UNIVERSITY OF OSLO
Department of informatics

A Literature Review on Code Smells and Refactoring

Master thesis
60 credits

Ruben Drøsdal Wangberg

18. May 2010

2010

A Literature Review on Code Smells and Refactoring

Ruben Drøsdal Wangberg

http://www.duo.uio.no/

Trykk: Reprosentralen, Universitetet i Oslo

Summary

This thesis reports the results from a literature review conducted on the topic of code smells and refactoring. Code smells are segments of the source code that display potential design issues. Refactoring is the process for modifying source code to improve its quality (e.g. maintainability) without affecting its functionality. Improving the code design is important for reducing costs involved in maintenance projects. Thus, refactoring has become an integral part of developer's everyday work, yet effects of refactoring on software quality are not well understood. I believe that an overview of the available empirical evidence on the effects of code smells and refactoring in software projects, as well as methods and tools available for supporting refactoring activities, will contribute significantly to the current practices in industry and at the same time, will provide a constructive stance towards scientific work within this field. This review was conducted on the three major databases related to software engineering: IEEE Xplore, ACM Digital Library and ISI Web of Knowledge, based on most of the features of a systematic literature review protocol. The main findings from this review are: A relatively small portion of the identified work reported empirical studies (24% of the articles) as opposed to design research contributions (61% of the articles), the latter includes both methods and tools for code smells detection and refactoring support. Only 13.8% of these design contributions reported *any* type of validation or evaluation in an industrial setting and of these, only half of them reported a thorough validation in a realistic setting (i.e. industrial). Most of the design contributions (22 out of 28) were partly or wholly concerned with the detection of code smells. The literature review identified several tools available to help developers detect and remove some code smells, but no significant evaluation was reported that could help to select the most suitable tool for a given context. The review found a significant increase in the number of publications on code smells and refactoring since 2005. Nevertheless, the review found in general a lack of empirically sound evidence that could help developers and architects interpreting, analyzing, and choosing the best refactoring strategies for improving maintainability. This leaves substantial areas for improvement within this area, mainly development of: (1) better and more concrete refactoring guidelines that are based on sound empirical evidence, and (2) better evaluation frameworks that could enable practitioners to choose the adequate tools and methods that would fit their specific needs in a given context.

Acknowledgements

The person who says that writing a master thesis is easy is either a very silly person or extremely talented. I am neither and would not have managed to survive this process without help from a bunch of extraordinary people.

I would like to thank my supervisor Aiko Yamashita for invaluable feedback and discussions throughout the work with this thesis and Dag Sjøberg for a thorough review and good help. I would also like to thank the Employees of Simula Research Laboratory for providing me with an inspiring work environment. Other grateful remarks go to Alexander Teinum and Oddmund Strømme for providing me with valuable feedback and to Håvard Tegelsrud and Jørgen Frøysadal for providing me with coffee, time wasting and spiritual guidance the last 5 years.

Last but not least I would like to thank my mother and father for the amazing genes and moderate support and Line for putting up with me throughout the last year of grumbling, refactorings and bad smells.

Oslo, May 2010
Ruben Drøsdal Wangberg

Table of contents

List of figures

List of tables

1 Introduction

1.1 Motivation and current state

Software maintenance projects are very costly. The total maintenance costs of a software project are estimated to 40%-70% of the total cost of the lifecycle of the project [1]. Consequently, reducing the effort spent on maintenance can be seen as a natural way of reducing the overall costs of a software project. This is one of the main reasons for the recent interest in concepts such as refactoring and code smells. Refactoring is to "improve the design after it has been written" [2]. Doing this will increase the understandability of code, make it easier to implement new features and debug the code [2]. Code smells are symptoms or indicators in the code suggesting that something may need to be refactored [2].

Refactoring does not add functionality, but is done under the assumption that it will make the code easier to work with. This premise focuses on "effectively spending time and money in order to save time and money in the future". It is difficult to judge which areas of the code and what kind of refactoring to use without measureable evidence on the effects of refactoring. Empirical evidence could make these decisions easier. The field of code smells and refactoring is fairly young, and consequently, it seems that empirical evidence is scarce. The purpose behind this thesis is to investigate what type of research exists within this field and to present an overview of the current state of the art with respect to code smells and refactoring research. More specifically, what has been the output from the research community that might help programmers to detect code smells, decide when to refactor, and actually perform the different refactorings? In order to have a comprehensive view on the different types of contributions, this review covered empirical contributions, design research contributions, theoretical contributions and summarizing contributions.

1.2 Objective of Research and Research Questions

This thesis consists of a review of relevant literature in the software engineering field concerned with refactoring and code smells. The main objective of the review is to examine the current research work and present the most relevant and interesting contributions that might be useful for practitioners working with these concepts. At the same time, a systematic and comprehensive overview of the research could constitute a contribution for the research community since it can facilitate an evaluative and strategic stance and discussion of the future directions within the field.

The main objective of this thesis is to get an overview of research related to each of the stages of the refactoring process: Detecting code smells, making decisions on which refactorings to choose, and performing the refactoring. The review also attempts to identify which methods and tools have been created to support these different stages of refactoring. The following questions (divided in Research Question and Sub-Questions) were formulated as a basis for identifying, analyzing and discussing the existing literature:

RQ: *What is the state of art in SE research to support analysis and detection of code smells and refactoring decisions?*

SQ1: *What is the state of art in SE research with respect to investigating empirically the effects of code smells in development and maintenance projects?*

SQ2: *What is the state of art in SE research with respect to investigating empirically the effects of refactorings in development and maintenance projects?*

SQ3: *Which tools and methods have been developed to support code smell analysis and detection or refactoring decisions?*

1.3 Research Method

A literature review was chosen as a suitable methodology to answer the research questions. A protocol for systematic literature review [3] was used as a guide to achieve a structured process and robust evidence for the validity of the results, although the protocol was not followed in details, which means the review reported here is not a *systematic* one. At the same time, certain level of flexibility was prioritized due to the exploratory nature of the study, bearing in mind that this is a relatively new topic, which may lack a standard terminology (many features of a systematic literature review could be extremely time consuming, thus more likely to be out of the scope for an MSc thesis period).

