recent studies show that developers often apply more than one refactoring, even if they aim for small design improvements (Bibiano *et al.* 2019, Oizumi *et al.* 2019, Brito, Hora and Valente 2020, Tufano *et al.* 2015, Palomba *et al.* 2017). In this chapter, we call such refactoring sequences as *composite refactorings*. Existing techniques for the recommendation of composite refactorings usually provide recommendations that involve re-designing the whole system (Alizadeh and Kessentini 2018). However, refactoring a large number of classes is often not feasible in a real project. Therefore, the literature still lacks techniques that assist developers in the application of composite refactorings for preventing major DPs. Despite making the importance of composite refactorings evident, existing studies provide little to no information on which composite refactoring patterns are applied in practice. Moreover, the literature still lacks guidelines for the effective recommendation of composite refactorings.

Given the aforementioned limitations, in this chapter we propose and evaluate three heuristics for composite refactoring recommendations. We are focused in recommendations based on three code smell types, namely Complex Class, Feature Envy, and God Class. Those smell types represent common forms of DPs that developers often consider harmful (Palomba *et al.* 2014), and that are early introduced with the creation of new code elements (Tufano *et al.* 2015).

To propose the heuristics, we relied on composite refactoring patterns that are often associated with the removal of code smells. In our previous work (Sousa *et al.* 2020a), we mined refactorings and code smells from 48 Java open source projects to identify such patterns. After analyzing 104,505 refactorings, we found the occurrence of 2,946 smell-removal composite refactorings. Such investigation resulted in the identification of 35 patterns, being 9 for Feature Envy, 11 for God Class, and 15 for Complex Class. In this work, for each smell type, we selected and implemented one pattern in the form of a composite refactoring recommendation heuristic.

The evaluation of our heuristics consisted of a quasi-experiment involving 12 software developers. We applied our heuristics to 9 Java smelly classes and asked participants to evaluate the resulting recommendations. The participants had to evaluate the recommendations according to their effectiveness in improving the design of the code. In addition, for each evaluated recommendation, the participants provided a detailed feedback on the impact they perceived. For the analysis of the provided feedback, we applied a qualitative data analysis method. This analysis allowed us to assess the proposed heuristics and to identify factors that contribute to the acceptance (or rejection) of refactoring recommendations. Such factors helped us to propose new guidelines for the effective recommendation of composite refactorings.

Our contributions can be summarized as follows: (1) we propose and evaluate three heuristics that can be used to improve state-of-the-art refactoring recommendation tools; (2) participants of our study considered that the impact of the evaluated recommendations was positive or partially positive in more than 93% of the cases; (3) finally, we proposed guidelines that may improve refactoring recommendation techniques.

6.2 Background and Related Work

6.2.1 Code Smells

Code smell is a surface indicator of deeper DPs in the software system (Fowler 1999, Sousa *et al.* 2020b, Palomba *et al.* 2014). In fact, the presence of code smells can even indicate the need for a refactoring (Fowler 1999). An example of code smell is the *God Class*, which indicates the Feature Overload DP. A God Class usually impacts quality attributes such as modifiability and extensibility.

To illustrate how a code smell manifests in the source code, let us consider the `LibraryMainControl` class that contains multiple methods and attributes, as shown in Listing 6.1.

```
public class LibraryMainControl {
  ...
  private Float fineAmount;
  private Person user;
  private Catalog catalog;
  private Item currentItem;

  public void doInventory() {...}

  public void checkOutItem(Item item) {...}
  public void checkInItem(Item item) {...}

  public void addItem(Item item) {...}

  public void deleteItem(Item item) {...}

  public void printCatalog(Catalog catalog) {...}

  public void sortCatalog(Catalog catalog) {...}

  public void searchCatalog(String term) {...}
  ...
}
```

Listing 6.1: Partial view of the *LibraryMainControl* class

The `LibraryMainControl` class was flagged as *God Class* because it implements multiple features, which should be modularized in other classes. For example, the methods `printCatalog`, `sortCatalog`, and `searchCatalog` should be moved to the `Catalog` class as they are concerned with catalog-related features.

In this study, we focus on three types of code smells, namely *Complex Class* (Palomba *et al.* 2014), *Feature Envy* (Lanza and Marinescu 2006), and *God Class* (Lanza and Marinescu 2006). Such smells are symptoms of the Complex Component, Scattered Featured, and Feature Overload DPs, respectively.

God Class and *Complex Class* are frequently considered relevant by developers and there is evidence that *Feature Envy* and *God Class* reflect important maintainability aspects (Yamashita and Moonen 2013). These three smells are also of major severity if compared with many others, and their resolution would eliminate/reduce other inner smelly structures (e.g., *Long Method*) (Abbes *et al.* 2011, Cedrim *et al.* 2017). Finally, their removal usually requires at least two refactorings (Sousa *et al.* 2020a, Bibiano *et al.* 2020).

6.2.2
Composite Refactoring

Refactoring consists in transforming the code design without changing the functional behavior of the system (Fowler 1999). Thus, we consider that refactoring is any design change that is aimed at improving quality attributes of the system's design. There are multiple refactoring types cataloged in the literature (e.g., (Fowler 1999) and (Tsantalis *et al.* 2018)). Each refactoring type is applied to perform a specific design transformation. For instance, Extract Class aims at creating a new class with methods and attributes of an existing class.

In Listing 6.1, existing catalogs and techniques (Fowler 1999, Tsantalis, Chaikalis and Chatzigeorgiou 2018) would probably recommend the application of Extract Class to solve the God Class smell. Although it is a correct strategy for many cases, it might not be the most appropriate for the *LibraryMainControl* class. The reason is that the additional responsibilities of that class would be better modularized into the *Catalog* and *Item* classes.

A **composite refactoring** (or batch refactoring) happens when two or more related refactorings are applied to one or more code elements (Bibiano *et al.* 2019, Brito, Hora and Valente 2020, Palomba *et al.* 2017, Tufano *et al.* 2015, Sousa *et al.* 2020a). The composites can be divided into two broader categories, namely temporally-related composite (*i.e.*, refactorings applied in the same commit) and spatial composite (*i.e.*, refactorings applied to structurally related code elements, on the same commit or not) (Sousa *et al.* 2020a). Even though some studies indicate that the composite refactorings are applied by a single developer (Bibiano *et al.* 2019, Murphy-Hill, Parnin and Black 2012), there are cases where the developers can work in groups to perform a composite (Kim, Zimmermann and Nagappan 2014) (*i.e.* when they are applying large design changes in the software system). For the *LibraryMainControl* class, a suitable composite refactoring would be to perform multiple Move Method refactorings, transferring methods from *LibraryMainControl* to the *Catalog* and *Item* classes.

6.3 Smell Removal Patterns

In our previous work (Sousa *et al.* 2020a), we analyzed composite refactorings from a dataset of 48 open source projects. Our results revealed composite refactoring patterns that are associated with the introduction or removal of code smells. Such patterns served as basis for creating composite refactoring recommendation heuristics. Next, we summarize the research procedures and the resulting patterns. More details are available in our replication package [1].

Procedures for Finding Patterns. Firstly, to identify code smell removal patterns, we detected the code smells and the refactorings. We implemented rule-based detection strategies for code smells (Lanza and Marinescu 2006)

[1]`https://refactoringheuristics.github.io/`

Table 6.1: Refactoring patterns that often remove smells

Smell Type	**Pattern**
Feature Envy	**Extract Method, Move Method**
	Inline Method{n}, Extract Method{n}
	Extract Method{n}, Move Attribute{n}
God Class	**Move Method{n}**
	Pull Up Method{n}, Move Method, Pull Up Method
	Pull Up Method{n}
Complex Class	**Extract Method{n}**
	Pull Up Method{n}, Move Method, Pull Up Method{n}
	Move Attribute{n}, Extract Method{n}

and used the the Refactoring Miner tool (Tsantalis *et al.* 2018) for detecting refactorings. After that, we applied a synthesis strategy that is able to identify both temporally-related composites and spatial composites (see Section 6.2.2). After finding composite refactorings, we identified recurrent composite refactoring patterns related to the removal of Feature Envy, God Class, and Complex Class. This identification was based on the frequency with which each pattern resulted in the removal of each smell type.

Refactoring Patterns. Following the aforementioned method, we found 2,946 smell-removal composite refactorings. Then, we identified 35 smell removal patterns, being 9 for Feature Envy, 11 for God Class, and 15 for Complex Class. Table 6.1 shows a sub-set of the most frequent patterns for each smell type. In the second column of Table 6.1 is the list of refactoring types that compose each pattern. The *{n}* symbol after some refactoring types means that the refactoring type was performed an *n* number of times in the analyzed cases. Such number was varied in each occurrence.

Selection of patterns for heuristics implementation. In Table 6.1, we highlighted in bold the smell removal patterns that were selected for the implementation of our heuristics. We applied the following criteria for selecting them. First, we opted for patterns that would be viable to implement using state-of-the-art algorithms for automated refactoring (e.g., (Charalampidou *et al.* 2017, Tsantalis, Chaikalis and Chatzigeorgiou 2018)). Second, we chose the patterns involving the least number of different refactoring types. Thus, we believe that the resulting recommendations would be easier to analyze and understand. Finally, we selected patterns that would make sense for most possible scenarios. This last criterion is important because there are patterns that can only be applied for classes with inheritance (e.g., *Pull Up Method{n}*). Given such restrictions, we selected *Extract Method, Move Method* for Feature Envy, *Move Method{n}* for God Class, and *Extract Method{n}* for Complex Class. In the next section, we present the heuristics that were implemented based on such patterns.

6.4 Smell Removal Heuristics

In this section we present three heuristics for removing code smells. Each one is focused in removing a particular type of code smell. They were derived from the refactoring patterns presented in Section 6.3. As presented in that section, there are many patterns that may be applied for removing code smells. For instance, it is possible to remove a *Complex Class* by applying several *Push Down Methods* or by applying a sequence of *Extract Methods*. Since our objective in this study is to check the viability of deriving useful heuristics, we implemented and evaluated only one pattern for each code smell type. Next, we present details about each heuristic.

6.4.1 Feature Envy Removal

One of the most common patterns found for removing *Feature Envy* is composed of *Extract Method* and *Move Method*. Although this sequence may have been applied independently – i.e., the *Move Method* may have been applied to a different method than the extracted one – many times when developers were successful in removing *Feature Envies*, they first extracted the foreign part of the method into a new one, and then moved the newly created method to a different class. Thus, we define a *Feature Envy* removal heuristic that always applies the *Move Method* to the extracted method. Formally, the *Feature Envy* removal heuristic is composed of four parts: (i) identification of method lines that are more interested in different class; (ii) extraction of these lines into a new method; (iii) identification of the class that suits the newly-created method; and (iv) application of a *Move Method* refactoring to move the new method to the identified class.

Each step of this heuristic poses a different challenge. The first one is that we need to identify lines of the *Feature Envy* method that are more interested in a different class other than the one they are contained in. It is not trivial to recommend lines of code to be extracted from a method because the section of code to be removed must execute a particular functionality in a way that the removal of the whole section makes sense. Several studies propose different techniques to accomplish the objective of discovering extract method opportunities (Tsantalis, Chaikalis and Chatzigeorgiou 2018, Charalampidou *et al.* 2017). The technique proposed by Charalampidou *et al.* (Charalampidou *et al.* 2017) is based on the functional relevance of the combined lines. They introduce an approach that aims at identifying source code chunks that collaborate to provide a specific functionality and propose their extraction as separate methods. Since their approach fits well with our first part for recommending *Feature Envy* removal, we adopt it.

Therefore, to accomplish the first part of this heuristic, we implemented the approach proposed by Charalampidou *et al.* (Charalampidou *et al.* 2017). As described in their paper, they propose an approach called SRP-based Extract Method Identification (SEMI). In particular, their approach recognizes fragments of code that collaborate for providing functionality by calculating the cohesion between pairs of statements. The extraction of such code fragments can reduce the size of the initial method, and subsequently increase the cohesion of the resulting methods. In our scope, we implemented their technique and treated this component as a black box, where the input is a method and the output is a set of line intervals that can be extracted.

By having these possible intervals, we have several possibilities to recommend extraction. However, only having these intervals is not enough to remove the *Feature Envy*, since we do not want to recommend an extraction that would still maintain the smell that we already had. Therefore, we run a verification step for each interval. We simulate the removal of such lines by disregarding their influence on the *Feature Envy* detection. Hence, we test, for each interval, if its removal would lead to a *Feature Envy*. If the removal of a single interval is not enough to remove the *Feature Envy*, then we look for a combination of two intervals. We keep increasing the number of intervals until we have a *Feature Envy* removal possibility. After completing this step, we can move on for the second part of the heuristic: *Extract Method* refactoring.

After the aforementioned step, we can test if there is still a *Feature Envy*. Unfortunately, the newly-created method could still have the *Feature Envy*. However, we have to leverage in the fact that we know its lines are cohesive and can be moved together to a different class. In this way, we check which class this new method relates to the most, either by method calls or attribute use. After discovering this class, we can recommend a *Move Method* refactoring of the newly-created method to this discovered class.

Therefore, our first heuristic is completed by executing the four described steps. We first recommend an *Extract Method* by combining our code smell detection strategy with the SEMI approach. After extracting the method, we can recommend a *Move Method* by examining the method calls and attributes use of the newly-created method. In this way, we reproduce programmatically the composite-smell pattern *Extract Method, Move Method* presented in Section 6.3. Notice that we do not only reproduce the removal pattern; we instead execute a verification to make sure that the composite would be able of removing the smells. The combination of the verification with the knowledge about the removal patterns is what increases the chance of our heuristic to get rid of a *Feature Envy*.

6.4.2 God Class Removal

As presented in Section 6.3, *Move Methods* play a central role on *God Class* removals. *God Class* is a class that assumes several responsibilities in a system. If we distribute these responsibilities (*i.e.*, methods) over several classes in the system, the developer can remove the smell (Section 6.3). Indeed, to perform this distribution, we found that developers can apply different types of refactorings; for instance, he can apply a composite composed of *Move Method{n}*. Hence, the heuristic we implemented to remove God Class is based on method-moving refactorings.

According to the rules of *God Class* detection (Bavota *et al.* 2015), a class has this smell if its cohesion is lower than the average of the system, and it contains more than 500 lines of code. This threshold can be tailored to particular projects or modules by using machine learning techniques (Hozano *et al.* 2017). Thus, for each method in the class, we identify a suitable class to which we move it. We used the same strategy presented in Section 6.4.1 to identify the destination class of the method. We keep recommending *Move Methods* until the *God Class* is removed. However, such operations can create new *God Classes* in the system (Sousa *et al.* 2020a). Therefore, before recommending a Move *Method*, we check if the destination class would become a *God Class.* If so, we change the recommendation to the second most suitable class.

It is worth mentioning that we find the suitable class by counting the number of method calls and accesses to attributes. For instance, let us assume that a particular method m calls 3 methods and accesses 2 attributes from class A. In this case, the "bonding factor" of m to A is 5. Let us assume the same method m calls 4 methods from the class B, leading to a bonding factor of 4. Now, assume our heuristic recommended to move m to A, but A would be transformed into a *God Class* if this occurs. In this case, the heuristic would recommend the method-moving change to class B, since in this case, B would still be smell-free.

In summary, the second heuristic is a sequence of *Move Methods*. However, it also uses the smell-detection strategy to understand when the target class would not be a *God Class* anymore. Additionally, the smell detection strategy is used to prevent the creation of new code smells after the recommended refactoring.

6.4.3
Complex Class Removal

In our studies, we consider a class as *Complex Class* if it has at least one method having a high cyclomatic complexity (CC) (Lanza and Marinescu 2006). So, the strategy to remove such smell is related to the reduction of the complexity of methods with high CC. As presented in Section 6.3, developers often apply *Extract Methods* to remove such complex structures. Hence, the heuristic to remove *Complex Class* is composed of four parts: identify all methods with high CC, identify *Extract Method* opportunities to reduce the complexity, evaluate the identified opportunities, and recommend *Extract Methods*.

The first part is composed of our code smell detection strategy, which finds all *Complex Methods* in a particular *Complex Class*. After finding them, we use the SEMI approach presented in Section 6.4.1 to generate possible line intervals to be extracted. After identifying such intervals, we need to evaluate the identified opportunities. For each interval found, we simulate its removal and compute what would be the new complexity of the method. When we find a interval (or a set of intervals) that reduces the complexity, we start recommending the *Extract Methods*.

Therefore, after running the steps of this heuristic, our tool can identify pieces of code that can be extracted to reduce the complexity of the methods found. After the recommendation of a composite of *Extract Methods*, we can distribute the complexity of the class into several smaller methods, getting rid of the original *Complex Class*. It is worth mentioning the recommended extractions can pose a risk and create new code smells, such as a new *Feature Envy*. If this occurs, we can trigger the *Feature Envy* removal heuristic to improve the composite by removing the introduced smell.

6.5
Empirical Evaluation: Study Design

This section presents the design of a quasi-experiment (Shadish, Cook and Campbell 2001). This experiment was designed to evaluate the three proposed heuristics.

6.5.1

Goal and Research Question

We hypothesize that our heuristics (Section 6.4) can be effective in real scenarios. Thus, our goal is to *evaluate the smell-removing heuristics regarding their effectiveness in improving the source code design quality*. To achieve this objective, we observed how the heuristics perform in real scenarios to evaluate their effectiveness. Thus, our quasi-experiment involved software developers and industry systems. To avoid bias, we applied the heuristics to projects that were not used to find the composite refactoring patterns (Section 6.3).

Our first research question – *RQ1: Are the smell-removing heuristics effective in improving the code design quality*? – is intended to assess the effectiveness of our heuristics in helping developers to combat design degradation. To address RQ1, we first applied the heuristics steps on different smelly code elements. Each heuristic comprises some steps (Section 6.4), and the application of each step in a code element delivers a new code state. In this way, we documented each code state obtained as a result of the application of each heuristic step. After this, we compiled all the results and asked for the evaluation of software developers. They had to evaluate the code states and inform us about their opinion concerning the impact of the code changes on the design quality. After this, we conducted quantitative and qualitative analysis.

The second research question – *RQ2: Why do developers accept or reject a composite refactoring recommendation?* – aims to characterize the factors that lead to the acceptance or rejection of refactoring recommendations. These factors are relevant to smell removal recommendations in general (not only ours), but they have never been investigated in previous studies. To answer this question, we relied on the qualitative analysis of responses provided by developers in the quasi-experiment. We applied systematic procedures to find and categorize the reasons that led participants to accept or reject the recommendations. Such procedures led us to build an initial set of guidelines related to the acceptance or rejection of refactoring recommendations. More details about the qualitative data analysis procedures are provided in Section 6.5.3.

6.5.2
Experimental Tasks

To evaluate the heuristics, our quasi-experiment was composed by three main activities, described as follows.

Activity 1: Sample Selection and Heuristics Execution. Since our objective is to evaluate the heuristics, we had to execute them in different classes affected by the studied code smells, then we selected 3 different classes for each smell. Then, we executed each of the proposed heuristics in the contexts of three classes containing the corresponding smells. We selected three classes for each smell, so that the experiment's participants could have the proper time to inspect each one of the 9 recommended composites. Also, we selected, for each smell, classes from three distinct projects. Besides that, we chose classes implemented with different purposes, from log-in services to classes that manage students data from an educational institution. Since our goal here was not to compare different heuristics, we decided to select classes with varying complexity and degradation characteristics. This helped us to evaluate the heuristics in heterogeneous scenarios. We then executed the heuristics for each sample. As presented in Section 6.4, each heuristic produces as output a list of recommended refactorings.

Activity 2: Recruitment and Characterization of Subjects. We invited 20 software developers to participate in this study, among which 12 met the minimum requirements and agreed to participate. We asked the developers to fill out a questionnaire to gather their information, including educational level, professional experience with software development in terms of years, experience with Java programming (in years), and whether they were familiar with code smells and refactoring or not. The data collected during this activity was used to understand if the participants met the minimum requirements needed to participate in the experiment. Since all code examples are in Java, the participants have to be able to read and understand the code. They also need to know how to refactor a piece of code. Otherwise, it would be very hard for them to understand the heuristics steps, invalidating their answers. Screen shots of the questionnaire are available in our replication package. Table 6.2 presents the data regarding the participants' years of experience with software development, years of experience with Java, and number of Java developed projects. Most participants have industry experience with software development, and with the Java language. Only one participant has no experience in industry. Nevertheless, such a participant does not pose a threat to the study due to his/her experience with code smells and refactoring research.

Table 6.2: Participants' characterization data

Answer	Median	Average	Std. Dev.	Max	Min
Years of programming experience	4	5.5	4.6	15	0
Years of experience with Java	1	2.8	3.9	14	0
Number of Java developed projects	1	5.4	13.4	50	0

Activity 3: Experiment Execution. As mentioned before, we executed the heuristics' steps on 9 different smelly classes. Each execution led us to a sequence of refactorings, implicating in several code changes. Each participant had to evaluate each sequence of refactorings generated for each one of the 9 classes. Hence, each participant had to visualize and evaluate 9 composite refactorings. After visualizing each composite, the participants had to answer the following question:

> What is your opinion about the impact of the sequence of refactorings on the design quality?

As we can see, this question does not involve the term *code smell*. Although the heuristics had been derived from relationships between composites and code smells, we are ultimately interested in improving the design quality. If developers feel that the code had its design quality improved by our heuristics, this is one more evidence that code smells are, in fact, good estimators to measure the design quality. In other words, we can keep developing heuristics focused in code smells because, in the end, the design quality can be improved by their removal.

Table 6.3: Possible answers during the quasi-experiment

Answer	Description
Positive	The design quality has improved
Intermediate	There are benefits, but I think there is room for improvement
Negative	The design quality has decreased

The participants had to choose between Positive, Intermediate, and Negative as an answer to the provided question. In each case, they had to provide a detailed feedback to justify the answer. Table 6.3 presents the three possible answers that each participant had to give for each one of the nine presented composites. In any case, they had always to provide a justification for the answer in an open text field, so we can use it to better understand their answers. We developed a web application in order to present composites and to collect the developers' answers.