1.4 Overview of the Thesis

The rest of the thesis is organized as follows: Section 2 introduces the context by presenting relevant background information and related work. The research methodology for this review is presented in Section 3. Section 4 presents the results and findings from the literature review and a summary of the identified contributions. Section 5 discusses some of the findings under the light of the research questions. Section 6 discusses threats to validity of this study. Section 7 presents the conclusions of this study, finalizing with Section 8, which presents future work.

2 Background and Related Work

This section explains briefly the history and nature of code smells and refactoring, and introduces related work on literature reviews conducted in the field.

2.1 Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written." [2]

The term *refactoring* precedes the code smell definition, but is still fairly young. Refactoring code has been done informally before the term was coined, but was first formally described by William Opdyke in his Ph.D. dissertation ([2], [4], [5]). Opdyke was also the researcher that coined the term, together with Ralph Johnson [4]. It is, however, after Fowler's book [2] that the term and practice started gaining popularity. Refactoring is a practice that has gained an increasing popularity and usage the last years and has been a common topic on large software practitioner conferences, such as *DevWeek* [6], *JavaOne* [7], *JavaZone* [8], and *Scandinavian Developer Conference* [8].

An example of a refactoring could be e*xtract method:* If a method is too long, it should be decomposed, using this refactoring. Find a clump of code (within the long method) that goes well together, create a new method with a descriptive name and move the code into the new method. If local variables are being used, they need to be passed as parameters. The last step is to add a call to the new method and test the code. [2]

```
void printOwing() {
  printBanner();
  //print details
  System.out.println ("name: " + _name);
  System.out.println ("amount     " + getOutstanding());
  }

should be refactored to:

void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}
void printDetails (double outstanding) {
  System.out.println ("name: " + _name);
  System.out.println ("amount     " + outstanding);
}
```

2.2 Code Smells

A code smell is a symptom or indicator in the source code that indicates potential problems. This is not to be confused with compiler errors or warnings or other signs of code that is not working properly. Code smells only *indicate* that the maintainability of the specific code might not be as good as its potential, or to put it in the words of Fowler, "*Any fool can write code that a computer can understand. Good programmers write code that humans can understand*" (p 15 [2]). The importance of writing code that computers understand is obvious, but lately the importance of writing understandable code, has got more focus and acceptance. It is believed that even small efforts could lead to improve the understandability of the code, and this may decrease the developers' effort on understanding and localizing relevant information for their tasks. This reduction of effort can lead to considerable reductions in maintenance costs. If one was able to get 1% reduction in effort needed for maintenance, this would count up to quite a lot of money if the project costs is estimated to $50 000 000.

The metaphor of code that smells was made popular in Fowler's book as "bad smells in code". It is now known as *code smells* and is described in Kent Beck and Martin Fowler's words as "*certain structure in the code that suggests (sometimes they scream for) the possibility of refactoring*" (p 75 in [2]). Some examples of these smells are:

- **Long method** – a method that has grown too large.
- **Lazy class** – a class that is not doing enough.
- **Comments** – comments might suggest that the commented code is bad.
- **Feature envy** – a method that is more interested in a class other than the one it actually is in.

The identifications of code smells are useful in the sense that they might constitute prescriptive guidance for performing certain types of refactoring. An example of this is illustrated in Figure 1, where in order to eliminate the *Feature Envy* smell, a potential refactoring could be *Move Method.*

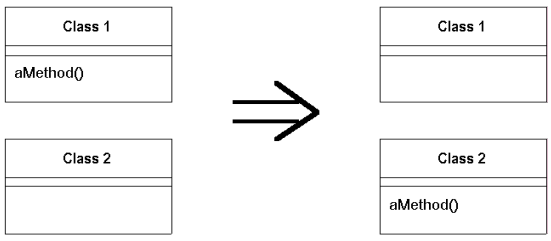


Figure 1. UML diagram describing an example of the Move Method refactoring [2]

2.3 Related Work

In this section, work related to the thesis topic is presented. The related work includes both early work on code smells and refactoring and other reviews and summarization work.

Van Emden and Moonen [9] provided the first formalization of code smells and described a tool that could detect them. Marinescu [10] further formalized the definition of code smells and extended the detection to a wider range of code smells and a number of design principle violations. Fowler introduced formalized refactoring in his book [2] and Kerievsky build upon that work when he introduced more refactorings and code smells in his book on refactoring [11] in 2004. Summarizing contributions to the field includes Mens and Tourwé [12]. They present a survey on refactoring, which mainly discusses different aspects of the refactoring process: general ideas, refactoring activities, various formalisms and techniques, considerations and how refactoring fits the software development processes.

In addition to work thematically related to this thesis, it would be meaningful to address some work that is related in terms of methodology. The usage of literature reviews in the software engineering field is relatively scarce and thus it is relevant to look into related research work for inspiration. Kitchenham proposed guidelines for systematic literature reviews appropriate for software engineering researchers [3], and Brereton et al. [13] presented lessons learned from performed systematic literature reviews within the software engineering domain. Holt [14] and Dybå [15] both performed systematic reviews within this domain.

3 Research Methodology

This chapter has two main purposes. First, it is meant to describe and argument for the choices of the methodological strategies followed in the review. Second, it documents the process followed to arrive to the results. By describing the methodological steps and documenting the data collection process, this review can be replicated and this will support the internal validity of the study.

Literature review was selected as the research method in this thesis work, since investigating the published work within this specific area was assumed to be sufficiently comprehensive to provide substantial information to answer the research questions. A literature review is a text that aims to gather relevant information in a specific field. Kitchenham [3] describes it as ‘‘*a means of evaluating and interpreting all available research relevant to a particular research question or topic area or phenomenon of interest*’’. While Kitchenham describes a *systematic* review, the purpose remains the same for the literature review reported in this thesis. It was decided to use some of the features of a systematic literature review, but at the same time keep a certain level of flexibility on the method, due to the exploratory nature of the study and the fact that the topic is relatively new and involves non-standardized terminology.