6.5.3 Qualitative Data Analysis

One of our goals in this study was to discover and understand the reasons that lead a refactoring recommendation to be (partially) accepted or rejected by developers. Therefore, we asked the participants to explain the reasons for classifying a recommendation as positive, negative, or intermediate. Thus, the experiment provided us with a robust material for building an initial set of guidelines that would be helpful for providing effective recommendations. To identify the guidelines, we have adapted procedures that are commonly used in qualitative data analysis methods such as the Grounded Theory (Lazar, Feng and Hochheiser 2017). To avoid bias, three researchers participated in this analysis, conducting discussions whenever necessary. Below we present details about our procedures.

Coding the data. Open coding is a commonly applied procedure to extract relevant codes from textual content. Thus, we started our analysis with open coding, following the recommendations of Lazar *et al.* (Lazar, Feng and Hochheiser 2017). Before identifying and extracting the codes, we defined three types of textual sentences that we were interested in. The first type comprises sentences indicating positive motivations that contribute to the acceptance of a recommendation. The second type includes sentences that indicate negative motivations regarding a recommendation; thus, contributing to its rejection. Finally, for the last type, we included sentences related to suggestions of improvement for the recommendations. We are interested in this last type of sentence because we observed that participants provided several relevant suggestions during the experiment. This occurred even when they considered recommendations to be positive or intermediate. Based on the extracted and classified sentences, we identified codes that briefly described each relevant sentence.

Creation of categories and relationships. After identifying the codes, we searched for relevant categories of codes. The categories were created through the analysis and exhaustive comparison of the different codes. This allowed us to identify similar codes that were repeated in the responses of different participants and, therefore, resulted in the creation of categories. Besides that, we also identified the relationships that exist among the different categories, which is an important step in a qualitative data analysis.

Synthesis and description of the guidelines. In this last step, we analyzed the categories and relationships created to synthesize and describe our guidelines in the form of factors that are often related to the acceptance or rejection of refactoring recommendations. Given the number of participants, it was not possible to reach theoretical saturation. However, the data provide a significant contribution for refactoring recommendation techniques and tools.

6.6 Evaluation Results

6.6.1 Effectiveness of Recommendations

Each participant evaluated 9 refactoring recommendations produced by our smell removal heuristics (Section 6.4). In each evaluation, they had to inform their opinion about the impact of the composite on the design quality. Table 6.4 summarizes the data regarding the answers given by the participants. The first column shows the smell type that was targeted by each recommendation, while the second column shows the name of the smelly class. The number of classifications (positive, intermediate or negative) provided by the participants in each case are summarized in the 3rd, 4th and 5th columns of Table 6.4.

Most of the recommendations were considered positive or intermediate. The majority of participants considered most recommendations as being positive (74%) or intermediate (20%) (Table 6.4). Only few recommendations were considered negative by some developers (6%). For 4 out of 12 recommendations, no negative classification was provided. This acceptance rate is slightly higher than that of other similar techniques. Bavota *et* al. (Bavota *et al.* 2014), for example, reported a study in which more than 70% of their recommendations were considered meaningful.

Suggestions for improving the recommendations. As described in Section 6.5.3, after executing the quasi-experiment, we also conducted a qualitative data analysis. For all recommendations, at least one participant suggested some improvement. Despite having received only one negative classification, the heuristic for Feature Envy received the highest number of suggestions for improvement (12), followed by Complex Class (9) and God Class (5).

One of the most recurring suggestions was related to the provision of additional information regarding the proposed refactorings and regarding the design quality. Some of the participants who had little experience with refactoring had difficulty in understanding the design changes caused by the refactorings. In addition, the participants raised the need for some support to visualize the impact of refactorings on design quality. We further discuss this suggestion in Section 6.6.3.

Positive and negative sentences in the detailed feedback. To avoid bias, we analyzed the detailed feedback from the participants for finding sentences that indicate positive and negative aspects of the recommendations (Section 6.5.3). We identified, for each recommendation, the number of participants that: wrote only positive sentences, only negative sentences, both positive and negative sentences, and did not write any relevant sentences for this analysis. We also identified the total number of positive sentences and the number of negative sentences found in each detailed feedback. We included such results in our replication package [2].

In this analysis, we observed that the percentage of classifications with only positive sentences (59%) is significantly smaller than the percentage of positive classifications (74%). This happened because, even classifying the recommendations as positive, some participants wrote feedback indicating that there were negative aspects that could be improved. Similarly, certain recommendations classified as intermediate contained only negative sentences. Thus, looking from this alternative perspective, the percentage of intermediate and negative feedback is higher, i.e., 14% and 12% respectively. Still, if we look at the total number of sentences, we can see that the number of positive sentences (128) is significantly higher than the number of negative sentences (50). In addition, this result does not invalidate the classifications provided by the participants because, even when writing about aspects that could be improved, the participants considered the recommendations to be positive or intermediate. Next, we present details about the recommendations for each type of smell.

6.6.2
Impact of Recommendation Heuristics

We did not compare our heuristics with existing ones, since previous studies detect different types of code smell and use different detection strategies (Section 6.5). However, in this section, we present the results for each recommendation heuristic and discuss how they can be used for improving existing tools. For this purpose, we used the JDeodorant tool (Tsantalis, Chaikalis and Chatzigeorgiou 2018) as an example. JDeodorant is a well known tool that is

[2] https://refactoringheuristics.github.io/

Table 6.4: Summarized results of the quasi-experiment

Code Smell	Class	Pos.	Int.	Neg.
Complex Class	Clause	6	4	2
	GenericTranspalBean	11	0	1
	DiarioClasseService	10	2	0
Feature Envy	IngressoUniversidadeService	7	5	0
	Media	10	2	0
	UserFactory	10	1	1
God Class	MatriculaAcademicaService	7	4	1
	LibraryMainControl	10	2	0
	EmployeeUtils	9	1	2
All		80 (74%)	21 (20%)	7 (6%)

able to detect and recommend refactorings for six code smell types. Like us, JDeodorant is able to recommend refactorings for Feature Envy and God Class but not for Complex Class.

Support for Removing Complex Class. JDeodorant does not have any heuristic for detecting and removing Complex Classes. Thus, our heuristics could be incorporated by JDeodorant to make it possible to recommend composite refactorings that remove Complex Class. As presented in Table 6.4, our heuristic for Complex Class presented satisfactory results for most classes. Nevertheless, for the *Clause* class, our recommendation was considered positive only by 6 out of 12 participants. When looking at the detailed feedback, 3 participants wrote only positive sentences, 2 wrote only negative sentences, and 6 wrote both positive and negative sentences.

The *Clause* class was flagged as *Complex Class* because of the method *toCriterion*, which has a high cyclomatic complexity. This method is composed of a chain of ifs, leading to low code readability. As presented in Section 6.4.3, the heuristic recommends a list of *Extract Methods* to distribute the complexity. In this particular case, the heuristic recommended two *Extract Methods*. The *Complex Class* was removed because the class no longer has methods with high cyclomatic complexity.

However, the legibility of the created methods is still not good, as considered by some participants. They suggested a complete rewrite of this method by using different data structures and even a different object model design to implement the same functionality. Unfortunately, the heuristics are not able to recommend that, since the suggested changes are not part of our composite refactoring patterns. Thus, a total reshape is probably the best solution for the case. Nevertheless, in the perspective of 10 participants, the recommendation achieved some improvements, while still having room for making the class structurally better. Below we quote the feedback of one participant regarding this difficult case.

> "...I do agree the method is less complex, but the complexity was merely spread into the other two methods..." – *Feedback about the Clause class refactoring.*

If we look at the recommendations provided by the JDeodorant tool, a different type of recommendation could be provided for the *Clause* class. JDeodorant supports the recommendation of the *Replace Type code with State/Strategy* refactoring when there is a State Checking smell. In fact, one of the participants recommended this refactoring for the *Clause* class. Therefore, the solution proposed by the JDeodorant tool would be the most appropriate in this case. It is important to note that, as mentioned earlier, we would not be able to recommend such refactoring because we have not identified refactoring patterns for State Checking smells.

Despite this unfavorable case, our Complex Class heuristic is relevant for other cases. Most participants mentioned that it was able to significantly reduce the complexity of the *GenericTranspalBean* and *DiarioClasseService* classes. In these cases, complexity was caused by the presence of conditionals and nested loops. Thus, the recommendation of *Extract Methods* was adequate to reduce complexity and improve readability of the code.

Feature Envy Affecting the Implementation of Multiple Methods. The heuristic of JDeodorant for removing Feature Envies consists of recommending Move Methods for each of the affected classes. As observed in our study, Move Method is indeed applied by developers during the removal of a Feature Envy. However, in many cases, the sole execution of this refactoring is not enough. The reason is that the envious code may be within a method mixed with non envious code. Therefore, we conjecture that JDeodorant recommendations could be improved by our heuristic. Besides recommending the *Move Method* for fully envious methods, JDeodorant would also be able to address more complex scenarios based on the recommendation of composite refactorings. In such cases, our heuristic for *Feature Envy*, would help developers to remove *Feature Envies* that impact the implementation of multiple methods.

In fact, the heuristic for *Feature Envy* removal was the most successful one in our quasi-experiment. Only one participant gave a *negative* answer, while 8 answers were *intermediate*, and 27 were *positive*. These results gave us confidence about the removal patterns we found. We were able to derive a heuristic that was able to remove the smell very often, according to the participants. The developer we quote below is one of the *intermediate* answers.

"I believe the refactorings improved the code, but its readability is still not perfect. I believe the constructor is still large." – *Participant with 4 years of experience*

In this case, the participant agrees the heuristic was positive, but there is a suggestion to keep improving the source code. Most of the *intermediate* answers mention the need for more refactorings in order to reach an ideal state. However, most of those improvements were not closely related to the purpose of the heuristic being evaluated. The most experienced participant said:

"I agree with the proposed refactorings. They were enough to remove the code smell." – *Participant with 14 years of experience.*

Removing God Class without a New Class. For removing a God Class, JDeodorant recommends the creation of a new class through the Extract Class refactoring. This heuristic is effective when a new class is necessary. Nevertheless, there are scenarios in which the God Class contains methods and attributes that could be placed in an existing correlated class. Thus, our heuristic for God Class could present effective recommendations for developers in such scenarios. As described in Section 6.4, in such cases, our heuristic is able to remove God Classes without introducing new ones and without the creation of new classes.

In the three cases that we presented to the participants, we recommended several *Move Methods* to remove the *God Class*. All three classes are very large and contains thousands of lines and dozens of methods. Even in these highly complicated scenarios, the heuristics achieved 7 intermediate, 26 positive, and only 3 negative answers. Interestingly, not even a single one developer criticized a proposed *Move Method*. There were no complaints regarding the suggested refactorings. All complaints were related to the continuity of the improvements, i.e., the developers were expecting more *Move Methods* to solve the problem, as the one we quote below.

"This class still needs more refactorings, because I think it still contains several responsibilities." – *Participant with 1 year of experience.*

These results are exciting because it shows a difference between what we consider *God Class*, and what some developers think. According to our *God Class* detection rules, the target classes were not affected by the smell anymore after the refactorings, while the developers still think it is. In this case, we could change the rule to be more severe on the *God Class* detection. For instance, we can reduce the threshold of the number of lines a class must have to be classified as *God Class*. If we change the threshold, the heuristic would continue to suggest *Move Methods*, and the unsatisfied developers might be satisfied if we use the new rule. Most developers agree that the outcome is positive, as the one we quote below:

"I believe the class was very confusing before the refactorings. Now, after the refactorings, the class is way easier to understand and maintain." – *Participant with 2 years of experience.*

Upon data analysis, we can say that the evaluated heuristics are indeed effective in removing code smells. Even with some preliminary heuristics, we were able to achieve a high acceptance from the developers. In this way, it seems worthwhile to follow the path of improving and creating more heuristics. Nevertheless, we do not believe that existing heuristics should be replaced by our heuristics. We also do not expect our heuristics to show satisfactory results in any case. The reason is that, despite finding several composite refactoring patterns, we implemented heuristics corresponding only to one pattern per type of smell. Thus, as discussed above, our results show that state-of-the-art recommendation tools can, in fact, benefit from our results.

6.6.3
Guidelines: Improving Recommendations

Based on our qualitative analysis, we were able to identify several factors that contributed to the developers classifying our recommendations as positive, intermediate, or negative. These factors compose the answer for our RQ2. In fact, we believe these factors can be more useful than just explaining why the participants accepted or rejected a composite refactoring recommendation. In this sense, we relied on these factors to propose a set of guidelines that can help other researchers to propose recommendation heuristics. These guidelines may increase the chance of developers perceive techniques for composite refactoring recommendations as useful.

From the participants' feedback, we extract 128 positive sentences, 50 negative sentences, and 26 improvement suggestions (see our replication package [3]). Following systematic procedures (Section 6.5.3), we analyzed the sentences and identified multiple recurring codes that resulted in the creation of 16 factors that contribute to the acceptance and 11 factors that cause the rejection of refactoring recommendations. Next, we present guidelines related to the most relevant factors.

Refactoring need and impact must be clear. A prominent factor noted in this study was related to convincing developers that they should conduct the proposed refactorings. To achieve such objective, we noted that: (1) the need for refactoring must be clear, (2) the effect of the refactorings in the design quality must be evident, and (3) the developer must agree that the resulting design is significantly better than before the refactorings.

Regarding the need for composite refactoring, just informing the existence of a smell is usually not enough, except in cases where the degradation is severe. For the common cases, we noted that, the developer must be also informed both about the metrics that indicate the presence of the smell and about the quality attributes that are being impacted. In the experiment, we did not provide such information as we would like the participants to judge the recommendations without any influence on our part. However, our results show that providing such information is essential for composite recommendations to be accepted.

Understanding the impact of composite refactorings on the code's design also proved to be a fundamental factor. Participants made constant mentions of design principles and quality attributes that were improved. The quality attributes that were frequently mentioned are the following: Legibility, Maintainability, and Reusability. To our surprise, legibility was the most cited quality attribute, even for when removing smells like God Class and Feature Envy, which usually involves significant changes to the code design. Besides that, there were mentions to design principles such as cohesion, coupling and separation of responsibilities.

Proposed composites should be intuitive. We also observed that the more intuitive the refactorings are, the more likely the developer is to accept

[3]`https://refactoringheuristics.github.io/`

them. This is not always possible to achieve because in some cases the removal of a smell may require composite refactorings that are not always easy to understand, especially for less experienced developers. Therefore, as discussed above, providing information about the code design and the impact of refactorings is essential. For less experienced developers, it may also be necessary to explain what type of design change each recommended refactoring will perform in the code.

Resulting design should not be worse than before. Even after the composite refactoring, the developer may consider that the code design has not improved, or became worse than before. This may happen when the composite removes a smell but introduces others. Such scenario may occur with God Class removal recommendations, for example. While the God Class refactorings have the potential to improve the separation of concerns, in some cases they may also result in the introduction of Lazy Class smells. Finally, the recommendation technique must also be able to remove all relevant smell occurrences from the refactored class. Developers are often not satisfied when a particular smell is removed, but others remain in the class. Thus, it is fundamental to assess the impact of a composite before its recommendation. In our case, we had heuristics for only three smell types, so we were unable to identify and remove other types of smells that already exist or that have been introduced during refactorings.

Developer-driven customization is fundamental. Developers can often disagree with the smells detected by the recommendation technique. They can also believe that the smell has not been completely removed after the recommended composite. This occurs because code smell detection is highly sensitive to the developer (Hozano *et al.* 2017, Pecorelli *et al.* 2020) and to other contextual factors (Oliveira *et al.* 2017, Oliveira *et al.* 2020). Thus, we believe that customized detection strategies should be used for each project or for each development team. There are, indeed, techniques based on Machine Learning that try to provide customized smell detection (*e.g.*, (Hozano *et al.* 2017)). However, their effectiveness is still far from ideal.

In conclusion, the factors discussed above answer our research question RQ2. They are related to technical and human issues. Such factors can be useful to improve existing techniques and to propose new techniques, since they show what developers believe is important in a composite refactoring recommendation technique.

6.7

Threats to Validity

This section presents some threats that could limit the validity of our findings. For each threat, we present the actions taken to mitigate their impact on the research results. The first threat to validity is related to the number of participants in the study. We have selected a sample of 12 participants, which may not be enough to achieve conclusive results. Nevertheless, our data analysis was mainly based on a qualitative method that does not require a large number of participants to achieve scientifically relevant results (Yin 2015).

The second threat is related to possible misunderstandings during the study. Participants may have conducted the study differently from what we asked. To mitigate this threat, we wrote thorough instructions in a web page and encouraged them to reach us in case of any doubt. We highlighted that our help would be limited to only clarifying the study in order to avoid bias in the results. In addition, the open answers may have been biased by the objective question, since it allowed only three response options. Nevertheless, our qualitative analysis involving the content of the open answers provided evidence that they were not influenced by the objective question.

In this experiment, we opted for not comparing our recommendations with a baseline. The reason is that existing refactoring recommendation techniques do not detect the same types of code smells that we are using in this study. Moreover, the detection strategies for the smell types in common are different. As a result, the baseline technique could detect different types of code smells in the classes used in our experiment. To mitigate this threat, we based our study on an in-depth qualitative data analysis.

Finally, there is a threat concerning the selected classes and composites. The complexity of the code and the refactorings may have caused the participants to not perform the experiment properly. To mitigate such threat, we described each class thoroughly, and we were very careful to explain what was going on on each step of the generated composites. In addition, we only selected participants with a minimum knowledge about Java, code smell and refactoring.

6.8
Conclusion

In this chapter, we proposed and evaluated three refactoring heuristics for removing code smells. Towards their evaluation, we executed and reported a quasi-experiment. Results show that the heuristics are promising, leading to interesting and well-evaluated recommendations. Such results encourage the use of our heuristics for the creation or improvement of recommendation systems.

Although we got a high number *positive* answers, we still got a reasonable number of *intermediate* ones (20% of all the answers). Thus, it is worthwhile to work towards the reduction of this number since we could increase the developers' satisfaction. To do this, we can explore more of the removal patterns (Section 6.3) and the guidelines (Section 6.6.3). As a future work, we intend to build and evaluate a composite refactoring recommendation tool based on findings from this study and also based on the existing literature.

7
OrganicRef: Towards Effective and Context-Sensitive Refactoring of Features

Software projects are commonly composed by multiple features, which represent functional and non-functional requirements of the project. Each feature should be properly modularized in the project's design. However, some design decisions may result in Design Problems (DPs) caused by the incorrect modularization of features. An example of such DPs is the Scattered Feature, which occurs when a feature is scattered in multiple non-cohesive elements. When DPs are not properly handled, the project may become difficult to maintain and evolve. Therefore, developers need to remove existing DPs through refactorings. However, deciding where and how to refactor remains a challenging task.

As we observed in Chapter 5, developers often perform refactorings with little to no impact on design quality. Not rarely, the inherent complexity of DPs removal may considerably hamper developers in identifying and applying the most effective sequences of refactorings. Moreover, developers tend to avoid the complete redesigning of a project. In fact, they tend to focus on refactoring delimited contexts, which are often related to the features being changed or improved. Therefore, developers would benefit from context-sensitive refactoring recommendations. Nevertheless, there is still a lack of recommendation techniques that address aforementioned challenges.

Therefore, in this chapter we took into consideration our findings from Chapters 3, 4, 5 and 6 to propose the *OrganicRef* technique. *OrganicRef* is intended to help developers in spotting and refactoring feature-related DPs in delimited contexts. The DPs are identified through information extracted from the project's design and source code. *OrganicRef* uses a topic modeling algorithm for finding existing features in the project. Then, *OrganicRef* combines features information with internal quality measures and code smells to find DPs. For creating refactoring recommendations, *OrganicRef* relies on a new refactoring recommendation strategy, which combines refactoring heuristics with search-based optimization. We evaluated *OrganicRef* with an empirical study involving open-source projects. Our results show that, when compared to a baseline – which includes part of the heuristics of Chapter 6, *OrganicRef* significantly improves the design quality of delimited contexts through effective refactoring recommendations. The implementation of *OrganicRef* as well as the study data are fully available for researchers and practitioners.

The main results presented in this chapter were submitted as a full paper to the *International Conference on Software Maintenance and Evolution* (ICSME). In this chapter we present analyzes and results that were not included in such submission. Therefore, in the future we intend to submit an extension of the ICSME paper to an international software engineering journal.

7.1

Introduction

Design Problems (DPs) occur when stakeholders make decisions that negatively impact quality attributes such as modifiability, modularity, and the like (Li, Avgeriou and Liang 2015, Lim, Taksande and Seaman 2012). Software systems may be discontinued or redesigned when DPs are allowed to persist (MacCormack, Rusnak and Baldwin 2006). In addition, the introduction of DPs is linked to: (1) the rejection of contributions in open source projects (Oliveira, Valente and Terra 2016), and (2) increased costs in industrial software projects (Curtis, Sappid and Szynkarski 2012). Therefore, DPs should be properly handled by software developers.

The identification of DP usually occurs through symptoms such as internal code measures and code smells (Sousa *et al.* 2018). Refactoring (Fowler 1999) is a practice adopted by many developers to remove DPs. Nevertheless, deciding where and how to refactor is far from trivial. Software projects often suffer massive changes, preventing their developers from keeping track of the source code locations impacted by DPs. Moreover, there is evidence that even when the locations of DPs are known, refactorings performed in practice may be unable to completely remove them (Chapter 5). In fact, developers consider that better techniques are necessary for tasks such as identification and removal of DPs (Rebai *et al.* 2020, Lim, Taksande and Seaman 2012, Ernst *et al.* 2015).

Given such a need, there are multiple techniques for assisting developers to identify and remove DPs through refactoring recommendations (Rebai *et al.* 2020, Alizadeh *et al.* 2019, Ouni *et al.* 2017, Lin *et al.* 2016, Xiao *et al.* 2016). There are also guidelines for building refactoring recommendation techniques and tools (Tsantalis, Chaikalis and Chatzigeorgiou 2018, Bavota *et al.* 2014). However, our previous chapters have shown the importance and necessity of requirements that are still not widely met by existing techniques.

Therefore, in this chapter we propose a new recommendation technique called *OrganicRef*. Our proposed technique aims to address three key requirements as described below.

Consideration of Heterogeneous Information. *Refactoring recommendations should be generated and provided based on multiple and diverse symptoms.* Each DP is usually related with multiple symptom types, which should be considered for effectively detecting DPs and generating refactoring recommendations (Oizumi *et al.* 2020, Oizumi *et al.* 2019, Oizumi *et al.* 2016). Moreover, before refactoring, developers need to understand the relations of such symptoms with DPs and their negative consequences (Sousa *et al.* 2018, Sousa *et al.* 2017). Without understanding such relations, the developer may not be confident enough to conduct refactorings.

Context-Sensitive Detection. *Recommendation techniques should provide mechanisms to filter the recommendations to a specific context.* To provide refactoring recommendations, a technique need to rely on the detection of DP symptoms. However, detecting DP symptoms and generating recommendations for the whole project is not an effective strategy (Oizumi *et al.* 2019, Alizadeh and Kessentini 2018, Rebai *et al.* 2020, Vidal *et al.* 2019). In fact, developers usually avoid changing code elements that are out of their context of interest (Alizadeh and Kessentini 2018, Alizadeh *et al.* 2019c). Therefore, recommendations should be focused in a specific context, helping developers to spend less effort with design improvements. An example of context is the set of elements being changed in a task. In such a context, the changed elements are usually determined by the features being introduced or changed in the task. Therefore, the recommendation technique should be able to provide recommendations aimed at improving such elements and their respective features.

Effective Recommendations. *Developers need assistance for effectively removing DPs through refactorings.* The removal of a DP usually involves the execution of a sequence of multiple refactorings (Oizumi *et al.* 2020, Sousa *et al.* 2020a, Cedrim 2018). Despite refactoring sequences being widely investigated in the literature – e.g., (Brito, Hora and Valente 2020), (Bibiano *et al.* 2020), (Bibiano *et al.* 2019), (Sousa *et al.* 2020a), there is evidence that many refactorings performed in practice are not effective (Cedrim 2018, Rebai *et al.*

2020). As a result, DPs end up not being completely removed. Moreover, some refactorings may even worsen the design quality (Sousa *et al.* 2020a, Cedrim 2018). Some of the reasons for the lack of effectiveness are that developers usually (1) perform incomplete refactorings (Bibiano *et al.* 2019, Bibiano *et al.* 2020) and (2) select combinations of refactorings that are not the most effective for their context (Rebai *et al.* 2020, Alizadeh and Kessentini 2018).

OrganicRef is able to work with either developer-defined or auto-detected contexts. The DPs are detected through information extracted from the project's design and source code. From the project's design, *OrganicRef* uses a topic modeling (Silva, Galster and Gilson 2021) algorithm to extract the distribution of features across the project's elements. Then, it relies on the source code to collect quality measures and code smells. To provide effective refactorings, we have designed a new refactoring recommendation heuristic based on feature modularity. *OrganicRef* combines our new heuristic with two heuristics inspired by state-of-the-art techniques.

Finally, we use search-based optimization to create and evolve multiple possible recommendations. Search-based optimization is a discipline that employs search algorithms to find (sub-)optimal solutions for complex problems (Mohan, Greer and McMullan 2016). In this study, we selected two search-based algorithms: NSGA-II (Deb *et al.* 2002) and MOSA (Ulungu *et al.* 1999, Fraire *et al.* 2020, Kessentini, Dea and Ouni 2017). Such algorithms showed good results in recent refactoring recommendation studies (e.g., (Kessentini, Dea and Ouni 2017) and (Alizadeh *et al.* 2019b)). Moreover, they were the best performing ones in our preliminary evaluations. Given the characteristics of *OrganicRef*, we selected *Feature Overload* and *Scattered Feature* (Garcia *et al.* 2009) as the target design problems of our evaluation. *Feature Overload* occurs when a design element implements multiple unrelated features. Finally, *Scattered Feature* is the result of a feature being implemented by different and non-cohesive design elements.

To enable *OrganicRef* evaluation, we implemented a reference tool and conducted an empirical study involving six industry-strength open-source projects. Our evaluation relied on both quantitative and qualitative analysis. The results show that *OrganicRef* is able to significantly improve feature modularization even when information about features is not fully accurate. We also observed that NSGA-II presented the best results based on our evaluation criteria. It was able to consistently reduce the number of DP symptoms and the lack of cohesion in the context's elements of all target projects. Thus, NSGA-II presented the best improvement of feature modularization.

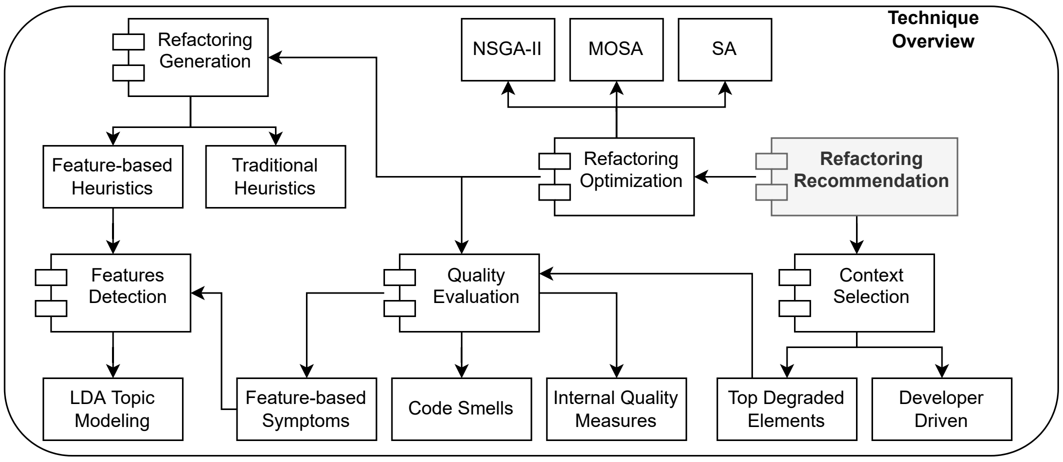


Figure 7.1: Overview of the *OrganicRef* technique

In a nutshell, in this work we: (1) proposed a new refactoring recommendation technique focused on key requirements from the literature; (2) implemented an open-source reference tool, which is fully available for researchers and practitioners; and (3) conducted an empirical study following the recommended open science policies.

7.2 OrganicRef: Components, Algorithms and Heuristics

The goal of *OrganicRef* is *to provide effective context-sensitive support for refactoring feature modularization problems.* We designed *OrganicRef* based on the results of our previous studies (Chapters 3 to 6) and existing literature about feature-related DPs and refactoring (e.g., (Rebai *et al.* 2020, Sousa *et al.* 2020a, Kessentini, Dea and Ouni 2017, Tsantalis, Chaikalis and Chatzigeorgiou 2018)). Figure 7.1 provides an overview of the core components of *OrganicRef*. The components are represented with the UML notation for components while rectangles represent their underline heuristics and algorithms. Dependencies between components, heuristics and algorithms are represented by arrows. In the next sub-sections, we describe all elements that compose *OrganicRef* as well as the relations between them.

7.2.1 Features Detection and Context Selection

One of the differences of *OrganicRef* from other similar techniques is that it goes beyond traditional symptoms. *OrganicRef* is able to detect symptoms and recommend refactorings based on information about feature modularization. Each feature is intended to represent a functional or non-functional requirement of the target project. The features may be automatically detected based on different approaches, such as using topic modeling (Oizumi *et al.* 2016, Bavota *et al.* 2013).

Using the feature detection component, *OrganicRef* is able to automatically collect semantic information which is often not possible only with traditional metrics and code smells. As a result, we expect *OrganicRef* to better identify possible DPs with heterogeneous symptoms.

We decided to include this component because of evidence from the literature that DP identification requires heterogeneous symptoms (Sousa *et al.* 2018, Sousa *et al.* 2017). In fact, there is an increasingly number of techniques relying on multiple and diverse information for generating recommendations (Nyamawe *et al.* 2019, Rebai *et al.* 2020, Yamanaka *et al.* 2021).

As illustrated in Figure 7.1, we relied on a Topic Modeling algorithm for finding features. For this task, we used the *Machine learning for language toolkit* (Mallet) (McCallum 2002)[1]. Mallet is a Java toolkit that provides the implementation of several machine learning algorithms applied to textual data. Among the available algorithms, we decided to use their Latent Dirichlet Allocation (LDA) parallel topic modeling implementation. LDA was selected because it was previously applied in multiple software engineering studies as demonstrated by a recent literature review (Silva, Galster and Gilson 2021).

Feature Identification and Association. After building a topic model, *OrganicRef* maps each topic to a feature. Each feature is represented by the most frequent tokens occurring in its corresponding topic. After that, *OrganicRef* assigns the features to the source code Types (Classes, Enumerates, and Interfaces) and Methods (including constructors). It does not consider code elements of fine granularity, such as blocks or statements, as *OrganicRef* is focused on DPs of coarser granularity. *OrganicRef* uses the textual representation of the code element for inferring up to N features, where N is an arbitrary value defined when running *OrganicRef.* In this study, after a trial and error process, we defined $N = 6$.

Context Selection. *OrganicRef* also allows the definition of a context for finding the DPs and generating refactoring recommendations. The context is defined on demand, according to the objectives of the developers. An example of context is the set of elements changed by the developer during a task. We included this component because there is evidence that developers

[1]Version *202108*.

usually avoid refactoring code elements that are out of their context of interest (Alizadeh and Kessentini 2018, Alizadeh *et al.* 2019, Alizadeh *et al.* 2019c). Moreover, developers tend to overlook long lists of symptoms and refactoring recommendations.

For this study, we defined a general purpose context selection strategy which consists of the *top ten code elements with higher chance of being impacted by DPs*. For instrumenting our selection strategy, we decided to rely on the number of DP symptoms. This decision is based on evidence that DP usually manifest themselves through multiple symptoms (Sousa *et al.* 2018, Oizumi *et al.* 2016). Therefore, this heuristic is expected to select the elements that are more likely to need feature modularization refactorings.

7.2.2

Quality Evaluation

Given the code elements provided by the previous component, in this component we conduct a quality assessment based on feature-related DP symptoms. *OrganicRef* is extensible enough for allowing the use multiple symptom types. In this study, we selected three types of symptoms, namely *Internal Quality Measures*, *Code Smells*, and *Feature-based Symptoms*.

Internal Quality Measures represent relevant characteristics for the design of a project. For the detection of *Feature Overload* and *Scattered Feature*, we selected the following measures: Coupling Dispersion, Coupling Intensity, Cyclomatic Complexity, Lack of Cohesion, Number of Clients, Number of Methods, Number of Fields, and Number of Statements. Such measures provide us with information about different design characteristics that may be impacted by the occurrence of *Feature Overload* or *Scattered Feature*.

Code Smells are surface symptoms of DPs in the source code (Fowler 1999). Smells can be automatically detected through different heuristics. In *OrganicRef*, we rely on the use of rule-based heuristics (Marinescu, 2004). The rules are based on the internal quality measures presented earlier. We selected such heuristics because they are often used by state-of-the-art techniques to find DP symptoms. In this study, we focused in the following smells types: Complex Class, Dispersed Coupling, Feature Envy, God Class, Large Class, and Lazy Class. Those smell types were selected because of their relation with feature modularization. Details about the detection rules and thresholds are available in our replication package.

Feature-based Symptoms indicate elements related to the lack of feature modularization. We defined two heuristics for feature-based symptoms, namely *Feature Concentration* and *Feature Dispersion*. The selection of such heuristics is justified by their direct relation with our target DPs (i.e., *Feature Overload* and *Scattered Feature*). The heuristics for finding feature-based symptoms in *OrganicRef* are based on the output generated by the *Feature Detection* component. Below we present a short description of our detection heuristics:

- *Feature Concentration.* In this heuristics *OrganicRef* analyzes the features detected for each class in the project. Then it selects the ones that have a number of features higher than the median number of features per class in the project. Such classes are marked as candidates for the *Feature Overload* DP.
- *Feature Dispersion.* For finding Scattered Feature candidates, this heuristics iterates over all features detected in the project. Then, for each feature, *OrganicRef* finds the classes implementing it. Among such classes, the heuristic selects the ones in which the feature is not the predominant one. The predominance is based on the probability of association between the feature and the class. This probability is determined by the Topic Modeling algorithm.

7.2.3 Refactoring Generation Heuristics

This component is responsible for generating the refactorings that will improve feature modularization in a given context. *OrganicRef* uses this component to generate an initial set of recommendations based on multiple heuristics inspired by existing state-of-the-art refactoring recommendation techniques (Bavota *et al.* 2013, Tsantalis, Chaikalis and Chatzigeorgiou 2018, Fokaefs *et al.* 2011, Oizumi *et al.* 2020). Although the selection of heuristics is flexible, we selected the following initial set of heuristics for this study:

Move Method. With this heuristic *OrganicRef* finds the suitable class for moving a method by counting the number of calls and accesses to fields. For instance, let us assume that a particular method m calls 3 methods and accesses 2 attributes from class A. In this case, the bonding factor of m to A is 5. The same method m calls 4 methods from the class B, leading to a bonding factor of 4. Thus, our heuristic generates a Move Method recommendation for moving m to A. We also store the information that B is another candidate for the move. As a result, we can later, during the optimization process, change the target of this Move Method from A to B.

This Move Method heuristic was recently proposed by Oizumi et al. (Oizumi *et al.* 2020) and showed good results in their evaluation. The rationale behind this heuristic is that a class with the highest bound factor can be the best candidate for cohesively receiving the moved method.

Extract Class. The Move Method refactoring may not be the best option for all cases. Therefore, we also included an Extract Class heuristic. For generating Extract Class refactorings, *OrganicRef* uses an heuristic inspired by the one proposed by Fokaefs et al. (Fokaefs *et al.* 2011). In summary, *OrganicRef* uses a hierarchical clustering algorithm for finding groups of fields and methods that can be extracted. The heuristic starts by including each method and fields in separated clusters. Then, the clusters are compared and merged based on the Jaccard distance metric. This process continues through multiple iterations until no clusters can be merged. As a result, our heuristic recommends extracting the smaller clusters of fields and methods.

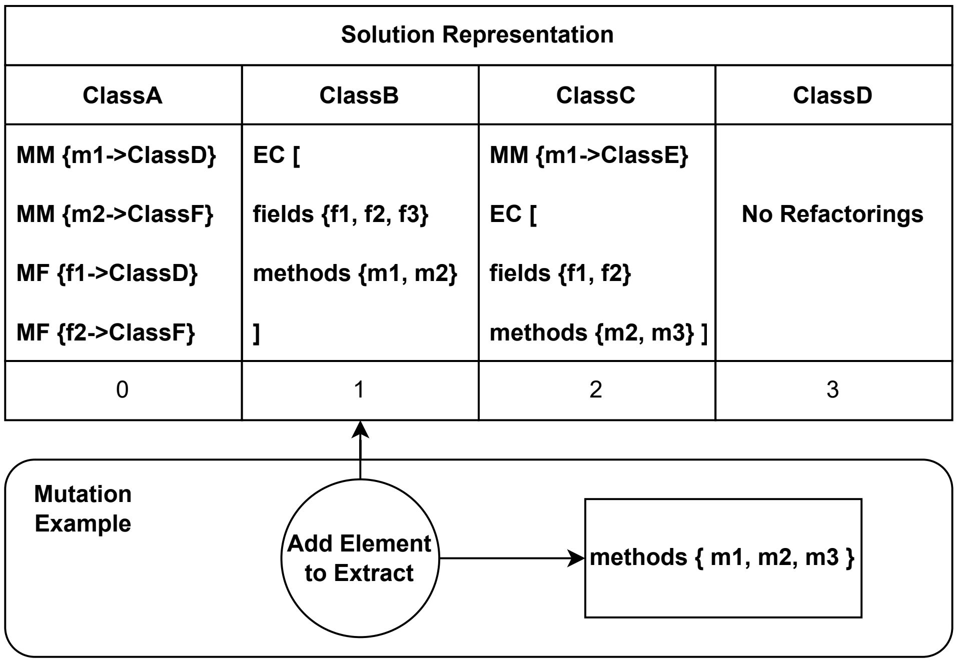
Feature-based Move Element. This heuristic is inspired by the *MethodBook* technique (Bavota *et al.* 2013), which relies on topic modeling for finding Move Method candidates. Similarly to them, *OrganicRef* identifies multiple candidates for receiving a method based on feature similarity. In this case, we only include as candidates the classes for which the predominant feature is the same of the method. The heuristic generates the Move Method targeting a randomly selected class among the available candidates. The other candidates are store for being used later in the refactoring optimization process.

Feature-based Extract Class. OrganicRef also includes a heuristic for generating Extract Class refactorings based on feature information. This heuristics selects the classes that contain two or more features and recommends the extraction of methods implementing the non-predominant features of the class. Besides the methods, it also selects the fields used only by the extracted methods.

Aforementioned heuristics were selected because they generate well suited refactorings for feature modularization problems. We are aware that other refactoring types could be used for improving modularization. Nevertheless, we decided to focus on a small set of refactorings to facilitate the evaluation and improvement of *OrganicRef*.

7.2.4
Search-based Refactoring Optimization

OrganicRef is also able to apply a search-based algorithm (Harman *et al.* 2012) for recommending to the developer the refactorings that do not include



Solution Representation			
ClassA	ClassB	ClassC	ClassD
MM {m1->ClassD} MM {m2->ClassF} MF {f1->ClassD} MF {f2->ClassF}	EC [fields {f1, f2, f3} methods {m1, m2}]	MM {m1->ClassE} EC [fields {f1, f2} methods {m2, m3}]	No Refactorings
0	1	2	3

Figure 7.2: Example of solution representation and search operator

violations and that best improve the source code structure according to our set of quality characteristics. *OrganicRef* builds an initial population based on the combination of aforementioned heuristics in a given context. This approach differs from other approaches that usually build a random initial population. The advantage of our approach is that the optimization already start with “good” solutions. Therefore, the optimization effort may be lower.

Our optimization problem is represented by a refactoring sequence vector. Each vector position contains the refactorings applied to a context’s element. Each element may contain zero to many recommended refactorings. Figure 7.2 shows an illustrative example of solution. It is possible to see that position zero corresponds to the refactorings for *ClassA*, which are two move methods and two move fields. We opted for this non-conventional representation because it allows *OrganicRef* to focus the optimization effort in the selected context.

We use search operators specific to feature modularization to change and improve solutions, as presented next.

Replace Refactoring Type will try to change the type of refactoring being applied to a given element. For instance, let’s take the solution presented in Figure 7.2. The Move Method and Move Field refactorings for *ClassA* can be replaced by an Extract Class refactoring with the same methods and fields that were being moved. **Add refactoring to Element** is intended to complement the refactorings for a given element. In the example of Figure 7.2, this operator is applied for including method *m3* to the Extract Class refactoring at *ClassB*.

Remove Refactoring aims at removing a refactoring from the solution. It can remove an entire refactoring or part of it. In the case of *Extract Class*, it can remove one of the elements that were being extracted. Finally, **Change Move Target** changes the target of a Move Method based on the candidates for receiving it (see Move Method heuristics above). The new target for the Move Method is randomly selected when there are more than one candidate.

For evaluating the solutions, we defined five objective functions. Four of them are related to feature modularization and one aims at reducing the number of recommended refactorings. All of them are minimization objectives and are defined next.

Quality Measures. Similarly to many of the search-based refactoring techniques (e.g., (Alizadeh *et al.* 2019)), one of our objective functions is composed by the sum of quality measures divided by the number of measures. In our case, we selected the measures presented in Section 7.2.2. The only exception is the Lack of Cohesion measure, which is in a separated objective.

Lack of Cohesion. We decided to consider lack of cohesion as a separated objective due to its direct relation with feature modularization. Despite being often criticized due to its lack of accuracy, it provides a complementary perspective for our feature modularization technique.

Density of Symptoms. The density of symptoms is also often applied as a objective function for search-based refactoring techniques. In *OrganicRef*, we follow the same approach of Kessentini *et al.* (Kessentini, Dea and Ouni 2017), defining the density of symptoms objective function as *the number of Symptoms After Refactoring* divided by *the number of Symptoms Before Refactoring.*

Number of Features per Element. Since the goal of *OrganicRef* is to improve feature modularization, we included this objective function that measures the impact of each solution on the number of features per element in the project. This objective is similar to the previous one and is calculated as *the sum of features per Class after refactoring* divided by *the sum of features per class before refactoring.*

Number of Refactorings. Another import aspect of the refactorings is to achieve the desired improvement with the smaller number of refactorings.

Therefore, this objective function is calculated as the number of refactoring operations recommended by the solution.

OrganicRef allows the use of any optimization algorithm provided by the JMetal framework. It is also possible to implement custom optimization algorithms based on the reusable structure of JMetal. Therefore, during the conception of *OrganicRef*, we could test different options of optimization algorithms.

After our tests, we selected two algorithms for creating our refactoring recommendation strategies. The first algorithm is the Multi-Objective Simulated Annealing (MOSA). MOSA is a multi-objective version of Simulated Annealing. Besides presenting good results in our preliminary evaluations, MOSA has also presented better results than SA in a closely related study (Kessentini, Dea and Ouni 2017). In addition to MOSA, we also selected the *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) (Deb *et al.* 2002).

The aforementioned search operators were applied as mutation operators in NSGA-II, whereas the crossover operator consists in randomly exchanging vector elements between two solutions to generate two new solutions. To our surprise, NSGA-II was more effective in our tests than NSGA-III (Deb and Jain 2014). It was not expected because NSGA-III tends to present better results when we use more than three objective functions. As we previously described, *OrganicRef* relies on the use of five objective functions. Despite having selected these two algorithms, future studies can be conducted to carry out more in-depth comparisons between the different optimization algorithms.