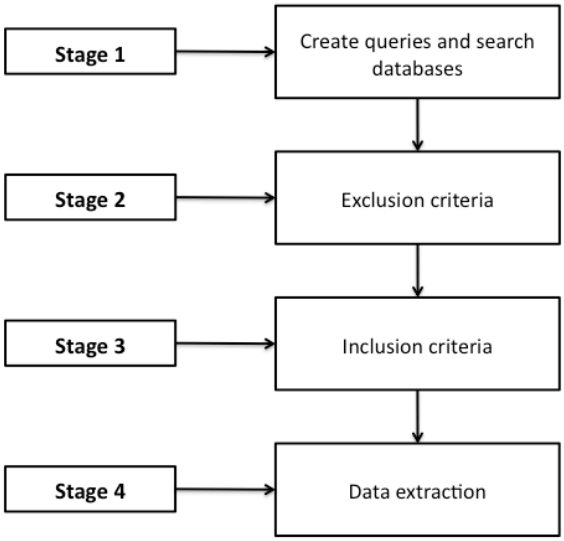


Figure 2: The main stages of the literature review.

Within this chapter, the steps used from the systematic review method will be reported alongside with each of the stages of the review, its inputs and corresponding outputs. Figure 2 presents the steps of the data collection part of the review and is inspired by Dybå and Dingsøyr’s work [15], which will be used as subsections in the remainder of this chapter.

3.1 Create Queries and Search Databases

Databases

Since the review topic consists of code smells and refactoring, the focus was placed in the main scholar research databases that specialize on computer and software research. The three largest and most commonly used databases in the software engineering field are ACM Digital Library[16], IEEE Xplore [17] and ISI Web of Knowledge [18]. Google Scholar was initially considered as a source, but it was excluded since it become clear that it would be out of scope. The first query created for the Google Scholar search (See Table 1) gave 8820 results in Google Scholar. The vast majority of these results were outside the field of software engineering. The query was modified to include only material from the year 2000 and onwards, and that specifically mentioned the keyword "software". This reduced the results to 657, but it was still 5 times bigger than the aggregated resulting set from all the three other databases. By manually inspecting the first 20 results, it was not possible to find any relevant article that did not already exist in the results from the other databases. This does not imply that the 657 results from Google Scholar would not contribute to the review. However, it indicated that the effort that would have to be spent, manually going through these results would probably outweigh the impact of these, presumably "grey", contributions. Grey literature is a term used when referring to a body of materials that is not present in research databases or published through conventionally scientific channels. The Grey Literature Network Service, which facilitates distribution and access to grey literature, defines it as "*Information produced on all levels of government, academics, business and industry in electronic and print formats not controlled by commercial publishing i.e. where publishing is not the primary activity of the producing body*" [19]. Such contributions are not necessarily thoroughly scientific in its form and will often not document their claims sufficiently and are by nature rarely peer-reviewed. Using grey literature in a scientific thesis may subsequently lead to problems with the validity of the work, especially concerning to empirical contributions.

Table 1: The initial query tested for Google scholar.

```
("code smell*" OR "bad smell*" OR "design
principle violation*" OR "structural symptom*")
AND ("tool" OR "method" OR "technique" OR
"knowledge" OR "decision")
```

Queries

The queries required for the literature review were mainly based on the research question: *What is the state of art in SE research to support analysis and detection of code smells and refactoring decisions?*

Initially, a generic query was constructed in order to adapt it to different databases and their specific syntax. All the information needed in order to answer the research questions was related to term "code smell" (i.e., detecting and analyzing code smells, performing refactoring to remove code smells, and refactoring decisions on which smells to remove). Neither of these topics would be meaningful to discuss or analyze without mentioning code smells. For this reason, the term "code smell" was central to the query. In order to filter out the non-relevant results, an additional restriction was added to specify that the contributions on code smells should be limited to knowledge, methods or tools. The resulting preliminary query is:

"code smell" and (tool or method or knowledge)

To decrease the chance of missing out results due to different wordings, the query included the most used synonyms of the term code smell. The phrases included wildcard characters to include plural forms. This resulted in the following query:

("code smell" OR "bad smell*" OR "design principle violation*" or "structural symptom*") and (tool or method or technique or knowledge)*

Some databases allowed limiting the results to time periods. The term "code smell" gained popularity only after it was documented in [2], which was released the last half of 1999. For this reason the queries were limited to contributions made from 2000 and onwards.

Data extraction

A preliminary data extraction sheet in Excel was made to prepare for further steps in the review, (i.e., applying the exclusion and inclusion criteria on the dataset). Tools used in this process were Zotero [20], Firefox, EndNote and Excel. Zotero is a plug-in for Firefox, which is used for managing references, both for academic and grey literature. Zotero has built in support for the most used research databases, including the ones used in this review: ACM Digital Library, IEEE Xplore and ISI Web of Knowledge. Running the queries in each of the search engines of the databases would result in a number of web pages with results.

Table 2: The specific queries and their corresponding databases

ACM Digital Library:	`("code smell*" or "bad smell*" or "design principle violation*" or "structural symptom*") and (tool or method or technique or knowledge)`

IEEE Xplore:	`(((code smell* <in> metadata) or (bad smell* <in> metadata) or (design principle violation* <in> metadata))) <and> (pyr >= 2000 <and> pyr <= 2009) <and> ((knowledge* <in> metadata) or (tool* <in> metadata) or (method* <in> metadata) or (decision* <in> metadata`
	
	`Time: 2000 – present`

ISI Web of knowledge:	`TS=("knowledge" OR "tool*" OR "method*" OR "technique*" OR "decision*") AND TS=("Code smell*" OR "Bad Smell*" OR "Design principle vi*" OR "Structural symptom*")`
	
	`Timespan=2000-2009. Databases=SCI-EXPANDED, CPCI-S, CPCI-SSH`

Zotero provides functionality to import all results on a web page into a built in reference library (as shown in Figure 3). The queries were executed in each of the search engines from the databases and the results were saved as a "library" in Zotero. Where available, a PDF-document of the contribution would be saved as well. If no PDF were found, a HTML page with the title and abstract of the contribution would be saved instead. The PDF-documents were saved in the following format: <authors – year – name of article>. The complete list or references from Zotero were exported to Excel via Endnote (See Figure 4, which displays Zotero functionality for storing the reference list including the PDF files of the articles).