Besides the strategies based on MOSA and NSGA-II, we created a third strategy, which we call SIMPLE. This strategy consists of using our refactoring generation heuristics without optimization. The idea behind this strategy is to provide a comparison parameter for the solutions generated by the optimization-based strategies. The SIMPLE strategy always creates a set of five different solutions. The first four solutions are the result of applying each of our refactoring generation heuristics in isolation. For example, one of the solutions will be the result of applying our *Move Methods heuristic* for all elements within the selected context. The last solution is the result of randomly combining the refactoring heuristics (e.g., Move Methods for *ClassA*, Extract Class for *ClassB*, etc). Finally, we also defined a BASELINE strategy which is similar to the SIMPLE strategy. The only difference is that the BASELINE does not use feature-related heuristics for generating the recommendations. Our intention with this heuristic is to evaluate our feature-driven refactoring heuristics.

7.3 Study Design

For evaluating the proposed technique, we designed a multi-case study based on industry-strength open source projects. The goal of this study is *to evaluate the effectiveness of refactoring sequences recommended by OrganicRef to improve feature modularization*. To guide our evaluation, we defined the following research questions.

RQ1. What is the effectiveness of feature-driven strategies when compared to a baseline?

The idea behind RQ1 is to evaluate whether our refactoring heuristics in combination with search-based optimization can improve the recommendation of refactorings for feature modularization. Therefore, we want to verify if the *OrganicRef* feature-driven heuristics and optimization operators can significantly improve the quality of the recommendations compared to non-optimized ones.

For answering RQ1, we evaluated the effectiveness of search-based algorithms for generating refactoring recommendations when compared to a baseline (see Section 7.2.4). These rule-based strategies were selected because they presented promising results for the removal of *Feature Overload* and *Scattered Feature* symptoms in previous studies (Tsantalis, Chaikalis and Chatzigeorgiou 2018, Oizumi *et al.* 2020, Colanzi *et al.* 2014).

RQ2. What are the characteristics of solutions provided by *OrganicRef*?

For answering RQ2, we compared the solutions generated by each strategy. Such a comparison was made based on the fitness of generated solutions and on a manual evaluation performed by ourselves (more details on Section 7.3.4). The idea behind RQ2 is to understand how the use of feature-driven heuristics and search-based algorithms impacted on the characteristics of generated solutions. Such an investigation is necessary to understand which aspects of the technique contributed for our goal and which ones should be improved.

Table 7.1: Target projects

Project	Domain	# Commits	# PRs	Release
Dubbo	RPC framework	5356	4702	3.0.6
Fresco	Image Library	3150	407	2.6.0
Jenkins	CI Server	31686	6351	2.319.3
OkHttp	HTTP client	5072	3457	3.14.0
RxJava	Async Library	5969	3678	3.1.0
Spring Security	Security Library	10142	1865	5.6.0

7.3.1

Target Projects

We selected a set of six industry-strength open source projects for our study. For composing this set, we selected projects that (1) are implemented in the Java programming language; (2) use pull request reviews as a mechanism to receive and evaluate contributions; (3) have at least 1,000 commits; (4) are at least five years old, and (5) are currently active.

We applied the first criterion once our tool is able to work with source code written in Java. The second allowed us to identify projects potentially concerned with code and design quality. Finally, we adopted the last three criteria to avoid selecting projects with reduced complexity, barely changed, and discontinued.

After applying the aforementioned criteria, there were several candidates to filter out. Therefore, we opted by cherry-picking projects widely known and employed in the software industry. From them, we tried to compose a diverse sample of projects from different domains and purposes. We understand that exploring this diversity is important to our study because it allows us to evaluate the proposed technology over a heterogeneous set of features and design decisions that may influence the required refactorings. Besides, we also picked projects considerably varying in the number of commits, pull requests, and age. These characteristics may be associated with different levels of degradation. Table 7.1 describes the six projects selected for our study. One can see that the set of target projects includes well-known libraries and frameworks.

Selection of Releases. After selecting the target projects, we selected the releases to compose the sample for evaluation. Except for the OkHttp project, we selected the last stable release of the target projects, as presented in the last column of Table 7.1. For OkHttp, we selected the last release developed in Java.

7.3.2 Execution Settings

We executed the planned evaluation on a Linux server running over a machine having 95 GB of primary memory, 1TB of storage, and Xeon Silver 2.2GHz (20 threads) CPU.

Topic Modeling. We configured the LDA Topic Modeling algorithm in a trial and error process, achieving the following configuration: two parallel threads, 1,500 iterations, optimization interval of 10, burn-in period of 20, no symmetric alpha, 50 topics, and 0.15 as the minimum proportion of a topic. We also defined a cleaning procedure to remove any character different from letters (A-Z) and numbers (0-9). For identifying the token to be ignored, we employed the default list of stop words in English provided by Mallet combined with the list of Java keywords.

We applied above configuration for all target projects, without any specific customization. We are aware that this is not the ideal scenario, since the effectiveness of LDA Topic Modeling depends on fine-tuning its parameters for each project. Nevertheless, using a default configuration helps us to evaluate *OrganicRef* in the worst case scenario.

Recommendation Strategies. For each strategy employed, we ran 30 independent executions of *OrganicRef* by target projects, storing the results. This amount of executions was necessary for finding the best results given the non-deterministic nature of topic modeling and search-based optimization. Moreover, this practice follows guidelines proposed in the technical literature on search-based optimization (Arcuri and Briand 2014, Colanzi *et al.* 2020). Next, we present the configurations adopted for each search-based algorithm. Those configurations were initially defined based on the ones used by similar techniques (e.g., (Kessentini, Dea and Ouni 2017, Alizadeh *et al.* 2019b)) and were adapted during our tests.

NSGA-II Configuration. For running NSGA-II, we adopted the following configuration: (1) 10,240 maximum fitness evaluations, (2) crossover probability of 0.2, (3) mutation probability of 0.8, (4) five objective functions (see Section 7.2), and (5) population size of 128 solutions.

MOSA Configuration. For the MOSA algorithm, we adapted the implementation provided in the work of Fraire Huacuja et al. (Fraire *et al.* 2020). We defined the following configuration: (1) initial temperature of 100, (2) final temperature of 0.01, (3) 0.98 alpha, (4) five objective functions (see Section 7.2), and (5) population size of 128 solutions. Search operators described in Section 7.2 were applied as neighborhood operators.

7.3.3 Data Collection Procedures

For running *OrganicRef* and collecting the results of the executions in each target project, we followed the systematic and automated procedures described below.

1) Topic modeling training. Using the configurations described in Section 7.3.2, we ran the Mallet tool for training our topic models. Since our target projects are from different domains, we opted to create a single model per project. The input is composed by the source code files of the target project and the result is topic model. In the next step, we use the topic model of each project in *OrganicRef* for the identification of features based on topic inference.

2) *OrganicRef* execution. For this study, we implemented all the refactoring recommendation strategies in the same tool. It means that *OrganicRef* contains both the baseline and the feature-driven strategies, which allows us to perform comparisons based on the same quality measures. For any recommendation strategy, *OrganicRef* provides the list of solutions together with their respective recommended refactorings and fitness values. Moreover, it provides information about the design quality – i.e., measures and symptoms – before and after performing the recommended refactorings.

For each target project, we ran *OrganicRef* and collected the aforementioned results for 30 executions of each recommendation strategy evaluated in this study.

7.3.4 Quantitative and Qualitative Analysis

For answering our research questions, we defined the following data analysis procedures.

Selection of best solutions. Given that our analysis is based on five objective-functions, for all strategies, we selected the non-dominated solutions (Coello *et al.* 2007) across the 30 executions. The non-dominated solutions are those for which there is no other solution that presents better fitness to all objectives.

Comparing Strategies. For comparing MOSA, NSGA-II and the baseline across all projects, we relied on multiple quality impact criteria. Such criteria were used to evaluate the generated solutions based on the differences before and after applying the recommended refactorings. For this purpose, we selected the following criteria: (1) number of symptoms, (2) number of features per element, and (3) lack of cohesion. For all criteria, we performed a quantitative analyses based on descriptive statistics and on the Mann-Whitney U test.

Manual Evaluation. Besides the quantitative analyses, we selected a reduced subset of recommendations to be manually analyzed by four collaborators. The collaborators were selected from our network of contacts and have previous experience conducting research related to DPs and refactoring. To avoid too much cognitive effort in understanding different projects, we selected only recommendations addressing the OkHttp project. This project was selected because of our collaborator's previous experiences studying it. Moreover, as we will discuss in Section 7.4.2, the different strategies evaluated in this study presented similar results for OkHttp. Therefore, we consider that this project serves as an appropriate benchmark for *OrganicRef*.

Analysis Procedures. For conducting the analysis, we provided each collaborator with three solutions. Two of them were generated by the MOSA and NSGA-II strategies. The other one was generated by the BASELINE strategy and was included only to verify that the collaborators did not carry out a biased evaluation. Each collaborator received the three solutions in different orders. Moreover, they were not aware of which strategy generated each solution.

We asked the collaborators to evaluate the solutions in the provided order focusing in three quality criteria, namely soundness, viability, and quality impact. Besides that, we informed them that the solutions were aimed at improving feature modularization. They were allowed to spend as much time as necessary for doing the analysis without communicating to each other. Finally, their feedback was sent through a discursive response form.

By performing the aforementioned analysis, we could understand better the results of the quantitative analysis, improving the discussion of the study findings.

7.4 Evaluation Results

7.4.1

On the Quality Impact of Feature-driven Strategies

For answering RQ1, we analyzed the effectiveness of four refactoring recommendation strategies from different perspectives. As described in the previous sections, we are comparing the use of feature-driven strategies to a rule-based strategy (BASELINE).

In Table 7.2 we summarize three quality criteria for non-dominated solutions generated with each strategy. The BASELINE and SIMPLE strategies are represented by the BL and SP acronyms, respectively. All the quality criteria presented in Table 7.2 are calculated exclusively for the selected contexts (i.e., top 10 degraded elements of each project). All the presented values are the medians obtained from the non-dominated solutions.

Table 7.2: Median quality impact of non-dominated solutions on the top 10 degraded elements

Project	**# Smell Difference**				**# Features Difference**				**LCOM Difference**			
	BL	**SP**	**MOSA**	**NSGA-II**	**BL**	**SP**	**MOSA**	**NSGA-II**	**BL**	**SP**	**MOSA**	**NSGA-II**
Fresco	-6	**-23**	-16	**-23**	-1,5	**-7**	-4	**-7**	0,0468	-0,2043	-0,0723	**-0,2212**
RxJava	-15	-35	-13	**-37**	-2	-10	-5,5	**-11**	-0,0544	0,0151	0,0415	**-0,0747**
Jenkins	-24	-27	-21	**-29**	-4	-5	-5	**-6**	-0,0261	-0,0923	-0,0921	**-0,1378**
Spring Security	-5	**-19**	-17	**-19**	0	**-5**	**-5**	-4	0,0007	-0,0913	-0,0469	**-0,1145**
Dubbo	-11	**-24**	-11	-23	-1	**-2**	**-2**	-1	0,1247	-0,1452	-0,0475	**-0,1972**
OkHttp	-6,5	-26	-20	**-27**	1	-3	**-4**	**-4**	-0,0741	-0,2193	-0,1549	**-0,2541**
All	-10,5	**-26**	-16	**-26**	-2	-5	-4	**-6**	-0,0202	-0,114	-0,066	**-0,151**

The second main column of Table 7.2 presents the difference on the number of DP symptoms after applying the recommended refactorings. In the third main column we present the difference on the sum of features per element. Finally, the last main column contains the difference of the Lack of Cohesion measure in the selected contexts. For all criteria, a negative value means that the evaluated quality indicator improved after the refactorings. We highlight in bold the best results of each criterion in each project.

DP Symptoms Reduction. For most projects, it is possible to observe that the SIMPLE and NSGA-II strategies presented the best results regarding DP symptom reduction. MOSA was not the best in any of the projects. However, with the exception of Dubbo, its results were very close to the ones presented by SIMPLE and NSGA-II. The BASELINE strategy, on the other hand, only presented a considerable reduction of symptoms for the Jenkins project.

We also applied the Mann Whitney U Test to compare the medians for each strategy. The results show, with 95% of confidence, that SIMPLE and NSGA-II are statistically different from the ones for BASELINE and MOSA.

Impact on the Number of Features per Element. For the feature reduction criterion, the differences between the compared strategies were smaller. Nevertheless, the NSGA-II strategy presented the best results for four out of six projects. On the other hand, MOSA and SIMPLE outperformed NSGA-II results for the Dubbo and Spring Security projects. Finally, besides the BASELINE being the worst performing strategy in all cases, it also increased the number of features per element in the OkHttp project.

Lack of Cohesion Reduction. As far as feature modularization is concerned, another key evaluation criteria is the Lack of Cohesion measure. It is possible to observe in Table 7.2 that NSGA-II was the best performing strategy for all evaluated projects. In this criteria, the results for SIMPLE and MOSA were not as close to NSGA-II results as in the previous criteria.

Impact on Coupling Measures. Besides the evaluation criteria presented in Table 7.2, another relevant aspect for feature modularization is the coupling of the refactored elements. Therefore, in Figure 7.3, we summarize the impact of each strategy in two coupling measures namely, Coupling Intensity and Coupling Dispersion. The former is calculated as the higher number of method calls and field uses that a class has with another one, while the latter counts the number of different classes that a class depends on. The values presented in Figure 7.3 are the medians for all projects.

For Coupling Intensity, MOSA presented the best impacting with a median intensity reduction close to six. On the other hand, when we consider Coupling Dispersion, NSGA-II outperformed all the other strategies. Such a result is aligned with the previous one, since a higher coupling dispersion may be linked to the lack of feature modularization.

Finding 1. NSGA-II presented the best results based on our evaluation criteria. It was able to consistently reduce the number of symptoms and the lack of cohesion in the context's elements of all target projects. Moreover, in most projects it provided the highest reduction of features implemented by context's elements, benefiting feature modularization.

Computing Time. As presented in Table 7.3, we also measured the median computing time for each strategy. As we expected, the computing time for BASELINE and SIMPLE is negligible, taking less than half a minute for all cases. On the other hand, the median computing time for MOSA was far from ideal. With the exception of the Fresco and OkHttp projects, for all cases MOSA took nearly one hour. For the Dubbo project, it was 1 hour and 14

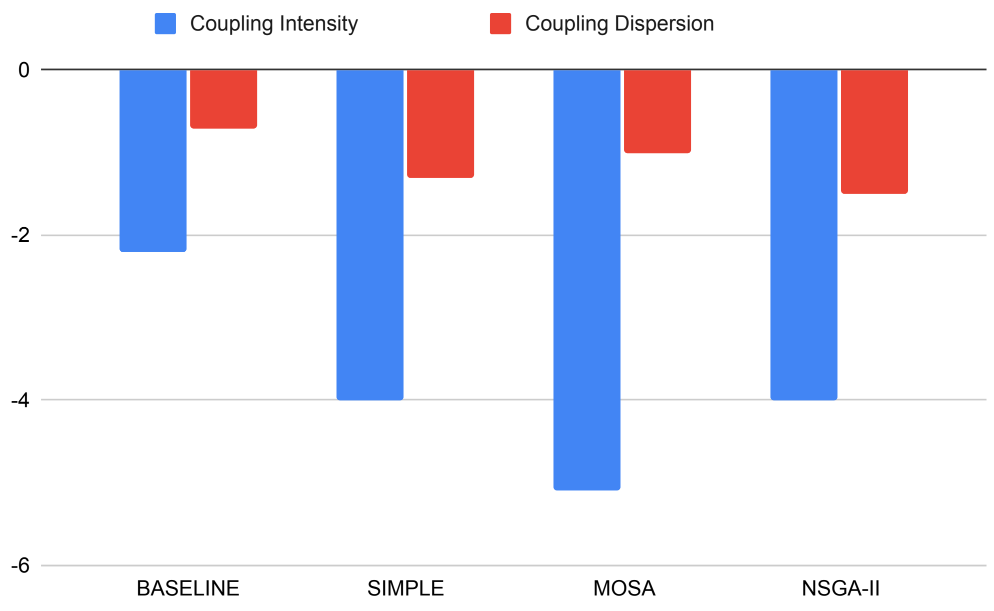


Figure 7.3: Median impact of non-dominated solutions on coupling measures minutes.

If we think about integrating *OrganicRef* into a software production flow, the computing times for MOSA can be considered prohibitive. In this sense, the NSGA-II strategy proved to be much more promising. It took less than four minutes in the best case, and less than sixteen minutes in the worst case. Although these values are still high when compared to the BASELINE and SIMPLE strategies, NSGA-II impact on a production flow would be much smaller than that presented by MOSA. In addition, this waiting time can pay off given the positive impact caused by the recommendations generated by the NSGA-II (see Section 7.4.1).

Finding 2. While the computing time is still not ideal, the NSGA-II strategy can be viable given its consistently positive impact.

Table 7.3: Median computing time (in minutes) for generating solutions

Project	**Computing Time (Minutes)**			
	BASELINE	**SIMPLE**	**MOSA**	**NSGA-II**
Fresco	0,0478	0,0766	22,1735	3,9935
RxJava	0,1502	0,1879	63,1804	14,7194
Jenkins	0,1438	0,1978	64,5497	12,1198
Spring Security	0,1139	0,1576	58,7528	13,3737
Dubbo	0,1622	0,2133	74,6844	15,9935
OkHttp	0,0508	0,1474	21,0723	3,2373
All	0,1288	0,1774	62,2587	12,4134

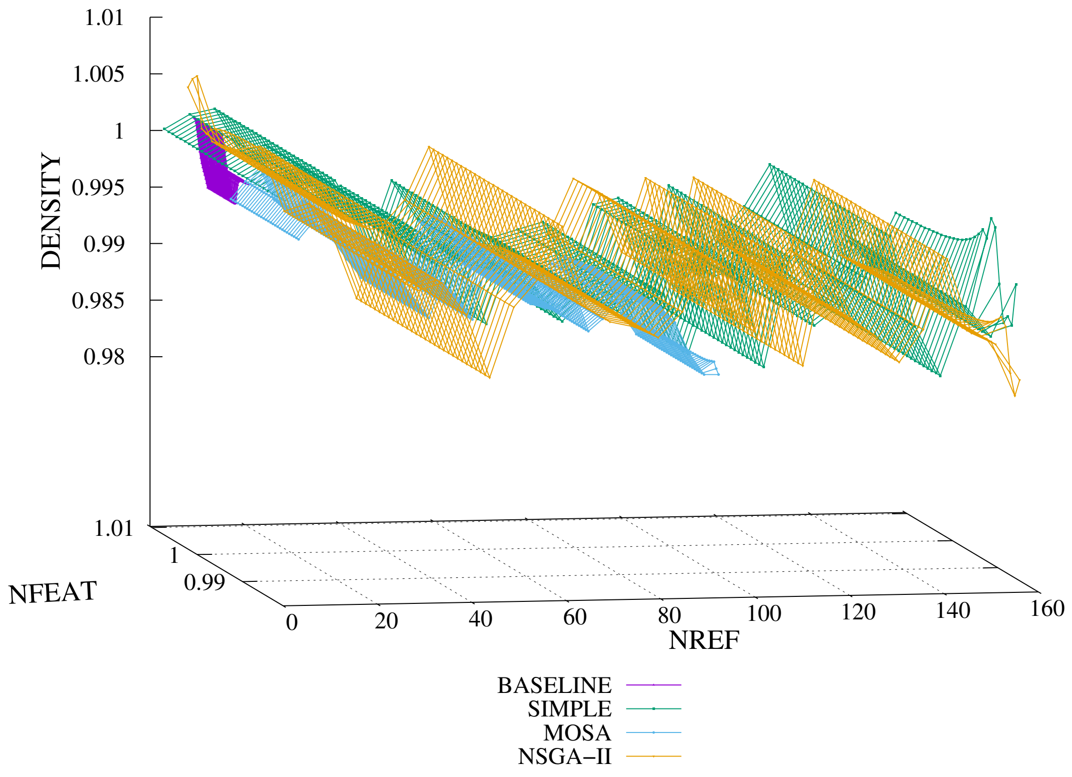


Figure 7.4: Partial view of the solution space for the Fresco project

7.4.2
Solution Space Analysis

Given the results observed for our quality impact criteria, in this section we analyze how the solutions obtained by each strategy are disperse in the solution space. For doing so, we selected three objective functions, namely Number of Features per Element (NFEAT), Number of Refactorings (NREF), and Density of Symptoms (DENSITY). The NFEAT and DENSITY objectives are direct indicators of impact on feature modularization. The NREF objective provides an overview of the required effort for applying each solution in practice. We discuss all the objectives in the next section.

Solution Space for Fresco. In Figure 7.4 we present a partial view of the solutions generated by each strategy. In this graph we position the solutions based on the three objective functions: NFEAT (x axis), NREF (y axis), and DENSITY (z axis).

We can observe that the solutions generated by the NSGA-II strategy are close to the ones generated by the SIMPLE strategy. Nevertheless, when we focus our analysis to the best solutions – i.e., the ones with objectives closest to zero, it is possible to see that NSGA-II is significantly better. In fact, there are only NSGA-II solutions with the simultaneously minimization of Density

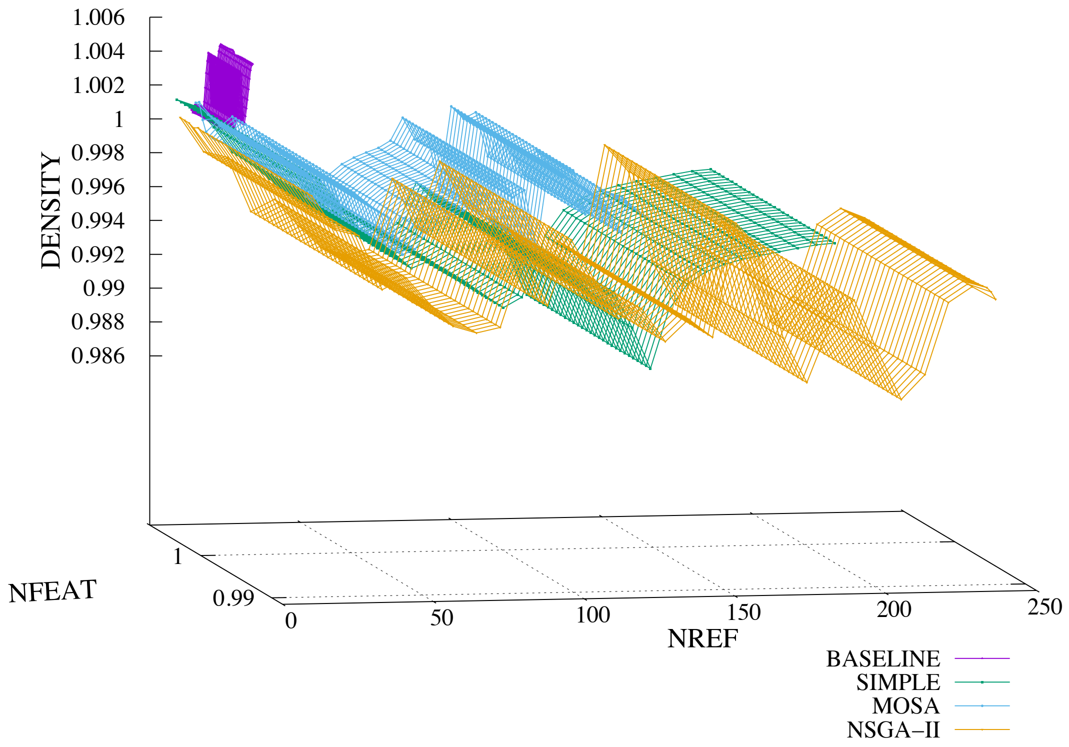


Figure 7.5: Partial view of the solution space for the Dubbo project

and NFEAT. Besides that, NSGA-II is able to improve Density and NFEAT with less refactorings than the SIMPLE solutions. This result is extremely important because it shows that NSGA-II is the strategy that is most aligned with *OrganicRef*'s goal, which is to improve feature modularization with the smaller possible number of refactorings.

When we compare the solutions of NSGA-II and MOSA, we can observe that NSGA-II was able to better explore the search space, resulting in a higher diversity of solutions in terms of our objective functions. This may be explained by the fact that NSGA-II uses crossover and mutation operators to perform the intensification and diversification, which are the two driving forces of genetic algorithms (Goldberg 1989).

While the BASELINE solutions presented the best NREF results, their results on the objective "number of features" are among the worst ones. In addition, their best impact on the number of DP symptoms is far from reaching the best results of other strategies for the same objective.

Solution Space for Dubbo. Figure 7.5 presents the solution space for the Dubbo project. In this and in the next figures, we follow the same structure and present the same objectives as presented for the Fresco project.

Dubbo shows a solution space that is similar to the one observed in the

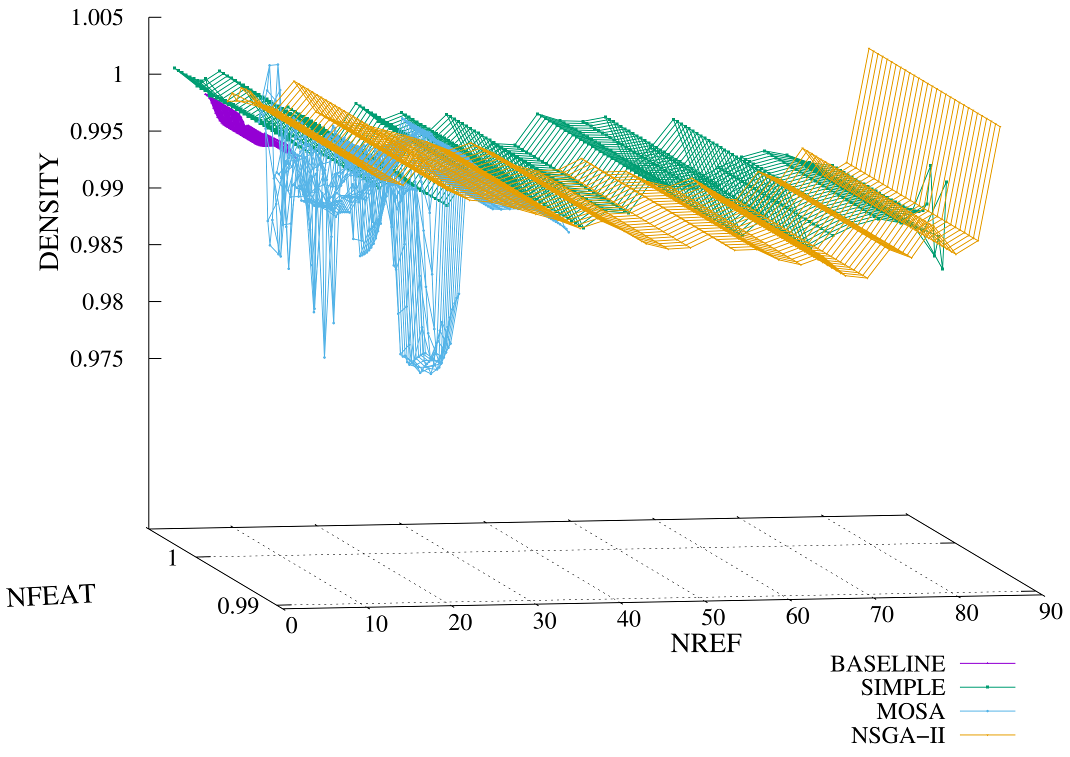


Figure 7.6: Partial view of the solution space for the Jenkins project

Fresco project. The main difference is that the performance of the BASELINE strategy was significantly worse than the performance of all other strategies. Even when considering only the solutions with smaller NREF, all other strategies presented better DENSITY and NFEAT. Finally, in this project, most MOSA solutions performed worse than NSGA-II solutions regarding the DENSITY objective.

Solution Space for Jenkins. Figure 7.6 presents the solution space for the Jenkins project. Jenkins was the only project for which all strategies presented close results regarding symptoms and features reduction. Such an outcome becomes more evident in Figure 7.6, where we can observe that solutions are much closer than in the Fresco project. It is also possible to observe that NSGA-II values for the DENSITY and NFEAT objectives are significantly worse when compared to the previous case.

When we focus our analysis to the DENSITY and NFREF objectives, it is possible to see that MOSA performed much better than NSGA-II. MOSA was able to present the best reduction of symptoms – i.e., best impact on the DENSITY objective, among all evaluated strategies. Moreover, MOSA also managed to keep the number of refactorings objective below 40 in most solutions. Such a result was also similar in the other projects.

When we look at the NFEAT objective, NSGA-II still presented the best results, while the other strategies were not able to significantly reduce the number of features per element. We attribute this result to the ability of NSGA-II to explore a more diverse set of solutions. On the other hand, the performance of NSGA-II in this case was worse than that presented for other projects regarding the NFEAT objective. In addition, the best NSGA-II solutions for NFEAT require a considerable amount of refactorings (more than 50), which may be impracticable considering the number of refactored elements (top-10 degraded elements).

Therefore, we conjecture that our strategies that use feature-based heuristics were significantly affected by the quality of the topic model generated for the Jenkins project. In such a case, MOSA was still able to reduce traditional symptoms (i.e., those that are not feature-based).

Solution Space for RxJava. Figure 7.7 presents the solution space for the RxJava project. In this case, we can observe that MOSA also performed better regarding the DENSITY objective.

Interesting enough, this was the only project for which the BASELINE approach presented a larger set of non-dominated solutions. This means that our rule-based refactoring heuristics generated more diverse solutions. However, the graph shows that after a certain number of refactorings the BASELINE solutions are worse than all the others for the DENSITY and NFEAT objectives.

Solution Space for Spring Security and OkHttp. Figures 7.8 and 7.9 present the solution space for the Spring Security and OkHttp projects, respectively. In both cases, we can observe that MOSA, NSGA-II, and SIMPLE solutions are very close when considering the first half of the NREF axis. For such solutions, MOSA performed a bit better regarding the DENSITY objective.

For both Spring Security and OkHttp, NSGA-II tended to improve its impact on DENSITY and NFEAT as the number of refactorings increased. Nevertheless, as we already discussed, a large number of refactorings may make such solutions not feasible in practice.

To summarize, our analysis here revealed that NSGA-II is able to explore a larger search space when compared to the other strategies. Consequently, it provided more opportunities for finding solutions that significantly improve feature modularization.

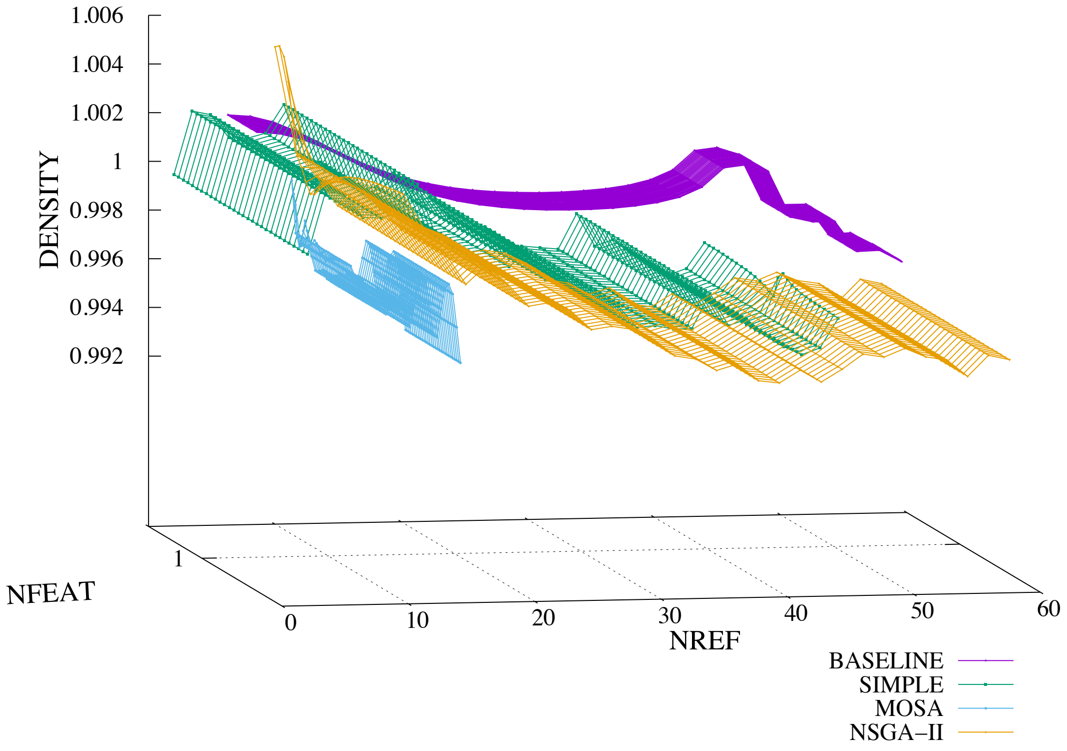


Figure 7.7: Partial view of the solution space for the RxJava project

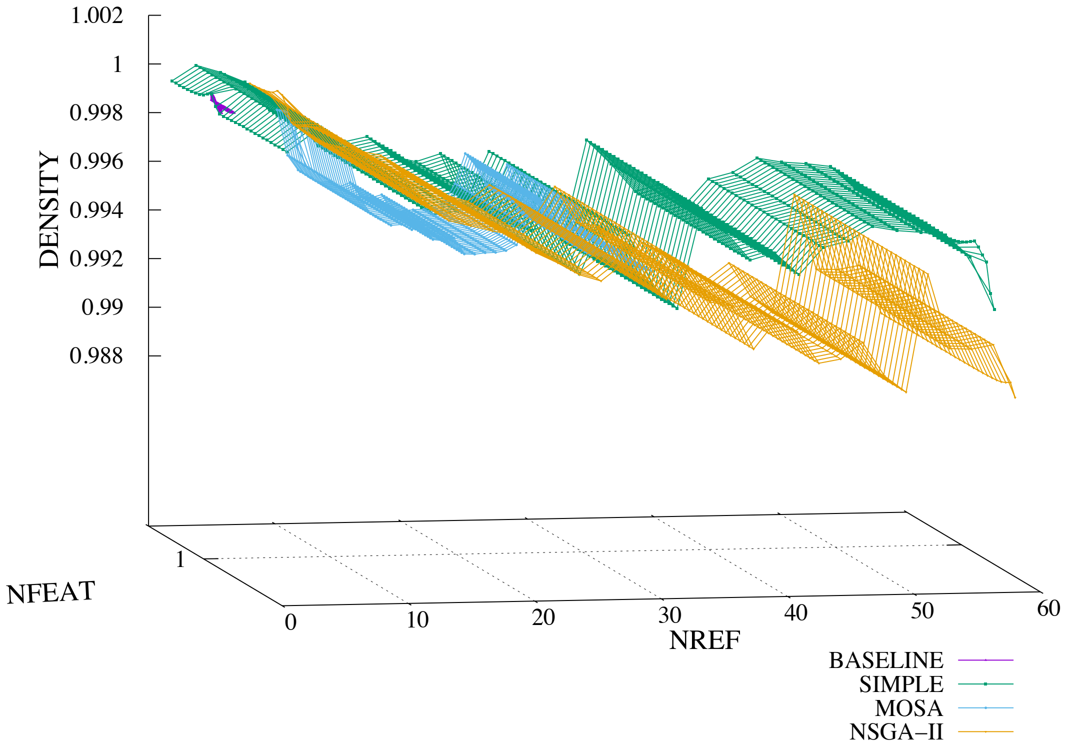


Figure 7.8: Partial view of the solution space for the Spring Security project

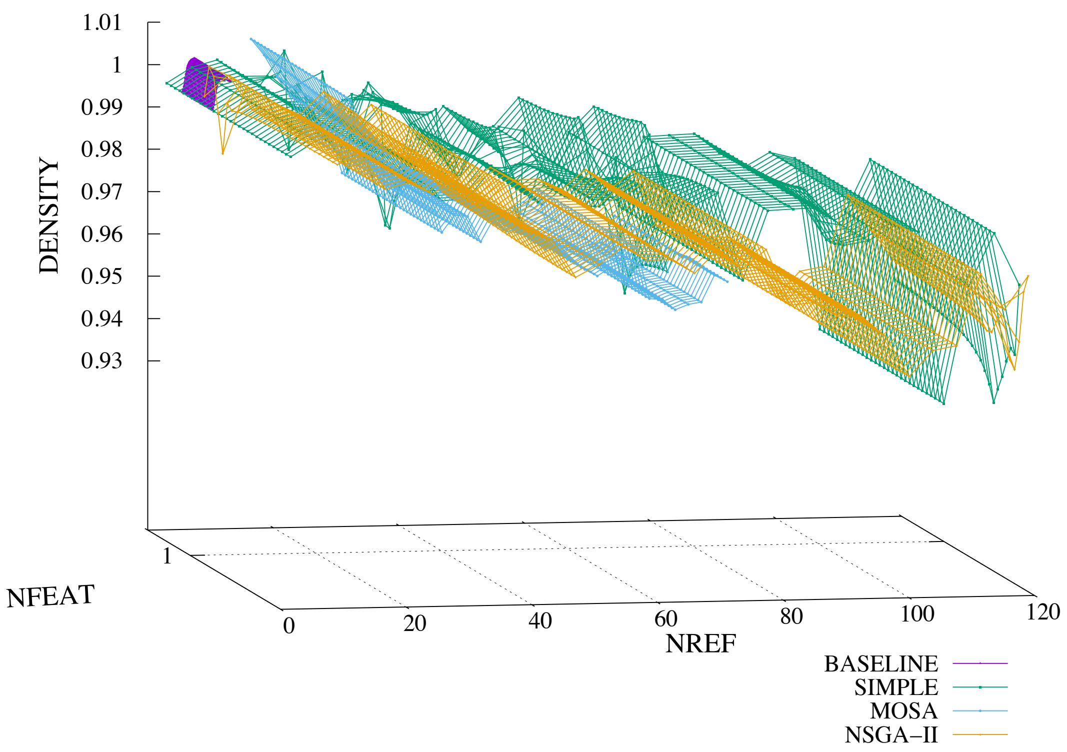


Figure 7.9: Partial view of the solution space for the OkHttp project

The downside of NSGA-II is that, for some projects, the best improvements of DENSITY and NFEAT required a large number of refactorings. On the other hand, when we consider only solutions with smaller NREF values, NSGA-II is able to outperform the other ones either in the improvement of DENSITY or NFEAT.

For some cases (Jenkins, RxJava, and Spring Security) MOSA presented a better impact on the DENSITY objective. It also tended to restrict its solutions to a smaller number of refactorings in all cases. However, the NFEAT results shows that NSGA-II tends to be better than MOSA. Thus, we conclude that NSGA-II is the best strategy for achieving *OrganicRef*'s goal.

Finding 3. NSGA-II better fits the *OrganicRef*'s goal. Due to the global search performed by NSGA-II, it is able to explore a larger search space when compared to the other strategies. As a result, it significantly improved the solutions of most projects.

7.4.3
Best Solutions Analysis

Since NSGA-II and MOSA presented the best results in our previous analysis, we need to better understand the role of optimization algorithms in the performance of *OrganicRef*. Thus, in this section we analyze the best solutions of both MOSA and NSGA-II. As we previously mentioned (Section 7.3.4), the selection of best solutions is based on the *Euclidean Distance* (i.e., distance from the ideal solution).

In Table 7.4 we show the fitness values before (original) and after (refactored) applying the best solution for each project (lines) and strategy (sub-columns). It is possible to observe that, for the same project, some fitness values in the original project vary between MOSA and NSGA-II. This happens because the DENSITY and NFEAT objective functions vary across different executions. Such a variation is caused by the topic modeling algorithm, which is not deterministic.

High impact on the DENSITY and NFEAT objectives. In Section 7.4.2, we explored the solution space of each strategy with a focus in the DENSITY, NFEAT and NREF objectives. In Table 7.4, we show the DENSITY and NFEAT of best solutions in the second and third columns, respectively. For all our target projects, both of them were consistently improved after applying the best solutions. This means that, even without applying any type of weight to the objective functions, the MOSA and NSGA-II strategies had a high impact on DENSITY and NFEAT. This result shows that our refactoring heuristics and our mutation operators were able to act directly on the modularization of features.

Negative impact on the LCOM objective. In Table 7.4, we highlight in red the fitness values that were worsened after applying the refactorings. Therefore, it is possible to observe that fitness values were improved or kept the same in most cases. The most negatively impacted objective function was the LCOM (fourth main column), with five cases with MOSA and three with NSGA-II. This shows that improving the LCOM objective with our heuristics and strategies is challenging. As *OrganicRef* only supports Move Method, Move Field and Extract Class, we believe that LCOM effective improvement may require the inclusion of other types of refactoring such as Extract Method. On the other hand, as we discussed in Section 7.4.1, our recommendations improved the cohesion of elements in the selected context. Another possibility to justify the negative impact on LCOM is a possible negative correlation with the DENSITY or FEAT objectives, which deserves further investigation.

Low to no impact on Quality Measures. In the last main column of Table 7.4, we show the fitness values for the Quality Measures (QMEASURES) objective function, which aggregates multiple quality measures. Such measures are also used for detecting part of the DP symptoms. Therefore, we would expect both the DENSITY and QMEASURES objective functions to be correlated. Nevertheless, as one can see in Table 7.4, in most cases QMEASURES was kept unchanged, while DENSITY was improved for all cases. We conjecture that this lack of correlation is due to the fact that we also use feature-based symptoms. Consequently, our feature-driven heuristics combined with optimization algorithms are able to significantly impact feature-based symptoms, while their impact on traditional symptoms (i.e., those related to the QMEASURES objective) turns out to be less.

Table 7.4: Fitness values before (original) and after (refactored) applying the best solution

Project		**DENSITY**		**NFEAT**		**LCOM**		**QMEASURES**	
		MOSA	**NSGA-II**	**MOSA**	**NSGA-II**	**MOSA**	**NSGA-II**	**MOSA**	**NSGA-II**
Fresco	*Original*	3.445	3.384	1.804	1.773	0.565	0.565	0.057	0.057
	Refactored	0.995	1.000	1.001	1.007	0.567	0.566	0.061	0.056
RxJava	*Original*	3.580	3.121	1.988	2.032	1.304	1.304	0.062	0.062
	Refactored	0.995	0.999	0.998	1.001	1.302	1.300	0.062	0.062
Jenkins	*Original*	2.990	3.073	1.484	1.533	0.485	0.485	0.032	0.032
	Refactored	0.981	0.998	1.000	1.001	0.487	0.485	0.032	0.032
Spring Security	*Original*	3.224	3.224	1.674	1.654	0.551	0.551	0.053	0.053
	Refactored	0.998	0.998	1.000	1.000	0.552	0.552	0.053	0.053
Dubbo	*Original*	3.218	3.297	1.753	1.790	0.540	0.540	0.061	0.061
	Refactored	1.000	0.999	1.002	1.004	0.542	0.545	0.060	0.060
OkHttp	*Original*	3.581	3.581	1.725	1.725	0.676	0.676	0.105	0.105
	Refactored	1.007	0.981	1.013	1.017	0.682	0.668	0.104	0.104

Number of Refactorings. In Table 7.5 we show the number of refactorings for the best solutions generated by the MOSA and NSGA-II strategies. It is possible to see that most of them were similar – between 8 and 10 refactorings. Such a number of refactorings is aligned with the number of elements in our selected context (i.e., top-10 degraded elements). Thus, we consider that the amount of refactorings recommended by the best solutions is adequate for the amount of degraded elements in our selected context.