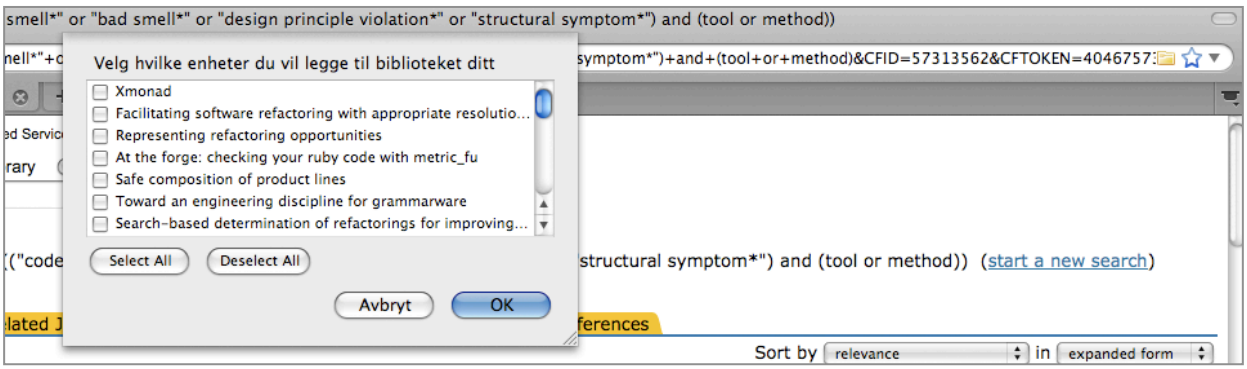


Figure 3: Screenshot showing a list of articles extractable from the search result.

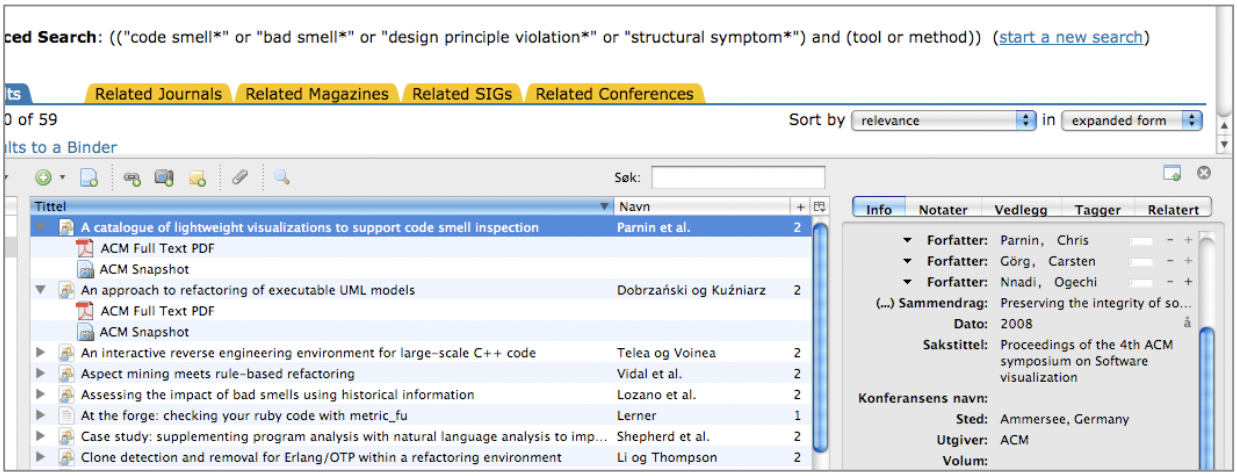


Figure 4: Screenshot showing the references with PDF-articles and data in Zotero.

The following information was found to be adequate to be included in the first extraction sheet: ***Name of article, Author(s), Year of publishing, Publisher, Link to the actual article.*** The reminder of this section will describe how this information was extracted from the reference list and registered in the extraction sheet.

Name of article, Author and Year of publishing

These were all present in the references imported into Excel, for all contributions. Columns with this information were identified and labeled accordingly.

Publisher

The publisher (research database) of each document was not given for every reference, but was obtainable through some of the other columns of information. ISI Web of Knowledge entries were the only ones provided with an "accession number". Some data manipulation on the excel sheet was done (i.e., sorting the sheet on this column and set the publisher for every row with this number to ISI). All articles from ACM Digital Library had the page number column formatted in a reminiscent way and could be separated and marked this way. The articles not marked as either ACM or ISI were subsequently marked as IEEE.

Link to article

Two Java-programs were developed for this purpose. The first program created a file system data structure for the downloaded PDF-documents. This structure stored the directory name where the PDF file was downloaded and the name of the file (besides other additional information). The PDF files and web pages were stored in a SVN repository. Another java-

program was made to create the links to the files so it will print out links in the same order as the list of articles in the extraction sheet. This list was subsequently copied into the extraction sheet. This was done in order to facilitate the access of the actual articles from the extraction sheet.

3.2 Exclusion Criteria

The exclusion criteria were used in order to filter out irrelevant articles. These criteria should be rather clear and straightforward. In case there is a doubt about excluding something or not, it should be kept. Material on the borderline for exclusion will rarely pass the inclusion criteria and therefore would be assumed to not "pollute" the result set anyway. The exclusion criteria used in this review are the following:

a) Articles not related to software development or maintenance of software (e.g. biology)
b) Articles not written in English
c) Articles not applicable to object-oriented (OO) programming languages
d) A position paper, an editorial, preface, discussion, article summary, or summary of tutorials, workshops, panels, poster sessions, book reviews, and conference companions
e) Articles already in the resulting set of articles

To apply the exclusion criteria, the initial set of articles from the queries was screened. All contributions listed in the extraction sheet were marked as either excluded or not. For many contributions, reading the title of the paper would be sufficient, while for some cases it was required to examine the abstract. When the content of neither the title nor the abstract would give enough information whether the article meets the exclusion criteria, the article was included for the second stage screening (the application of the inclusion criteria). The contributions marked as "excluded" were double-checked and then removed from the extraction sheet. Some contributions would be found by more than one database and so the initial result set had several duplicated entries. These were also marked and all but one version were removed from the extraction sheet.