Table 7.5: Number of refactorings for the best solutions

Project	**Number of Refactorings**	
	MOSA	**NSGA-II**
Fresco	11	8
RxJava	8	10
Jenkins	9	9
Spring Security	8	8
Dubbo	9	10
OkHttp	14	9

Finding 4. Considering only the best solutions, both NSGA-II and MOSA presented closed results for all objective functions. However, considering the impact on the LCOM objective together with the computing

time and diversity of solutions, NSGA-II keeps being the best performing strategy. More importantly, the NSGA-II results show that our refactoring heuristics and our mutation operators can recommend refactoring operations that directly improve the modularization of features and reduce the density of symptoms.

7.4.4 Qualitative Evaluation of Recommendations

As a last step of our evaluation, we recruited four collaborators to analyze the best solutions for the MOSA and NSGA-II strategies in the OkHttp project. Below, we summarize our observations regarding relevant aspects of each strategy. All analyzed solutions are in our replication package.

A recurring observation of the different evaluators is that feature-driven solutions – i.e., generated by MOSA and NSGA-II strategies – contained one or more refactorings that were suitable for the goal of feature modularization. In the analyzed cases, with a few exceptions, the heuristics for Extract Class presented sound and viable recommendations. We also recurringly observed that many of the refactoring candidates were indeed suffering from feature modularization problems. For such cases, our perception is that the proposed refactorings proved capable of (partially) improving feature modularization.

Best solution of MOSA strategy. The best solution generated by MOSA for OkHttp is composed by 14 refactoring operations (6 Extract Classes and 8 Move Methods) applied to 9 different classes. This means that this strategy was able to improve 9 out of 10 classes from the top-10 degraded elements. Below we present a quotation from one of the evaluators:

QT1. *Overall, the refactorings can improve feature modularization. However, in some cases some moved methods or extracted classes are not necessary. Some move methods were unrelated to the target class feature. As far as class extractions are concerned, the tool suggested extracting some methods that should stay in the class as they were related to the class' main feature.*

Given the feedback above, we performed an additional analysis of the code elements mentioned by the author of QT1. For the unnecessary refactorings, we noticed that the effectiveness of MOSA was directly impacted by the topic model quality. In our manual analysis, we observed that a recommendation may move elements to distant and unrelated components. We also found cases in which generic terms (e.g., cancel) induced the technique to move methods to classes implementing totally unrelated features.

For example, one recommendation consisted of moving the *cancel* method from the *RealCall* class to the *RealWebSocket.CancelRunnable* class. The target class is a lazy class with just one method. However, the functionality provided by the moved method has not relation with *RealWebSocket.CancelRunnable*. In fact, *RealCall.cancel()* is concerned with canceling a http request, while *RealWebSocket.CancelRunnable* is aimed at canceling a thread. Therefore, despite the textual similarities between the method and the target class, this recommendation would not make sense in practice.

Thus, a direct solution to the problem presented above is to require some inspection and improvement of the topic model by the developers. Such an improvement should be performed before using the topic model as an input in *OrganicRef*. Moreover, this kind of problem may rarely occur in projects where certain good practices for choosing code elements names are followed.

Next, we present quotations related to the Extract Class refactorings generated with the MOSA strategy:

QT2. *I agree with the extract class of 'okhttp3.internal.platform.Platform'. Multiple methods should be placed in other classes, since they deal with features like logging and protocol communication between layers of security. However the 'findPlatform' method should stay in the class.*

QT3. *I agree with the extract method from 'okhttp3.HttpUrl'. It has multiple methods related to encoding, which should be placed in a different class.*

QT4. *On the 'HttpUrl', extractions make sense because the extracted methods are not directly related to the 'HttpUrl' feature. These methods only helps to manipulate URLs. For example, the methods 'percentDecode', 'get', 'parse', and 'canonicalize' only manipulate Buffer and Strings. However, the methods 'defaultPOrt', 'get', 'url' may be kept on the 'HttpURL' class because they are directly related to the class.*

From quotations QT2 to QT4, we can notice that using MOSA, *OrganicRef* was able to generate sound and viable recommendations which would improve feature modularization in the refactored classes. In the case of QT3 and QT4, different evaluators reached similar conclusions with independent analyses. However, as revealed in quotation QT5 presented below, in some cases the recommended Extract Class also failed to correctly identify the features being realized in the class. Possible workarounds may involve inspections of topic models and certain ID naming practices in the software project as already mentioned above.

QT5. *I disagree with extracting a class from 'okhttp3.Headers'. Most of the methods are related to the building of the headers.*

Best solution of NSGA-II strategy. Now let us move on to the NSGA-II strategy. Its best solution was composed by 9 refactoring operations (8 Extract Classes and 1 Move Method) applied to 9 different classes. In this case, we can get more evidence that our Extract Class heuristics are the best performing ones. The following quotations (QT6 and QT7) provide us with more awareness about that:

QT6. *Extract Class on 'Transmitter' class is recommended because this class implements many responsibilities and the recommended extractions are related to a single responsibility, the Response feature. Thus, it is indicated to extract these fields and methods to another class. Indeed, 'Transmitter' needs many refactorings to improve its feature modularization because its a God Class.*

QT7. *The 'Transmitter' class indeed can be extracted. The methods are related to connections. Thus, they could be better modularized.*

However, we also observe that, similarly to MOSA, NSGA-II suffered from inaccurate feature identification. This fact becomes more evident from QT8 below:

QT8. *Extract Class on class 'Builder' does not make much sense because these attributes and methods are related to building a response. Thus, it is not appropriate to extracting these fields and methods to another class.*

For avoiding the limitations presented above, we envision the improvement of *OrganicRef* with the specification of design constraints. As a result, we expect to avoid creating recommendations that would not be allowed by the projects intended design. Another alternative is to improve the feature-driven heuristics to consider complementary information like locality (the component in which the classes are implemented) and syntactic dependencies for filtering target candidates for moving elements.

This was, indeed, an expected limitation since we ran our evaluation in the worst scenario regarding the topic model quality (see Section 7.3.2). We created the models for all projects using the same generic configuration without customizing parameters or stop-words. Therefore, we conjecture that the recommendations could be improved in case we had better topic models.

Additional Steps Required. There were also some cases for which applying the recommended refactorings would require extra steps, from the developer, besides the ones described in the recommendations. For example, some move methods would require the implementation of additional method calls as mentioned for a recommendation from NSGA-II:

QT9. *Compilation errors may happen because some extracted methods use internal methods (from the source class), then the developer need to updated the extracted class to call these methods.*

Given this observation, we conjecture that one way we can help developers more effectively is through the (semi-)automated application of recommended refactorings. That way, developers do not have to worry about extra, purely manual steps during refactoring application.

Summing up, our observations altogether indicate that the results presented by *OrganicRef* are promising. There are indeed limitations, as discussed above, but these limitations can be addressed by some guidelines for feature model inspections and be better investigated by future studies. The costs of topic model inspections are unlikely to be prohibitive as performing coarse grained refactorings without such a support is certainly much more costly. Moreover, previous work revealed that many mistakes of feature mapping in the source code can be automatically repaired (Nunes *et al.* 2014). As new evaluations and improvements are carried out, we believe that *OrganicRef* can evolve to the point of being useful in practice.

Finding 5. The quality of solutions generated by *OrganicRef* depends directly on adequately configuring and training a topic model for feature identification. However, even in the worst case scenario, *OrganicRef* was able to generate sound and viable recommendations. On the other hand, we need to improve our Move Method and Move Field heuristics since they presented some undesirable results.

7.4.5 Threats to Validity

The settings of the detection algorithms/heuristics are a common threat to the validity for empirical studies evaluating recommender techniques. To mitigate this threat, we initially followed the same configurations applied by state-of-the-art studies. Then, we followed a trial and error process for tuning each parameter.

Our technique relies on the automated detection of DP symptoms, which may result in false negatives and false positives. Therefore, the quantitative analysis of the recommendations generated by the tool is subject to this threat. We mitigated it by conducting a later manual and individual validation over a reduced subset of the generated recommendations.

As in the case of DPs, the rules and thresholds employed for detecting code smells can be also considered threats to validity. We opted to not perform additional validations over the detected smells once *OrganicRef* relies on automatically detected symptoms of code smells. Thus, the manual validation of these symptoms does not directly address our research goal. Alternatively, we employed consolidated rules and thresholds employed in previous work.

We curated a set composed by six open source projects to analyze, which may raise questions on the representativeness of our study sample and the replicability of the study findings. We mitigate this threat by establishing a systematic process for selecting active and relevant projects written in Java. As a result, we depicted a small but diverse sample of projects varying in architecture, size, and domain.

We compared *OrganicRef* with a baseline defined and implemented by ourselves. We are aware that, ideally, we should compare our technique to state-of-the-art techniques. However, we are not aware of any similar already existing technology with the same goal fully available for comparison. Differently from *OrganicRef*, most of the refactoring recommendation techniques are not focused exclusively on feature modularization problems. To mitigate this threat, we created a baseline inspired by existing recommendation techniques. In addition, our implementation and data are fully available in the replication package. Therefore, other researchers can verify, reproduce and replicate this study.

Finally, another threat is the selection of the search-based algorithms and their parameter settings. We used MOSA that is employed in a closely related work (Kessentini, Dea and Ouni 2017) and NSGA-II that is one of the best genetic algorithms to deal with more problems impacted by multiple objectives (Colanzi *et al.* 2020). We adopted canonical parameter settings and parameters used in related work. Due to the multi-objective nature of NSGA-II, the solutions may converge to a different set of local optimum (*i.e.*, refactoring recommendations) in each run, without finding the global optimum. To mitigate any bias, we set the execution of the algorithm to 30 independent runs, what is recommended for optimization studies (Arcuri and Briand 2014, Colanzi *et al.* 2020).

7.5 Concluding Remarks

In this chapter we proposed and evaluated the *OrganicRef* technique. *OrganicRef* is intended to provide effective and context-sensitive refactoring recommendations for feature modularity. Despite the vast literature about refactoring recommendation, our proposed technique has a number of characteristics that are not found in state-of-the-art techniques.

First, to the best of our knowledge, *OrganicRef* is the first technique that provides feature-driven refactoring recommendations, which do not require the complete redesign of the refactored project. For enabling our evaluation, in this study we focused in creating recommendations for the top ten degraded elements. Nevertheless, *OrganicRef* allows the use of different heuristics for restricting DP symptoms and refactoring recommendations to the developer's context of interest.

Besides having novel characteristics, *OrganicRef* also incorporates characteristics that showed to be effective in other studies. For example, we designed *OrganicRef* to provide optimized refactoring recommendations based on the use of search-based algorithms. Such a characteristic is important for exploring the vast search space of the refactoring problem.

Our evaluation helped us to validate the design decisions behind *OrganicRef*. Moreover, the results show that optimized recommendations have the advantage of better improving the context's elements without negatively impacting other elements.

As future work, we envision the improvement of the feature detection component through the use of information extracted from other project's artifacts (e.g., issue tracking system). We also intend to include heuristics that use more refactoring types, such as Move Class and Extract Method. Finally, we intend to conduct more robust studies to evaluate and improve our technique.

8 Conclusion

The early identification and refactoring of DPs is fundamental to maintain the structural quality of complex and long-lived systems. Therefore, there is a growing need for better techniques and tools to help developers and reviewers to tackle the incidence of design problems. In this work, we contributed to this matter through the proposal and evaluation of a technique that provides refactoring recommendations for removing DPs. In the next sections, we revisit this thesis contributions (Section 8.1), summarize our publications and collaborations (Section 8.2), and point out directions for future studies (Section 8.3).

8.1 Revisiting our Contributions

To make our contributions clear, in this section we revisit and summarize the main contributions of each chapter. As we describe below, each of our studies provided significant contributions towards our main goal of *providing effective support for developers in the identification and refactoring of design problems.*

Effective identification of design problems. In Chapter 3 we started our contributions with a deep investigation on how to provide effective support for DP identification. For performing this investigation, we conducted two evaluations. The first one was a quasi-experiment involving professional software developers. This evaluation revealed that developers report much less false positives when reasoning about multiple and correlated DP symptoms. Nevertheless, we also found several aspects that should be improved in order to help developers in using the information provided by the DP symptoms. Besides that, we also conducted a qualitative evaluation focused on human-computer interaction. For this evaluation, we relied on the Communicability Evaluation Method (De Souza and Leitão 2009). As a result, we observed that the effectiveness of DP identification is directly impacted by the way symptoms are presented in a tool.

Evaluation of strategies for ranking and filtering refactoring candidates. In Chapter 4, we continued our investigations towards to goal of providing effective support for DP identification. In fact, in the previous chapter we observed that even using combinations of symptoms, developers still need support to focus their analyzes in specific contexts. Thus, we conducted a collaborative study in which we proposed and evaluated a set of filtering and prioritization criteria for refactoring candidates. Our results showed that no criteria is consistently effective across different software projects. However, our evaluations also showed that the *Flood Criterion* – which consists of combining multiple correlated symptoms – is the best criterion among the evaluated ones. Therefore, this criterion can be a default choice when developers are not entirely sure about their context of interest.

Relation of refactorings with DPs and their symptoms. In Chapter 5 we investigated the relation of symptoms, DPs, and refactorings through a multi-case study involving C# and Java projects. In summary, such an investigation showed that, in practice, refactored classes tend to present higher density and diversity of symptoms. Such a result provided additional evidence towards the theory that DPs should be identified through multiple and diverse symptoms (Sousa *et al.* 2018).

Impact of refactorings in practice. Nevertheless, our results also revealed that refactorings performed in practice produce little to no impact on DP symptoms. As a matter of fact, such a result is corroborated by multiple studies, which indicate that performing effective refactorings is challenging (Cedrim *et al.* 2017), (Bibiano *et al.* 2019), (Bibiano *et al.* 2020).

Initial requirements for refactoring recommendation. Given the limitations of refactorings performed in practice, as a final contribution of Chapter 5, we identified an initial set of requirements for refactoring recommendation techniques. Among the requirements, we highlight the following. First, for increasing its effectiveness, besides considering the density of symptoms, a technique should also rely on the use of diverse symptom types for finding refactoring candidates. Second, each project and developer may have different contexts of interest for refactoring. For instance, to find highly degraded elements in the project, a developer may rely on the agglomeration flood criterion proposed in Chapter 4. Nevertheless, there may be other contexts of interest such as the elements involved in a current or future task. Thus, a recommendation technique should provide a flexible approach for filtering and prioritizing elements based on the developer's context of interest.

A rule-based refactoring recommendation technique. In Chapter 6 we proposed and evaluated a refactoring recommendation technique. Such a technique is focused in creating recommendations for the removal of three DP types: Feature Overload, Scattered Feature, and Complex Component. For achieving such an objective, our proposed technique relies on automatically detected code smells and on rule-based heuristics. Such heuristics were developed based on a previous study, in which we investigated recurrent refactoring patters that occur in open source projects. Our evaluation showed that our heuristics can be successful in multiple scenarios. However, it revealed several aspects that should be improved for generating effective recommendations.

A feature-driven and context-sensitive recommendation technique. In Chapter 7 we proposed and evaluated the *OrganicRef* technique. Such a technique was designed based on state-of-the-art evidence, which includes the results presented in our previous studies. More specifically, *OrganicRef* is designed to address four key requirements for refactoring recommendation techniques, which are (1) consideration of heterogeneous information, (2) context-sensitive detection, (3) feature awareness, and (4) effective recommendations.

Based on aforementioned requirements, *OrganicRef* is intended to help developers in spotting and refactoring feature-related DPs in delimited contexts. The DPs are detected through information extracted from the project's design and source code. *OrganicRef* uses a topic modeling algorithm for finding existing features in the project implementation. Then, *OrganicRef* combines features information with internal quality measures and code smells to find DPs. For creating refactoring recommendations, *OrganicRef* relies on a new refactoring recommendation strategy, which combines refactoring heuristics with search-based optimization.

OrganicRef Evaluation. We evaluated *OrganicRef* with an empirical study involving open-source projects. Our results show that, when compared to a baseline, *OrganicRef* significantly improves the design quality of delimited contexts through effective refactoring recommendations.

Open Science Contributions. Besides all the aforementioned results and contributions, in this thesis we also made an effort for following the modern open science standards. All of our empirical studies include replication packages containing the study's artifacts.

Besides that, we also made available all the tools developed and improved

Table 8.1: Direct contributions of this thesis

Publication	Qualis
Oizumi et al.: OrganicRef: Towards Effective and Context-Sensitive Refactoring of Features. ICSME 2022 (submitted).	A2
Oizumi et al.: On the identification of design problems in stinky code: Experiences and tool support. JBCS 2018.	A2
Oizumi et al.: Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation. SBES 2020 [**Distinguished Paper Award**].	A3
Oizumi et al.: On the density and diversity of degradation symptoms in refactored classes A multi-case study. ISSRE 2019.	A3
Eposhi, **Oizumi**, et al.: Removal of design problems through refactorings Are we looking at the right symptoms? ICPC Negative Results Track 2019.	A3
Vidal, **Oizumi**, et al.: Ranking architecturally critical agglomerations of code smells. Science of Computer Programming 2019.	A4
Oizumi, W. N. Recommendation of Refactorings for Improving Dependability Attributes. ISSREW 2019.	B1

during this PhD research. As presented in Chapter 3, we improved and evaluated the *Organic* tool, which was initially developed during my master's research (Oizumi *et al.* 2016, Oizumi *et al.* 2015). *Organic* implementations is fully available [1] as an open source tool and has been already used by other researchers.

Finally, as a last contribution, we created a new reference tool for the *OrganicRef* technique. We consider that making *OrganicRef*'s source code available is a considerable contribution for the search-based refactoring community. Despite the existence of multiple studies and techniques in the search-based refactoring literature, the vast majority are based on closed source tools. Therefore, we consider that our tool can contribute significantly to the refactoring researchers. In addition to being a viable baseline for future studies, *OrganicRef*'s code can also be extended and improved for addressing new requirements.

8.2 Publications and Collaborations

This PhD research resulted in seven papers (six publications and one submission) at diverse conferences and journals. Such papers are listed in Table 8.1. Table 8.2 presents the list of papers resulting from collaborations that are indirectly related to this thesis. As one can see in both tables, this research resulted in high quality papers (mostly Qualis A1-A4).

As evidenced by our publications (Table 8.2), we collaborated with multiple researchers on studies related to topics such as DPs, code smells, refactoring, code review, search-based software engineering, and software architecture. Besides collaborating with colleagues from the Opus Research Group, we

[1] `https://opus-research.github.io/tools.html`

also worked with researchers from: *Carnegie Mellon University*, *UNICEN University - Argentina*, *University of Maringá (UEM)*, and *State University of Goiás (UEG)*.

Such collaborations proved to be fundamental for the development of this thesis. As we showed in Chapter 4, our long-term collaboration with UNICEN's researchers has allowed us to deeply investigate different criteria for prioritizing and filtering candidates for refactoring. Such an investigation help us to better understand the requirements for refactoring recommendation techniques. Moreover, we also collaborated with our colleagues from the Opus Research Group in a large scale mining software repositories study. As a result, we were able to propose and evaluate a refactoring recommendation technique (Chapter 6).

In summary, as we presented in this section, throughout this thesis we were able to carry out several studies and collaborations that resulted in multiple publications. Furthermore, as discussed in the previous section, the *OrganicRef* technique opens up several possibilities for future studies. Therefore, in the following section we discuss some of these possibilities.

8.3 Future Work

Empirical studies with *OrganicRef*. In Chapter 7, we proposed and evaluated the *OrganicRef* technique. Although we consider our evaluation robust, there is still a need for further studies involving *OrganicRef*. For instance, we conducted a manual evaluation of recommendations generated by *OrganicRef*. Nevertheless, there is a need for an evaluation involving professional software developers. Moreover, it would be interesting to evaluate OrganicRef in the context of software companies and their closed-source projects. Finally, we explored the use of two search-based algorithms, namely MOSA and NSGA-II. However, we may want to further explore the use of other algorithms, such as NSGA-III.

Support for removing more DP types. Although our studies involved the support for identifying and removing multiple DP types, *OrganicRef* was initially designed to focus on the *Feature Overload* and *Scattered Feature* problems. Therefore, future studies may extend *OrganicRef* to support the identification and removal of additional DP types. This may require the detection of more DP symptoms and also the inclusion of more refactoring types. Examples of such refactorings types include, Extract Method and Move

Table 8.2: Publications indirectly related to this thesis

Publication	**Qualis**
Sousa et al.: Identifying Design Problems in the Source Code: A Grounded Theory. ICSE 2018: 921-931 [**IEEE/ACM Distinguished Paper Award**].	A1
Uchôa et al.: Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study. MSR 2021.	A1
Sousa et al.: Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. MSR 2020.	A1
Fernandes et al.: Refactoring effect on internal quality attributes: What haven't they told you yet?. IST 2020.	A1
Uchôa et al.: How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. ICSME 2020.	A2
Oliveira et al.: Evaluating Smell Patterns for Refactoring Opportunities. SBES 2022 (under submission).	A3
Sousa et al.: When Are Smells Key Symptoms of Refactoring Opportunities? A Study of 50 Software Projects. ICPC 2020 [**Invited to a Special Issue of EMSE**].	A3
Oliveira et al.: Applying Machine Learning to Customized Smell Detection: A Multi-Project Study. SBES 2020.	A3
Mello et al.: Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis. ESEM 2019.	A3
Mello et al.: Investigating the Social Representations of the Identification of Code Smells by Practitioners and Students from Brazil. SBES 2019.	A3
Sousa et al.: How Do Software Developers Identify Design Problems? A Qualitative Analysis. SBES 2017: 54-63.	A3
Madrigar et al.: OPLA-Tool-ASP: A Tool to Prevent Architectural Smells in Search-based Product Line Architecture Design. JSERD 2021.	B1
Oliveira et al.: On the Prioritization of Design-Relevant Smelly Elements A Mixed-Method, Multi-Project Study. SBCARS 2019 [**Best Paper Award**].	B1
Perissato et al.: On Identifying Architectural Smells in Search-based Product Line Designs. SBCARS 2018.	B1
Oizumi et al.: Revealing Design Problems in Stinky Code A Mixed-method Study. SBCARS 2017 [**2nd Best Paper Award**].	B1
Madrigar et al.: Prevenção de Anomalias Arquiteturais na Otimização de Projeto de Linha de Produto de Software. CIbSE 2020 [**Best Paper Candidate**].	B2

Class.