3.3 Inclusion Criteria

The inclusion criteria are the last filtering mechanism that separates the list of possibly relevant articles from the final result set. Articles were only included if they met at least one of the inclusion criteria. An article was included if one of the following were met:

a) An article that reports empirical results on code smell detection or analysis or on refactoring decisions
b) An article that reports a tool or method which could be used for code smell detection/analysis
c) An article that reports a tool or method which could be used for refactoring decisions

These initial inclusion criteria were made for the inclusion stage, but when starting to apply it to the preliminary set of articles, it was found to be slightly insufficient. Some topics were borderline cases such as:

- UML-refactorings and related smells
- Refactoring of unit-tests and related smells
- Aspect-oriented refactorings and related smells
- Architectural (high impact) refactorings and smells
- Software visualizations that were made for other reasons but that could still be used for code smells as an supportive method/tool
- Other articles that can be used as a part of decision making or smell detection (like deletion patterns, evolution patterns, OO-ontology)

The phrases "code smell" and refactoring were found to be applied to domains outside the object oriented software domain, where it was coined. This was not taken into consideration when the inclusion criteria were formulated initially, so another criterion was included in order to include other domains that seemed relevant to answer some of the research questions posed in this review. The final resulting criteria are as follows:

a) An article that reports empirical results on code smell detection or analysis or on refactoring decisions
b) An article that reports a tool or method which could be used for code smell detection/analysis
c) An article that reports a tool or method which could be used for refactoring decisions
d) An article that reports the usage of code smells or refactoring in closely related domains

These criteria were applied to the output from the exclusion stage and the result was the primary list of articles. The list of contributions in the extraction sheet was examined by reading the abstract of each contribution to decide whether it met one or more of the inclusion criteria. After screening each contribution and marking those that met the inclusion criteria, all articles not marked for inclusion were removed. The final list of articles is presented in the appendix.

3.4 Data Extraction

The data extraction step consisted of examining closely the contributions and extracting all the relevant data required to answer the research questions. As topics for categorizing the contributions emerged from the screening performed during the inclusion and exclusion stages, these were used as input to the build the categorization schema. The final categorization schema is shown in Figure 5, and is based on several types of contributions: a) empirical, b) methods, c) tools, d) theoretical, and e) summarizing contributions. For each main category of contribution, the information considered to be the most relevant and available was included in the schema in an iterative fashion. In order to extract all the relevant information, abstracts, results and conclusion sections were examined from each of the articles for most of the cases. In other cases, the entire article was examined.

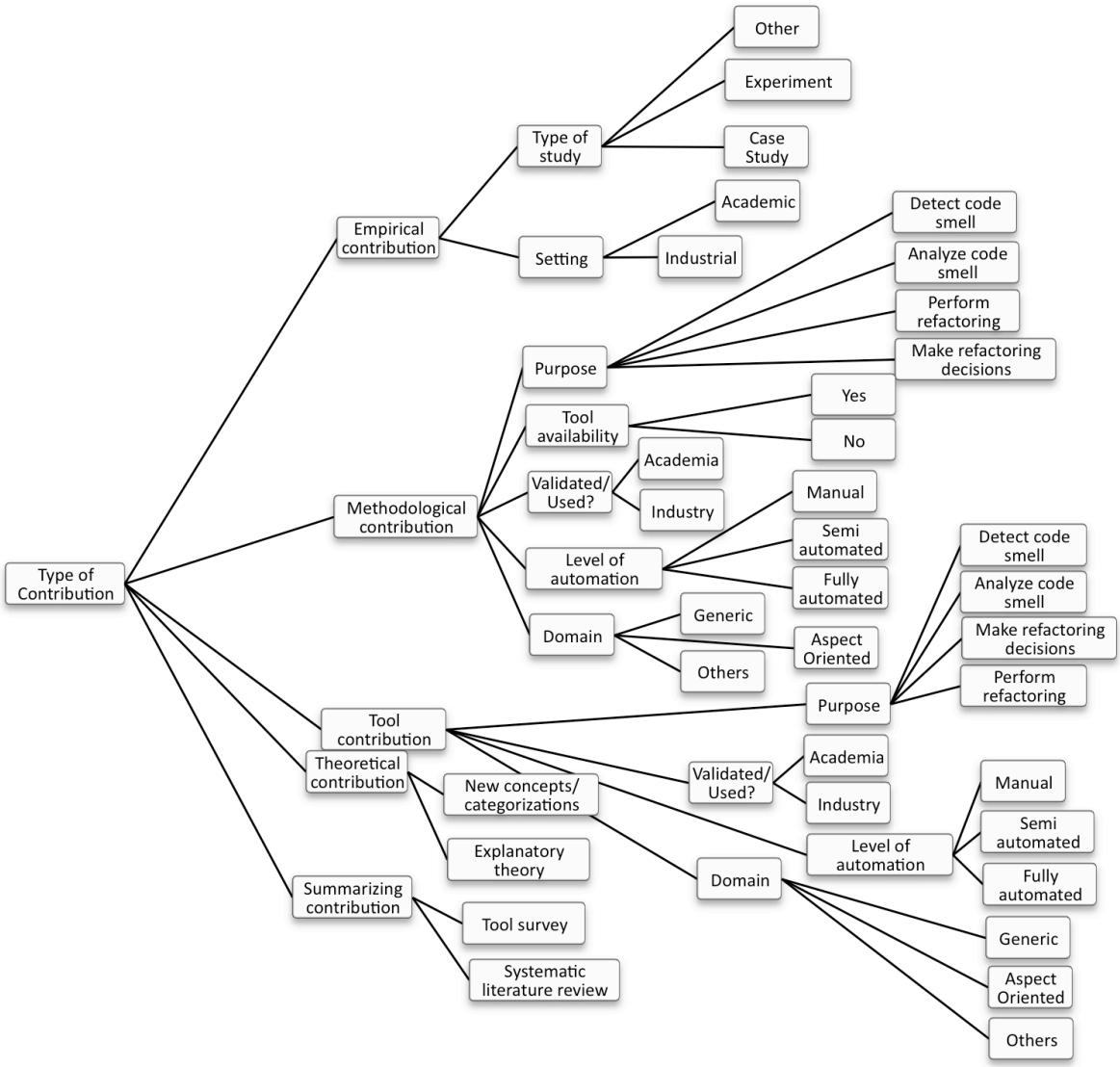


Figure 5: The categorization schema used in the data extraction stage

In the following sections the different categories for contributions included in the schema are explained in detail, as they are of vital importance for the rest of the extraction process. Five main types were defined and all contributions were categorized using these definitions.

a) Empirical contribution. Empirically validated knowledge related to code smells, refactorings or refactoring decisions. Empirically contributions can include case studies, surveys, experiments or other type of studies.

b) Methodological contribution. A proposed way for performing some activity related to refactoring or code smell detection/analysis: For instance counting the number of lines of code in a method in order to identify the *large method* code smell or using certain types of code visualization in order to make refactoring decisions.

c) ***Tool contribution.*** Either as a standalone program or as an extension to an integrated development environment (IDE) for detecting code smells, performing refactorings or support the process of making refactoring decisions (it can be used for one or more of these purposes). Some examples of such tools include: a visualization tool which focuses on visualizing code characteristics useful when making refactoring decisions, an IDE-plug-in for detecting the shotgun surgery code smell or a plug-in that would suggest a refactoring-order when performing several extract method refactorings in a project.

d) Theoretical contribution. Consists of a contribution that alters or adds to the theoretical framework of the field. Proposing new code smells or transferring the paradigms of code smells and refactorings to fields like aspect oriented programming or UML-modeling would be examples of such contributions.

e) Summarizing contribution. Consists of a survey, literature review, or other article where the collection of information itself is a major part of the contribution. An example of this can be a survey of refactoring tools or a literature review on code smells.

After categorizing all contributions into these groups, the extraction for additional relevant data was completed. For each main type of contribution the following information was extracted and added into the extraction sheet:

a) Empirical contribution: Data extracted from empirical contributions relate to the way the empirical data is collected itself. Firstly the type of study conducted was derived (i.e., case

study, experiment, action research or another kind of research). The second kind of data extracted from these contributions consisted of the setting in which the study was conducted in order to assess the external validity of the findings (e.g., were the given code or subjects part of an industrial project or a classroom setting?).

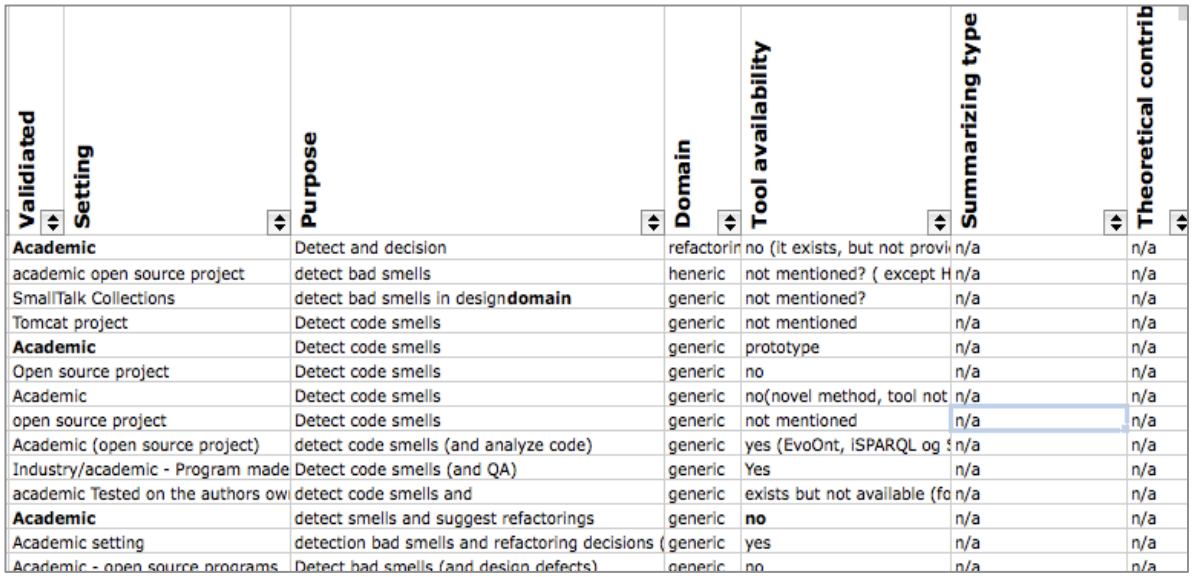
b) Methodological contribution: The first piece of information gathered for the methodological contributions was the purpose of the method. Was it made to detect code smells, perform refactoring or to help make refactoring decisions? The contribution was also examined to determine whether the method was evaluated and in which setting (academically or in an industrial setting). Data on the level of automation for the different methods where collected as well (some methods/tools only present suggestions on different refactorings while others were created to automatically perform refactorings). The domain in which the methodology could be applied was also noted (e.g., some methods were created under the Aspect Oriented Programming paradigm). Although methods in different domains are interesting, the main focus was kept for methods compatible with OO software domain.

c) Tool contribution: The information collected on articles reporting tool contributions were rather similar to the methodological contributions.

d) Theoretical contribution: The theoretical contributions do not share as much relevant data as methods and tools and only type of theoretical contribution was gathered from these types of contributions.

e) Summarizing contribution: These contributions are not as directly linked to the research question and so only the type of summarizing contribution was registered for them.

Figure 6 displays a fragment of the extraction sheet where all the relevant data according to the categorization schema was introduced per each article.



Validiated	Setting	Purpose	Domain	Tool availability	Summarizing type	Theoretical contrib
Academic		Detect and decision	refactorin	no (it exists, but not provi	n/a	n/a
academic open source project		detect bad smells	heneric	not mentioned? (except H	n/a	n/a
SmallTalk Collections		detect bad smells in design**domain**	generic	not mentioned?	n/a	n/a
Tomcat project		Detect code smells	generic	not mentioned	n/a	n/a
Academic		Detect code smells	generic	prototype	n/a	n/a
Open source project		Detect code smells	generic	no	n/a	n/a
Academic		Detect code smells	generic	no(novel method, tool not	n/a	n/a
open source project		Detect code smells	generic	not mentioned	n/a	n/a
Academic (open source project)		detect code smells (and analyze code)	generic	yes (EvoOnt, iSPARQL og S	n/a	n/a
Industry/academic - Program made		Detect code smells (and QA)	generic	Yes	n/a	n/a
academic Tested on the authors ow		detect code smells and	generic	exists but not available (fo	n/a	n/a
Academic		detect smells and suggest refactorings	generic	**no**	n/a	n/a
Academic setting		detection bad smells and refactoring decisions (	generic	yes	n/a	n/a
Academic - open source programs		Detect bad smells (and design defects)	generic	no	n/a	n/a

Figure 6: Screenshot of the data extraction sheet. It shows some of the data gathered and how it was organized.