Improvements in feature identification. As we observed in Chapter 7, the topic model quality directly impacts the effectiveness of *OrganicRef*. Therefore, we envision future studies focused in improving the detection of features. For performing such improvements, we can continue relying on topic modeling or we may explore other approaches for feature identification. For example, our topic modeling strategy may be enhanced through the use of textual information extracted from other software artifacts, such as the issue tracking system. We may also adapt existing approaches, such as the use of automated tests for features identification (Carvalho *et al.* 2020).

Intended design awareness. In our early literature investigations, we observed that little to no technique considers the intended design of a project for recommending refactorings. The intended design of a software project determines which structure the implementation should follow (Bass *et al.* 2003, Gurp and Bosch 2002, Terra *et al.* 2012, Hickey and Cinnéide 2015). It is important to guide and help developers in preserving the quality attributes (Bass *et al.*

2003, Gurgel *et al.* 2014, Barbosa and Garcia 2017, Terra *et al.* 2012, Hickey and Cinnéide 2015). Not taking the intended design into account may result in recommendations that violate it. However, such a need did not emerge as a requirement during our empirical studies. Consequently, we decided to not include it in the first version of *OrganicRef*. Nevertheless, as we observed in our qualitative analysis in Chapter 7, design-awareness could indeed be useful for improving the quality of refactoring recommendations. Thus, we let the inclusion and investigation of such an relevant requirement as a future work.

Automatic application of refactoring recommendations. Finally, despite automatically generating refactoring recommendations, *OrganicRef* is still unable to automatically apply the recommendations in the source code. Such an ability is desired to facilitate the adoption of *OrganicRef* in practice. However, this is not a trivial task. Therefore, future studies may focus on the development of efficient approaches for the automated application of the recommendations generated by *OrganicRef*.

Bibliography references

[Abbes *et al.* 2011] ABBES, M.; KHOMH, F.; GUEHENEUC, Y. ; ANTONIOL, G.. **An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension**. In: PROCEEDINGS OF THE 15TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE; OLDENBURG, GERMANY, p. 181–190, 2011.

[Abid *et al.* 2020] ABID, C.; ALIZADEH, V.; KESSENTINI, M.; FERREIRA, T. D. N. ; DIG, D.. **30 years of software refactoring research: a systematic literature review**. arXiv preprint arXiv:2007.02194, 2020.

[Alenezi and Zarour 2018] ALENEZI, M.; ZAROUR, M.. **An empirical study of bad smells during software evolution using designite tool**. i-Manager's Journal on Software Engineering, 12(4):12–27, Apr 2018.

[Alizadeh and Kessentini 2018] ALIZADEH, V.; KESSENTINI, M.. **Reducing interactive refactoring effort via clustering-based multi-objective search**. In: PROCEEDINGS OF THE 33RD ASE, p. 464–474, New York, NY, USA, 2018. ACM.

[Alizadeh *et al.* 2019] ALIZADEH, V.; KESSENTINI, M.; MKAOUER, W.; OCINNEIDE, M.; OUNI, A. ; CAI, Y.. **An interactive and dynamic search-based approach to software refactoring recommendations**. IEEE Transactions on Software Engineering, p. 1–1, 2018.

[Alizadeh *et al.* 2019b] ALIZADEH, V.; OUALI, M. A.; KESSENTINI, M. ; CHATER, M.. **Refbot: intelligent software refactoring bot**. In: 2019 34TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 823–834. IEEE, 2019.

[Alizadeh *et al.* 2019c] ALIZADEH, V.; FEHRI, H. ; KESSENTINI, M.. **Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction**. In: 2019 19TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 181–192, 2019.

[Arcoverde *et al.* 2013] ARCOVERDE, R.; GUIMARÃES, E.; MACÍA, I.; GARCIA, A. ; CAI, Y.. **Prioritization of code anomalies based on architecture sensitiveness**. In: 2013 27TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 69–78, Oct 2013.

[Arcuri and Briand 2014] ARCURI, A.; BRIAND, L.. **A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering**. Software Testing, Verification and Reliability, 24(3):219–250, 2014.

[Azadi, Fontana and Taibi 2019] AZADI, U.; ARCELLI FONTANA, F. ; TAIBI, D.. **Architectural smells detected by tools: a catalogue proposal**. In: INTERNATIONAL CONFERENCE ON TECHNICAL DEBT (TECHDEBT 2019), 2019.

[Barbosa and Garcia 2017] BARBOSA, E. A.; GARCIA, A.. **Global-aware recommendations for repairing violations in exception handling**. IEEE Transactions on Software Engineering, 44(9):855–873, Sept 2018.

[Bass *et al.* 2003] BASS, L.; CLEMENTS, P. ; KAZMAN, R.. **Software Architecture in Practice**. Addison-Wesley Professional, 2003.

[Bavota *et al.* 2013] BAVOTA, G.; OLIVETO, R.; GETHERS, M.; POSHYVANYK, D. ; DE LUCIA, A.. **Methodbook: Recommending move method refactorings via relational topic models**. IEEE Transactions on Software Engineering, 40(7):671–694, 2013.

[Bavota *et al.* 2014] BAVOTA, G.; DE LUCIA, A.; MARCUS, A. ; OLIVETO, R.. **Recommending Refactoring Operations in Large Software Systems**, p. 387–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Bavota *et al.* 2014] BAVOTA, G.; GETHERS, M.; OLIVETO, R.; POSHYVANYK, D. ; LUCIA, A. D.. **Improving software modularization via automated analysis of latent topics and dependencies**. ACM Trans. Softw. Eng. Methodol., 2014.

[Bavota *et al.* 2015] BAVOTA, G.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring**. Journal of Systems and Software, 107:1–14, 2015.

[Besker, Martini and Bosch 2017] BESKER, T.; MARTINI, A. ; BOSCH, J.. **Time to pay up: Technical debt from a software quality per-**

spective. In: PROCEEDINGS OF THE XX IBEROAMERICAN CONFERENCE ON SOFTWARE ENGINEERING, BUENOS AIRES, ARGENTINA, MAY 22-23, 2017., p. 235–248, 2017.

[Bibiano *et al.* 2019] BIBIANO, A. C.; FERNANDES, E.; OLIVEIRA, D.; GARCIA, A.; KALINOWSKI, M.; FONSECA, B.; OLIVEIRA, R.; OLIVEIRA, A. ; CEDRIM, D.. **A quantitative study on characteristics and effect of batch refactoring on code smells**. In: 13TH ESEM, p. 1–11, 2019.

[Bibiano *et al.* 2020] BIBIANO, A. C.; SOARES, V.; COUTINHO, D.; FERNANDES, E.; CORREIA, J. A. L.; SANTOS, K.; OLIVEIRA, A.; GARCIA, A.; GHEYI, R.; FONSECA, B.; RIBEIRO, M.; BARBOSA, C. ; OLIVEIRA, D.. **How does incomplete composite refactoring affect internal quality attributes?** In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC '20, p. 149–159, New York, NY, USA, 2020. Association for Computing Machinery.

[Booch 2004] BOOCH, G.. **Object-Oriented Analysis and Design with Applications (3rd Edition)**. Addison Wesley, Redwood City, CA, USA, 2004.

[Brito, Hora and Valente 2020] BRITO, A.; HORA, A. ; VALENTE, M. T.. **Refactoring graphs: Assessing refactoring over time**. In: 2020 IEEE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 367–377, 2020.

[Brown *et al.* 1998] BROWN, W. J.; MALVEAU, R. C.; MCCORMICK III, H. W. ; MOWBRAY, T. J.. **Refactoring software, architectures, and projects in crisis**, 1998.

[Campbell and Papapetrou 2013] CAMPBELL, G.; PAPAPETROU, P. P.. **SonarQube in action**. Manning Publications Co., 2013.

[Carvalho *et al.* 2020] CARVALHO, L.; GARCIA, A.; COLANZI, T.; ASSUNÇÃO, W.; PEREIRA, J.; FONSECA, B.; RIBEIRO, M.; LIMA, M. ; LUCENA, C.. **On the performance and adoption of search-based microservice identification with tomicroservices**. In: 36TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME) [TO APPEAR], Sep 2020.

[Cedrim 2018] CEDRIM, D.. **Understanding and improving batch refactoring in software systems**. PhD thesis, Ph. D. dissertation, Informatics Department, PUC-Rio, Brazil, 2018.

[Cedrim *et al.* 2016] CEDRIM, D.; SOUSA, L.; GARCIA, A. ; GHEYI, R.. **Does refactoring improve software structural quality? a longitudinal study of 25 projects**. In: PROCEEDINGS OF THE 30TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 73–82. ACM, 2016.

[Cedrim *et al.* 2017] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects**. In: PROCEEDINGS OF THE 2017 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2017, p. 465–475, New York, NY, USA, 2017. ACM.

[Charalampidou *et al.* 2017] CHARALAMPIDOU, S.; AMPATZOGLOU, A.; CHATZIGEORGIOU, A.; GKORTZIS, A. ; AVGERIOU, P.. **Identifying extract method refactoring opportunities based on functional relevance**. IEEE Transactions on Software Engineering, 2017.

[Chatzigeorgiou and Manakos 2014] CHATZIGEORGIOU, A.; MANAKOS, A.. **Investigating the evolution of code smells in object-oriented systems**. Innovations in Systems and Software Engineering, 10(1):3–18, 2014.

[Chávez *et al.* 2017] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes?: A multi-project study**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES'17, p. 74–83, New York, NY, USA, 2017. ACM.

[Ciupke 1999] CIUPKE, O.. **Automatic detection of design problems in object-oriented reengineering**. In: PROCEEDINGS OF TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS - TOOLS 30 (CAT. NO.PR00278), p. 18–32, Aug 1999.

[Coello *et al.* 2007] COELLO, C. A. C.; LAMONT, G. B.; VAN VELDHUIZEN, D. A. ; OTHERS. **Evolutionary algorithms for solving multi-objective problems**, volumen 5. Springer, 2007.

[Colanzi *et al.* 2014] COLANZI, T. E.; VERGILIO, S. R.; GIMENES, I. M. S. ; OIZUMI, W. N.. **A search-based approach for software product line design**. In: PROCEEDINGS OF THE 18TH INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE - VOLUME 1, SPLC '14, p. 237–241, New York, NY, USA, 2014. Association for Computing Machinery.

[Colanzi *et al.* 2020] COLANZI, T. E.; ASSUNÇÃO, W. K. G.; VERGILIO, S. R.; FARAH, P. R. ; GUIZZO, G.. **The Symposium on Search-Based Software Engineering: Past, Present and Future**. Information and Software Technology, 127:106372, 2020.

[Curtis, Sappid and Szynkarski 2012] CURTIS, B.; SAPPIDI, J. ; SZYNKARSKI, A.. **Estimating the size, cost, and types of technical debt**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL WORKSHOP ON MANAGING TECHNICAL DEBT, MTD '12, p. 49–53, Piscataway, NJ, USA, 2012. IEEE Press.

[De Souza and Leitão 2009] DE SOUZA, C. S.; LEITÃO, C. F.. **Semiotic engineering methods for scientific research in hci**. Synthesis Lectures on Human-Centered Informatics, 2(1):1–122, 2009.

[Deb and Jain 2014] DEB, K.; JAIN, H.. **An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints**. IEEE Transactions on Evolutionary Computation, 18(4):577–601, 2014.

[Deb *et al.* 2002] DEB, K.; PRATAP, A.; AGARWAL, S. ; MEYARIVAN, T.. **A fast and elitist multiobjective genetic algorithm: Nsga-ii**. IEEE Trans. Evolutionary Comp., 6(2):182–197, 2002.

[Easterbrook *et al.* 2008] EASTERBROOK, S.; SINGER, J.; STOREY, M.-A. ; DAMIAN, D.. **Selecting Empirical Methods for Software Engineering Research**. Springer London, London, 2008.

[Emden and Moonen 2002] EMDEN, E.; MOONEN, L.. **Java quality assurance by detecting code smells**. In: PROCEEDINGS OF THE 9TH WORKING CONFERENCE ON REVERSE ENGINEERING; RICHMOND, USA, p. 97, 2002.

[Eposhi *et al.* 2019] EPOSHI, A.; OIZUMI, W.; GARCIA, A.; SOUSA, L.; OLIVEIRA, R. ; OLIVEIRA, A.. **Removal of design problems through refactorings: Are we looking at the right symptoms?** In: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC '19, p. 148–153, Piscataway, NJ, USA, 2019. IEEE Press.

[Ernst *et al.* 2015] ERNST, N. A.; BELLOMO, S.; OZKAYA, I.; NORD, R. L. ; GORTON, I.. **Measure it? manage it? ignore it? software practi-**

tioners and technical debt. In: PROCEEDINGS OF THE 2015 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2015, p. 50–60, New York, NY, USA, 2015. ACM.

[Fokaefs *et al.* 2011] FOKAEFS, M.; TSANTALIS, N.; STROULIA, E. ; CHATZIGEORGIOU, A.. **Jdeodorant: identification and application of extract class refactorings**. In: 2011 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1037–1039, May 2011.

[Fowler 1999] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, Boston, 1999.

[Fraire *et al.* 2020] FRAIRE HUACUJA, H. J.; SOTO, C.; DORRONSORO, B.; SANTILLÁN, C. G.; VALDEZ, N. R. ; BALDERAS-JARAMILLO, F.. **AMOSA with Analytical Tuning Parameters and Fuzzy Logic Controller for Heterogeneous Computing Scheduling Problem**, p. 195–208. Springer International Publishing, Cham, 2020.

[Freeman and David 2004] FREEMAN, P.; DAVID, H.. **A science of design for software-intensive systems**. Communications of the ACM, 47(8):19–21, 2004.

[Garcia *et al.* 2009] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Identifying architectural bad smells**. In: CSMR09; KAISERSLAUTERN, GERMANY. IEEE, 2009.

[Garcia *et al.* 2009b] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Toward a catalogue of architectural bad smells**. In: Mirandola, R.; Gorton, I. ; Hofmeister, C., editors, ARCHITECTURES FOR ADAPTIVE SOFTWARE SYSTEMS, p. 146–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Garcia *et al.* 2013] GARCIA, J.; IVKOVIC, I. ; MEDVIDOVIC, N.. **A comparative analysis of software architecture recovery techniques**. In: PROCEEDINGS OF THE 28TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING; PALO ALTO, USA, 2013.

[Girba, Ducasse and Lanza 2004] GÎRBA, T.; DUCASSE, S. ; LANZA, M.. **Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes**. In: 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2004. PROCEEDINGS., 2004.

[Godfrey and Lee 2000] GODFREY, M.; LEE, E.. **Secrets from the monster: Extracting Mozilla's software architecture**. In: COSET-00; LIMERICK, IRELAND, p. 15–23, 2000.

[Goldberg 1989] GOLDBERG, D. E.. **Genetic Algorithms in Search, Optimization and Machine Learning**. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[Guimaraes, Garcia and Cai 2014] GUIMARAES, E.; GARCIA, A. ; CAI, Y.. **Exploring blueprints on the prioritization of architecturally relevant code anomalies**. In: 2014 IEEE 38TH ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2014.

[Guimaraes *et al.* 2018] GUIMARAES, E.; VIDAL, S.; GARCIA, A.; DIAZ PACE, J. ; MARCOS, C.. **Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support**. Software: Practice and Experience, 48(5):1077–1106, 2018.

[Gurgel *et al.* 2014] GURGEL, A.; MACIA, I.; GARCIA, A.; VON STAA, A.; MEZINI, M.; EICHBERG, M. ; MITSCHKE, R.. **Blending and reusing rules for architectural degradation prevention**. In: PROCEEDINGS OF THE 13TH INTERNATIONAL CONFERENCE ON MODULARITY, MODULARITY '14, p. 61–72, New York, NY, USA, 2014. Association for Computing Machinery.

[Gurp and Bosch 2002] VAN GURP, J.; BOSCH, J.. **Design erosion: problems and causes**. Journal of Systems and Software, 61(2):105 – 119, 2002.

[Harman and Jones 2001] HARMAN, M.; JONES, B. F.. **Search-based software engineering**. Information and Software Technology, 43(14):833 – 839, 2001.

[Harman and Tratt 2007] HARMAN, M.; TRATT, L.. **Pareto optimal search based refactoring at the design level**. In: PROCEEDINGS OF THE 9TH ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, GECCO '07, p. 1106–1113, New York, NY, USA, 2007. Association for Computing Machinery.

[Harman *et al.* 2012] HARMAN, M.; MCMINN, P.; DE SOUZA, J. T. ; YOO, S.. **Search Based Software Engineering: Techniques, Taxonomy, Tutorial**, p. 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Herman, Melancon and Marshall 2000] HERMAN, I.; MELANCON, G. ; MARSHALL, M. S.. **Graph visualization and navigation in information visualization: A survey**. IEEE Transactions on Visualization and Computer Graphics, 6(1):24–43, Jan 2000.

[Hickey and Cinnéide 2015] HICKEY, S.; CINNÉIDE, M. O.. **Search-based refactoring for layered architecture repair: An initial investigation**. In: PROC. 1ST NORTH AMERICAN SEARCH BASED SOFTWARE ENGINEERING SYMPOSIUM, 2015.

[Hozano *et al.* 2017] HOZANO, M.; GARCIA, A.; ANTUNES, N.; FONSECA, B. ; COSTA, E.. **Smells are sensitive to developers! on the efficiency of (un)guided customized detection**. In: 2017 IEEE/ACM 25TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 110–120, May 2017.

[ISO-IEC 25010 2011] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models**. ISO, 2011.

[Kazman *et al.* 2015] KAZMAN, R.; CAI, Y.; MO, R.; FENG, Q.; XIAO, L.; HAZIYEV, S.; FEDAK, V. ; SHAPOCHKA, A.. **A case study in locating the architectural roots of technical debt**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 2, ICSE '15, p. 179–188, Piscataway, NJ, USA, 2015. IEEE Press.

[Kessentini, Dea and Ouni 2017] KESSENTINI, M.; DEA, T. J. ; OUNI, A.. **A context-based refactoring recommendation approach using simulated annealing: Two industrial case studies**. In: PROCEEDINGS OF THE GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE, GECCO '17, p. 1303–1310, New York, NY, USA, 2017. Association for Computing Machinery.

[Kim, Zimmermann and Nagappan 2014] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An Empirical Study of Refactoring Challenges and Benefits at Microsoft**. IEEE Transactions on Software Engineering, 40(7):633–649, 2014.

[Kim *et al.* 2013] KIM, D.; TAO, Y.; KIM, S. ; ZELLER, A.. **Where should we fix this bug? a two-phase recommendation model**. IEEE Transactions on Software Engineering, 39(11):1597–1610, Nov 2013.

[Kumar and Kumar 2011] KUMAR, M. R.; KUMAR, R. H.. **Architectural refactoring of a mission critical integration application: A case study**. In: PROCEEDINGS OF THE 4TH INDIA SOFTWARE ENGINEERING CONFERENCE, ISEC '11, p. 77–83, New York, NY, USA, 2011. ACM.

[Lacerda *et al.* 2020] LACERDA, G.; PETRILLO, F.; PIMENTA, M. ; GUÉHÉNEUC, Y. G.. **Code smells and refactoring: A tertiary systematic review of challenges and observations**. Journal of Systems and Software, 167:110610, 2020.

[Lanza and Marinescu 2006] LANZA, M.; MARINESCU, R.. **Object-Oriented Metrics in Practice**. Springer, Heidelberg, 2006.

[Lazar, Feng and Hochheiser 2017] LAZAR, J.; FENG, J. H. ; HOCHHEISER, H.. **Research methods in human-computer interaction**. Morgan Kaufmann, 2017.

[Le *et al.* 2018] LE, D. M.; LINK, D.; SHAHBAZIAN, A. ; MEDVIDOVIC, N.. **An empirical study of architectural decay in open-source software**. In: 2018 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE (ICSA), p. 176–17609, April 2018.

[Li, Avgeriou and Liang 2015] LI, Z.; AVGERIOU, P. ; LIANG, P.. **A systematic mapping study on technical debt and its management**. Journal of Systems and Software, 101:193 – 220, 2015.

[Lim, Taksande and Seaman 2012] LIM, E.; TAKSANDE, N. ; SEAMAN, C.. **A balancing act: What software practitioners have to say about technical debt**. IEEE Software, 29(6):22–27, Nov 2012.

[Lin *et al.* 2016] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: PROCEEDINGS OF THE 2016 24TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE 2016, p. 535–546, New York, NY, USA, 2016. Association for Computing Machinery.

[MacCormack, Rusnak and Baldwin 2006] MACCORMACK, A.; RUSNAK, J. ; BALDWIN, C.. **Exploring the structure of complex software designs: An empirical study of open source and proprietary code**. Manage. Sci., 52(7):1015–1030, 2006.

[Macia 2013] MACIA, I.. **On the Detection of Architecturally-Relevant Code Anomalies in Software Systems**. PhD thesis, Pontifical Catholic University of Rio de Janeiro, Informatics Department, 2013.

[Macia *et al.* 2012] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying architecture degradation symptoms**. In: CSMR12, p. 277–286, March 2012.

[Macia *et al.* 2012a] MACIA, I.; ARCOVERDE, R.; CIRILO, E.; GARCIA, A. ; VON STAA, A.. **Supporting the identification of architecturally-relevant code anomalies**. In: ICSM12, p. 662–665, Sept 2012.

[Macia *et al.* 2012b] MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N. ; VON STAA, A.. **Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems**. In: AOSD '12, p. 167–178, New York, NY, USA, 2012. ACM.

[Mariani and Vergilio 2017] MARIANI, T.; VERGILIO, S. R.. **A systematic review on search-based refactoring**. Information and Software Technology, 83:14–34, 2017.