3.5 Data Analysis

This section describes how the results from the extracted data material were analyzed. The data used for this part included the extraction sheet and PDF-files of the contributions. To be able to answer the main research question, it is required to answer the three sub-questions formulated at the beginning of the thesis. Sub-questions **SQ1** and **SQ2** focus on the empirical evidence to understand the effects of code smells and refactorings respectively, thus all the empirical contributions were examined and summarized. The empirical work was then grouped according to a set of topics that emerged during reading process. These topics emerged from informal coding done on the content of the articles, which consequently were sorted following a somewhat similar approach to grounded theory [21]. The topics were created based on the focus of the contributions and how they related to the research questions. Research sub-question **SQ3** relates to method and tool contributions in a similar way and thus these contributions were also summarized and grouped based on the purpose of the method or tool (which was already available in the data extraction sheet). A meta-analysis on the resulting primary list of articles was performed through descriptive statistics (in the form of charts, summarizing tables and diagrams). The focus topics of the contributions and the data gathered in the extraction sheet were used as the main data sources to generate the graphs and tables.

4 Results

This section reports the findings from the literature review. Figure **7** displays the output from the different stages of the review. The initial queries yielded a total of 134 contributions. This did, however, comprise both duplicates and irrelevant material. After applying the exclusion and inclusion criteria, only 46 articles remained.

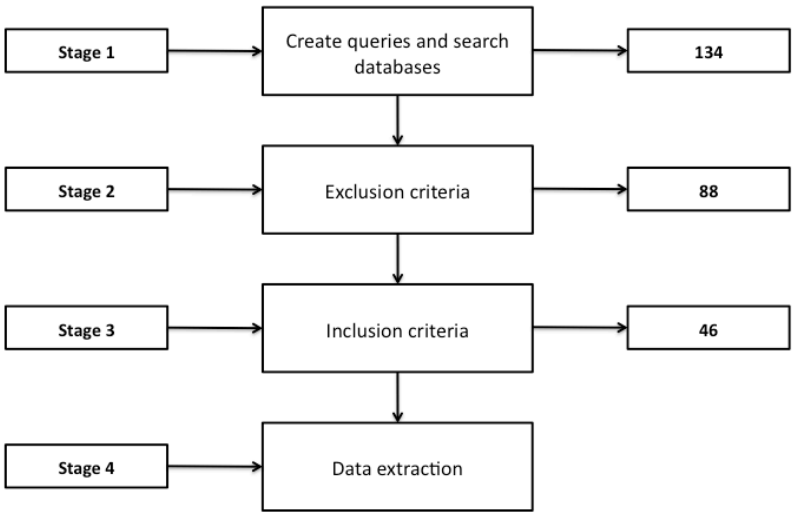


Figure 7:Output from the previously defined stages.

4.1 Overview of the Studies

Table 3 presents the distribution of the contributions according to the categorization previously described in the methodology section. The majority of the contributions (61%) are design research. This includes both methods and tools – which is a difference that only occasionally will cause them to be separated when I discuss these contributions in the remainder of this thesis. The difference is mainly whether the method is implemented as a tool or not, and this difference is often not important. Detailed results on the design contributions are presented in Section 4.3. The second most represented type of contribution is empirical research (24%). These contributions are further grouped and summarized in Section 4.2. Summarizing and theoretical research were the least represented contribution types with only a total of 7 contributions (15%). These types of contributions are not directly relevant for any of the research questions but could obviously provide input to answer the main research question on the *state of the art* in SE research, and are both summarized in Section 4.4.

Table 3: Classification according to type of contribution

Type of research	Contribution	Number	Percent
Empirical research	-	11	24%
Design research	Method	13	28.5%
	Tool	6	13 %
	Method + tool	9	19.5%
	Subtotal	28	61%
Summarizing research	-	3	6.5%
Theoretical research	-	4	8.5%
Total		46	100%

Figure **8** shows the number of contributions published each year starting from 2000. The contributions within this research filed were relatively scarce until 2004. The number of contributions from 2009 does not include the last few months because the data was collected in the autumn 2009.

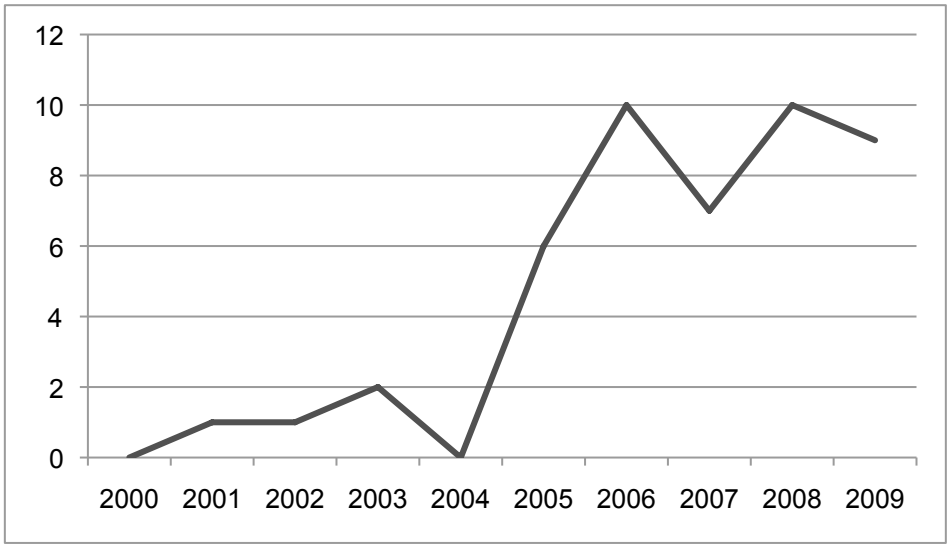


Figure 8: Number of relevant contributions published each year starting from 2000

The remaining subsections will summarize the articles according to the categories in which each of them were grouped into. The articles are numbered and for the remainder of the thesis, they will be referred to as S1-S46. The complete list of primary articles and authors is included in Appendix A.