[Marinescu, 2004] MARINESCU. **Detection strategies: metrics-based rules for detecting design flaws**. In: PROCEEDINGS OF 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM); CHICAGO, USA, p. 350–359, 2004.

[Martin 2002] MARTIN, R.. **Agile Principles, Patterns, and Practices**. Prentice Hall, New Jersey, 2002.

[Martin 2008] MARTIN, R. C.. **Clean Code: A Handbook of Agile Software Craftsmanship**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[Martin and Martin 2006] MARTIN, R. C.; MARTIN, M.. **Agile Principles, Patterns, and Practices in C# (Robert C. Martin)**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[Mattmann *et al.* 2006] MATTMANN, C.; CRICHTON, D.; MEDVIDOVIC, N. ; HUGHES, S.. **A software architecture-based framework for highly distributed and data intensive scientific applications**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE

ON SOFTWARE ENGINEERING: SOFTWARE ENGINEERING ACHIEVEMENTS TRACK; SHANGHAI, CHINA, p. 721–730, 2006.

[McCallum 2002] MCCALLUM, A. K.. **Mallet: A machine learning for language toolkit (2002)**, 2002.

[McIntosh *et al.* 2014] MCINTOSH, S.; KAMEI, Y.; ADAMS, B. ; HASSAN, A. E.. **The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects**. In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 192–201, Hyderabad, India, 2014.

[Moha *et al.* 2010] MOHA, N.; GUEHENEUC, Y.; DUCHIEN, L. ; MEUR, A. L.. **Decor: A method for the specification and detection of code and design smells**. IEEE Transaction on Software Engineering, 36:20–36, 2010.

[Mohan, Greer and McMullan 2016] MOHAN, M.; GREER, D. ; MCMULLAN, P.. **Technical debt reduction using search based automated refactoring**. Journal of Systems and Software, 120:183–194, 2016.

[Moreno *et al.* 2013] MORENO, L.; APONTE, J.; SRIDHARA, G.; MARCUS, A.; POLLOCK, L. ; VIJAY-SHANKER, K.. **Automatic generation of natural language summaries for java classes**. In: 2013 21ST INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 23–32, May 2013.

[Murphy-Hill, Parnin and Black 2012] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A. P.. **How we refactor, and how we know it**. IEEE Transactions on Software Engineering, 38(1):5–18, Jan 2012.

[Murphy-Hill and Black 2008a] MURPHY-HILL, E.; BLACK, A. P.. **Seven habits of a highly effective smell detector**. In: PROCEEDINGS OF THE 2008 INTERNATIONAL WORKSHOP ON RECOMMENDATION SYSTEMS FOR SOFTWARE ENGINEERING, RSSE '08, p. 36–40, New York, NY, USA, 2008. ACM.

[Murphy-Hill and Black 2008b] MURPHY-HILL, E.; BLACK, A. P.. **Refactoring tools: Fitness for purpose**. IEEE Software, 25(5):38–44, Sep. 2008.

[Murphy-Hill and Black 2010] MURPHY-HILL, E.; BLACK, A. P.. **An interactive ambient visualization for code smells**. In: PROCEEDINGS OF

THE 5TH INTERNATIONAL SYMPOSIUM ON SOFTWARE VISUALIZATION; SALT LAKE CITY, USA, p. 5–14. ACM, 2010.

[Nguyen *et al.* 2011] NGUYEN, T. T.; NGUYEN, H. V.; NGUYEN, H. A. ; NGUYEN, T. N.. **Aspect recommendation for evolving software.** In: ICSE'11, p. 361–370, New York, NY, USA, 2011. ACM.

[Nunes *et al.* 2014] NUNES, C.; GARCIA, A.; LUCENA, C. ; LEE, J.. **Heuristic expansion of feature mappings in evolving program families.** Software: Practice and Experience, 44(11):1315–1349, 2014.

[Nyamawe *et al.* 2019] NYAMAWE, A. S.; LIU, H.; NIU, N.; UMER, Q. ; NIU, Z.. **Automated recommendation of software refactorings based on feature requests.** In: 2019 IEEE 27TH INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE (RE), p. 187–198. IEEE, 2019.

[Oizumi 2019] NALEPA OIZUMI, W.. **Recommendation of refactorings for improving dependability attributes.** In: 2019 IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING WORKSHOPS (ISSREW), p. 89–92, 2019.

[Oizumi *et al.* 2014a] OIZUMI, W.; GARCIA, A.; COLANZI, T.; FERREIRA, M. ; STAA, A.. **When code-anomaly agglomerations represent architectural problems? An exploratory study.** In: PROCEEDINGS OF THE 2014 BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES); MACEIO, BRAZIL, p. 91–100, 2014.

[Oizumi *et al.* 2014b] OIZUMI, W.; GARCIA, A.; SOUSA, L.; ALBUQUERQUE, D. ; CEDRIM, D.. **Towards the synthesis of architecturally-relevant code anomalies.** In: PROCEEDINGS OF THE 11TH WORKSHOP ON SOFTWARE MODULARITY; MACEIO, BRAZIL, p. 39–52, 2014.

[Oizumi *et al.* 2015] OIZUMI, W.; GARCIA, A.; COLANZI, T.; STAA, A. ; FERREIRA, M.. **On the relationship of code-anomaly agglomerations and architectural problems.** Journal of Software Engineering Research and Development, 3(1):1–22, 2015.

[Oizumi *et al.* 2016] OIZUMI, W.; GARCIA, A.; SOUSA, L. S.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '16, 2016.

[Oizumi *et al.* 2017] OIZUMI, W.; SOUSA, L.; GARCIA, A.; OLIVEIRA, R.; OLIVEIRA, A.; AGBACHI, O. I. A. B. ; LUCENA, C.. **Revealing design problems in stinky code: A mixed-method study**. In: PROCEEDINGS OF THE 11TH BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES, AND REUSE, SBCARS '17, p. 5:1–5:10, New York, NY, USA, 2017. ACM.

[Oizumi *et al.* 2018] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; GARCIA, A.; AGBACHI, A. B.; OLIVEIRA, R. ; LUCENA, C.. **On the identification of design problems in stinky code: experiences and tool support**. Journal of the Brazilian Computer Society, 24(1):13, Oct 2018.

[Oizumi *et al.* 2019] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; CARVALHO, L.; GARCIA, A.; COLANZI, T. ; OLIVEIRA, R.. **On the density and diversity of degradation symptoms in refactored classes: A multi-case study**. In: IEEE 30TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), October 2019.

[Oizumi *et al.* 2020] OIZUMI, W.; CEDRIM, D.; SOUSA, L.; BIBIANO, A. C.; OLIVEIRA, A.; GARCIA, A. ; TENORIO, D.. **Recommending composite refactorings for smell removal: Heuristics and evaluation**. In: 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), 2020.

[Oliveira, Valente and Terra 2016] SILVA, M. C. O.; VALENTE, M. T. ; TERRA, R.. **Does technical debt lead to the rejection of pull requests?** In: PROCEEDINGS OF THE 12TH BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS, SBSI '16, p. 248–254, 2016.

[Oliveira *et al.* 2017] OLIVEIRA, R. F.; DA SILVA SOUSA, L.; DE MELLO, R. M.; VALENTIM, N. M. C.; LOPES, A.; CONTE, T.; GARCIA, A. F.; DE OLIVEIRA, E. C. C. ; DE LUCENA, C. J. P.. **Collaborative identification of code smells: A multi-case study**. In: 39TH IEEE/ACM INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: SOFTWARE ENGINEERING IN PRACTICE TRACK, ICSE-SEIP, p. 33–42, 2017.

[Oliveira *et al.* 2020] OLIVEIRA, R. F.; DE MELLO, R. M.; FERNANDES, E.; GARCIA, A. ; LUCENA, C.. **Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers**. Inf. Softw. Technol., 120, 2020.

[Ouni *et al.* 2017] OUNI, A.; KESSENTINI, M.; Ó CINNÉIDE, M.; SAHRAOUI, H.; DEB, K. ; INOUE, K.. **More: A multi-objective refactoring rec-**

ommendation approach to introducing design patterns and fixing code smells. Journal of Software: Evolution and Process, 29(5):e1843, 2017.

[Paixão *et al.* 2020] PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J. ; ARVONIO, E.. **Behind the Intents: An In-Depth Empirical Study on Software Refactoring in Modern Code Review**, p. 125–136. Association for Computing Machinery, New York, NY, USA, 2020.

[Palomba *et al.* 2014] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R. ; LUCIA, A. D.. **Do they really smell bad? a study on developers' perception of bad code smells.** In: 2014 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 101–110, Sept 2014.

[Palomba *et al.* 2017] PALOMBA, F.; ZAIDMAN, A.; OLIVETO, R. ; DE LUCIA, A.. **An exploratory study on the relationship between changes and refactoring.** In: 2017 IEEE/ACM 25TH ICPC, p. 176–185. IEEE, 2017.

[Pecorelli *et al.* 2020] PECORELLI, F.; PALOMBA, F.; KHOMH, F. ; DE LUCIA, A.. **Developer-driven code smell prioritization.** In: INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES, 2020.

[Perry and Wolf 1992] PERRY, D. E.; WOLF, A. L.. **Foundations for the study of software architecture.** SIGSOFT Softw. Eng. Notes, 17(4):40–52, Oct. 1992.

[Peruma *et al.* 2022] PERUMA, A.; SIMMONS, S.; ALOMAR, E. A.; NEWMAN, C. D.; MKAOUER, M. W. ; OUNI, A.. **How do i refactor this? an empirical study on refactoring trends and topics in stack overflow.** Empirical Software Engineering, 27(1):1–43, 2022.

[Pinto and Kamei 2013] PINTO, G. H.; KAMEI, F.. **What programmers say about refactoring tools?: An empirical investigation of stack overflow.** In: PROCEEDINGS OF THE 2013 ACM WORKSHOP ON WORKSHOP ON REFACTORING TOOLS, WRT '13, p. 33–36, New York, NY, USA, 2013. ACM.

[Prates *et al.* 2000] PRATES, R. O.; DE SOUZA, C. S. ; BARBOSA, S. D. J.. **Methods and tools: A method for evaluating the communicability of user interfaces.** interactions, 7(1):31–38, Jan. 2000.

[Rachow 2019] RACHOW, P.. **Refactoring decision support for developers and architects based on architectural impact**. In: 2019 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE COMPANION (ICSA-C), p. 262–266, March 2019.

[Ran *et al.* 2015] MO, R.; CAI, Y.; KAZMAN, R. ; XIAO, L.. **Hotspot patterns: The formal definition and automatic detection of architecture smells**. In: SOFTWARE ARCHITECTURE (WICSA), 2015 12TH WORKING IEEE/IFIP CONFERENCE ON, p. 51–60, May 2015.

[Ratzinger, Fischer and Gall 2005] RATZINGER, J.; FISCHER, M. ; GALL, H.. **Improving evolvability through refactoring**, volumen 30. ACM, 2005.

[Rebai *et al.* 2020] REBAI, S.; KESSENTINI, M.; ALIZADEH, V.; SGHAIER, O. B. ; KAZMAN, R.. **Recommending refactorings via commit message analysis**. Information and Software Technology, 126:106332, 2020.

[Ricci *et al.* 2011] RICCI, F.; ROKACH, L. ; SHAPIRA, B.. **Introduction to Recommender Systems Handbook**, p. 1–35. Springer US, Boston, MA, 2011.

[Rizzi *et al.* 2018] RIZZI, L.; FONTANA, F. A. ; ROVEDA, R.. **Support for architectural smell refactoring**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON REFACTORING, IWoR 2018, p. 7–10, New York, NY, USA, 2018. Association for Computing Machinery.

[Rosik *et al.* 2008] ROSIK, J.; LE GEAR, A.; BUCKLEY, J. ; ALI BABAR, M.. **An industrial case study of architecture conformance**. In: PROCEEDINGS OF THE SECOND ACM-IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, p. 80–89, 2008.

[Sant'Anna *et al.* 2007] SANT'ANNA, C.; FIGUEIREDO, E.; GARCIA, A. ; LUCENA, C.. **On the modularity assessment of software architectures: Do my architectural concerns count**. In: PROC. INTERNATIONAL WORKSHOP ON ASPECTS IN ARCHITECTURE DESCRIPTIONS (AARCH. 07), AOSD, 2007.

[Schach *et al.* 2002] SCHACH, S.; JIN, B.; WRIGHT, D.; HELLER, G. ; OFFUTT, A.. **Maintainability of the linux kernel**. Software, IEE Proceedings -, 149(1):18–23, 2002.

[Shadish, Cook and Campbell 2001] SHADISH, W. R.; COOK, T. D. ; CAMPBELL, D. T.. **Experimental and Quasi-Experimental Designs for Generalized Causal Inference**. Houghton Mifflin, 2 edition, 2001.

[Sharma 2020] SHARMA, T.. **A taxonomy of software smells**, Mar. 2020.

[Sharma and Spinellis 2018] SHARMA, T.; SPINELLIS, D.. **A survey on software smells**. J. Syst. Softw., 138:158 – 173, 2018.

[Sharma *et al.* 2016] SHARMA, T.; MISHRA, P. ; TIWARI, R.. **Designite: A software design quality assessment tool**. In: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON BRINGING ARCHITECTURAL DESIGN THINKING INTO DEVELOPERS' DAILY ACTIVITIES, BRIDGE '16, p. 1–4, New York, NY, USA, 2016. ACM.

[Silva, Galster and Gilson 2021] SILVA, C. C.; GALSTER, M. ; GILSON, F.. **Topic modeling in software engineering research**. Empirical Software Engineering, 26(6):1–62, 2021.

[Silva, Tsantalis and Valente 2016] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor? confessions of github contributors**. In: PROCEEDINGS OF THE 2016 24TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE 2016, p. 858–870, New York, NY, USA, 2016. ACM.

[Soares *et al.* 2002] SOARES, S.; LAUREANO, E. ; BORBA, P.. **Implementing distribution and persistence aspects with aspectj**. In: PROCEEDINGS OF THE 17TH ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS; SEATTLE, USA, p. 174–190. ACM Press, 2002.

[Sousa *et al.* 2017] SOUSA, L.; OLIVEIRA, R.; GARCIA, A.; LEE, J.; CONTE, T.; OIZUMI, W.; DE MELLO, R.; LOPES, A.; VALENTIM, N.; OLIVEIRA, E. ; LUCENA, C.. **How do software developers identify design problems?: A qualitative analysis**. In: PROCEEDINGS OF 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES'17, 2017.

[Sousa *et al.* 2018] SOUSA, L.; OLIVEIRA, A.; OIZUMI, W.; BARBOSA, S.; GARCIA, A.; LEE, J.; KALINOWSKI, M.; DE MELLO, R.; FONSECA, B.; OLIVEIRA, R.; LUCENA, C. ; PAES, R.. **Identifying design problems in the source code: A grounded theory**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 921–931, New York, NY, USA, 2018. ACM.

[Sousa *et al.* 2020a] SOUSA, L.; CEDRIM, D.; GARCIA, A.; OIZUMI, W.; BIBIANO, A. C.; TENORIO, D.; KIM, M. ; OLIVEIRA, A.. **Characterizing and identifying composite refactorings: Concepts, heuristics and patterns.** In: 17TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), 2020.

[Sousa *et al.* 2020b] SOUSA, L.; OIZUMI, W.; GARCIA, A.; OLIVEIRA, A.; CEDRIM, D. ; LUCENA, C.. **When are smells indicators of architectural refactoring opportunities: A study of 50 software projects.** In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC '20, p. 354–365, New York, NY, USA, 2020. Association for Computing Machinery.

[SpotBugs 2019] SPOTBUGS. **Spotbugs: Find bugs in java programs**, May 2019.

[Suryanarayana, Samarthyam and Sharma 2014] SURYANARAYANA, G.; SAMARTHYAM, G. ; SHARMA, T.. **Refactoring for Software Design Smells: Managing Technical Debt.** Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[Taylor *et al.* 2009] TAYLOR, R.; MEDVIDOVIC, N. ; DASHOFY, E.. **Software Architecture: Foundations, Theory, and Practice.** Wiley Publishing, 2009.

[Terra and Valente 2009] TERRA, R.; DE OLIVEIRA VALENTE, M. T.. **A dependency constraint language to manage object-oriented software architectures.** Softw., Pract. Exper., 39(12):1073–1094, 2009.

[Terra *et al.* 2012] TERRA, R.; VALENTE, M. T.; CZARNECKI, K. ; BIGONHA, R. S.. **Recommending refactorings to reverse software architecture erosion.** In: 2012 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 335–340, 2012.

[Trifu and Marinescu 2005] TRIFU, A.; MARINESCU, R.. **Diagnosing design problems in object oriented systems.** In: WCRE'05, p. 10 pp., Nov 2005.

[Tsantalis, Chaikalis and Chatzigeorgiou 2018] TSANTALIS, N.; CHAIKALIS, T. ; CHATZIGEORGIOU, A.. **Ten years of jdeodorant: Lessons learned from the hunt for smells.** In: 2018 IEEE 25TH SANER, p. 4–14. IEEE, 2018.

[Tsantalis *et al.* 2018] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history.** In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 483–494, New York, NY, USA, 2018. ACM.

[Tufano *et al.* 2015] TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; DI PENTA, M.; DE LUCIA, A. ; POSHYVANYK, D.. **When and why your code starts to smell bad.** In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '15, New York, NY, USA, 2015. ACM.

[Ulungu *et al.* 1999] ULUNGU, E. L.; TEGHEM, J.; FORTEMPS, P. ; TUYTTENS, D.. **Mosa method: a tool for solving multiobjective combinatorial optimization problems.** Journal of multicriteria decision analysis, 8(4):221, 1999.

[Vidal, Marcos and Díaz Pace 2014a] VIDAL, S. A.; MARCOS, C. ; DÍAZ PACE, J. A.. **Analyzing the history of software systems to predict class changes.** In: IEEE BIENNIAL CONGRESS OF ARGENTINA (ARGENCON), 2014.

[Vidal, Marcos and Díaz Pace 2014b] VIDAL, S.; MARCOS, C. ; DÍAZ PACE, J. A.. **An approach to prioritize code smells for refactoring.** Automated Software Engineering, p. 1–32, 2014.

[Vidal, Marcos and Díaz-Pace 2016] VIDAL, S. A.; MARCOS, C. ; DÍAZ-PACE, J. A.. **An approach to prioritize code smells for refactoring.** Automated Software Engg., 23(3):501–532, Sept. 2016.

[Vidal *et al.* 2015] VIDAL, S.; VAZQUEZ, H.; DIAZ-PACE, J. A.; MARCOS, C.; GARCIA, A. ; OIZUMI, W.. **JSpIRIT: a flexible tool for the analysis of code smells.** In: 2015 34TH INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY (SCCC), p. 1–6, Nov 2015.

[Vidal *et al.* 2016] VIDAL, S.; GUIMARAES, E.; OIZUMI, W.; GARCIA, A.; PACE, A. D. ; MARCOS, C.. **Identifying architectural problems through prioritization of code smells.** In: SBCARS16, p. 41–50, Sept 2016.

[Vidal *et al.* 2018] VIDAL, S.; ZULLIANI, S.; MARCOS, C.; PACE, J. ; OTHERS. **Assessing the refactoring of brain methods.** ACM Transactions on

Software Engineering and Methodology (TOSEM), 27(1):2, 2018.

[Vidal *et al.* 2019] VIDAL, S.; OIZUMI, W.; GARCIA, A.; PACE, A. D. ; MARCOS, C.. **Ranking architecturally critical agglomerations of code smells**. Science of Computer Programming, 182:64 – 85, 2019.

[Wettel and Lanza 2008] WETTEL, R.; LANZA, M.. **Visually localizing design problems with disharmony maps**. In: PROCEEDINGS OF THE 4TH ACM SYMPOSIUM ON SOFTWARE VISUALIZATION, p. 155–164. ACM, 2008.

[Wong *et al.* 2011] WONG, S.; CAI, Y.; KIM, M. ; DALTON, M.. **Detecting software modularity violations**. In: IN PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING; HONOLULU, USA, p. 411–420, 2011.

[Xiao *et al.* 2016] XIAO, L.; CAI, Y.; KAZMAN, R.; MO, R. ; FENG, Q.. **Identifying and quantifying architectural debt**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '16, p. 488–498, New York, NY, USA, 2016. ACM.

[Yamanaka *et al.* 2021] YAMANAKA, J.; HAYASE, Y. ; AMAGASA, T.. **Recommending extract method refactoring based on confidence of predicted method name**, 2021.

[Yamashita and Moonen 2012] YAMASHITA, A.; MOONEN, L.. **Do code smells reflect important maintainability aspects?** In: ICSM12, p. 306–315, 2012.

[Yamashita and Moonen 2013] YAMASHITA, A.; MOONEN, L.. **Do developers care about code smells? an exploratory survey**. In: 2013 20TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 242–251, Oct 2013.

[Yamashita *et al.* 2015] YAMASHITA, A.; ZANONI, M.; FONTANA, F. A. ; WALTER, B.. **Inter-smell relations in industrial and open source systems: A replication and comparative analysis**. In: SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2015 IEEE INTERNATIONAL CONFERENCE ON, p. 121–130, Sept 2015.

[Yin 2015] YIN, R. K.. **Qualitative research from start to finish**. Guilford publications, 2015.

[Young 2005] YOUNG, T. J.. **Using aspectj to build a software product line for mobile devices. MSc dissertation**. In: UNIVERSITY OF BRITISH COLUMBIA, DEPARTMENT OF COMPUTER SCIENCE, p. 1–6, 2005.

[Zimmermann 2017] ZIMMERMANN, O.. **Architectural refactoring for the cloud: a decision-centric view on cloud migration**. Computing, 99(2):129–145, Feb 2017.

[de Souza *et al.* 2009] DE SOUZA, C. S.; LEITÃO, C. F.. **Semiotic engineering methods for scientific research in hci**. Synthesis Lectures on Human-Centered Informatics, 2(1):1–122, 2009.