4.2 Empirical Contributions

The list of articles reporting empirical studies on code smells and refactoring is shown in Table 4 together with their corresponding topics. The topics for the different contributions in the empirical field were not given when the work started. The primary goal for categorizing the empirical contributions into different topics is to describe them in a structured and understandable way, which may enable to answer the main research question: *What is the state of art in SE research to support analysis and detection of code smells and refactoring decisions?* The focus on the different topics in the result section is derived from the topics identified within the contributions. This is opposed to having a predefined categorization and forcing them upon the data, which may result in a less descriptive and understandable reporting of results.

Table 4: List of empirical contributions and topics

Nr.	Name of contribution	Category
S35	*Is the Need to Follow Chains a Possible Deterrent to Certain Refactorings and an Inducement to Others?*	*Evidence – help refactoring decisions*
S36	*Size and Frequency of Class Change from a Refactoring Perspective*, in *Software Evolvability*	*Evidence – help refactoring decisions*
S37	*The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph*,	*Evidence – help refactoring decisions*
S38	*Common Refactorings, a Dependency Graph and Some Code Smells: an Empirical Study of Java OSS*	*Evidence – help refactoring decisions*
S3	*Leveraging Code Smell Detection With Inter-Smell Relations*	*Evidence – help identify code smells*
S33	*Relation of Code Clones and Change Couplings*	*Evidence – help identify code smells*
S40	*Impact of Metrics Based Refactoring on the Software Quality: a Case Study*	*Impact of refactorings*
S45	*An Empirical Study of the bad smells and class error probability in the post-release object-oriented system evolution*	*Effects of code smells*
S14	*Code Smell Eradication and Associated Refactoring*	*Effects of code smells*
S23	*An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement*	*Subjective evaluation*
S24	*Subjective evaluation of software evolvability using code smells: An empirical study*	*Subjective evaluation*
S41	*Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study*	*Subjective evaluation*

The empirical contributions were summarized and studied to extract the main claims, purpose and results. These articles were especially relevant to the research sub-questions SQ1 and SQ2 and as such were given additional attention. The sub-categories of the empirical contributions were created after the summary of the articles was finished and were based on the stated purpose and results of the individual contributions. Figure **9** shows the distribution

of the contributions. The rest of this sub-chapter will present the emerged topics of the empirical contributions and how they relate to the research questions, as well as a short summary of each article.

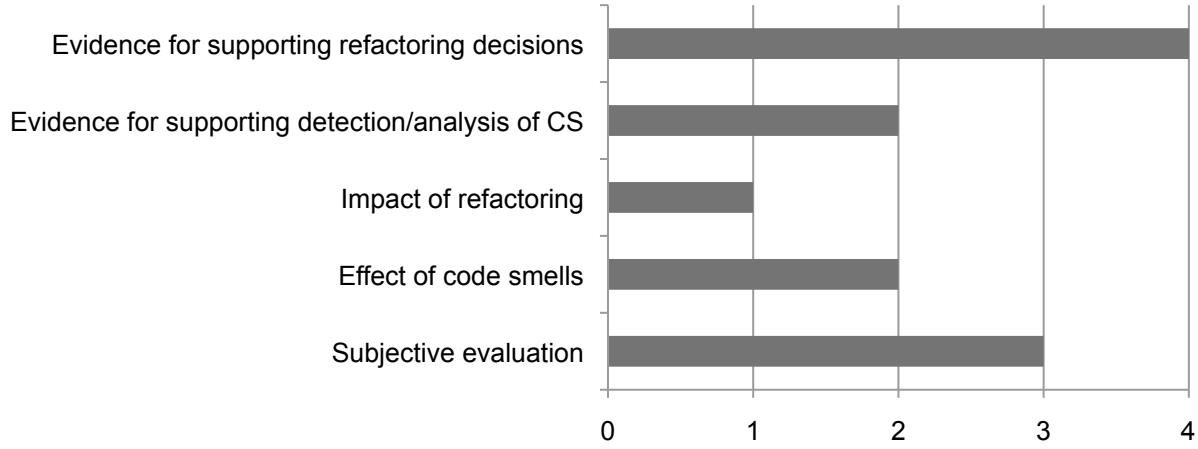


Figure 9: Distribution of the topics for all empirical contributions

4.2.1 Evidence for Supporting Refactoring Decisions

Empirical results that might help practitioners making refactoring decisions are obviously interesting from several points of view. Practitioners that are to make such decisions and researchers that are working on refactoring methods and tools would possibly be interested in such research. These are contributions that do not look at how refactoring directly affect the code. They do, however, investigate various other implications of refactoring as cost ([S38] and [S35]), impact on tests [S37]), and when or under which situations refactoring tend to occur [S36]).

Common Refactorings, a Dependency Graph and Some Code Smells: An Empirical Study of Java OSS

Counsell et al. [S38] report results from a tool whose purpose was to identify and extract refactorings from seven open-source Java systems to be able to get data on how frequent they were used. They chose to include 15 refactorings thought to be the most commonly employed and most interesting. Counsell et al., coined a "Gang of Six", which represented the six most common refactorings (in ascending order of frequency): Pull Up Method, Move Method, Add Parameter, Move Field, Rename Method, and Rename Field. Most of these refactorings had a high in-degree and low out-degree impact. They also found that inheritance and encapsulations-based refactorings were applied relatively infrequently.

Is the Need to Follow Chains a Possible Deterrent to Certain Refactorings and an Inducement to Others?

In [S35], Counsell et al. investigate how refactorings relate to each other. They analyzed 14 refactorings and a dependency graph was created based on description of code refactorings in Fowler's book [2]. A dependency graph is a directed graph that represents the dependencies between objects. Arrows mark objects that are dependent on the object to which the arrow points. Figure 10 shows an example of a dependency graph. A depends on B and C, and B also depends on D, making A depends on B, C and D.

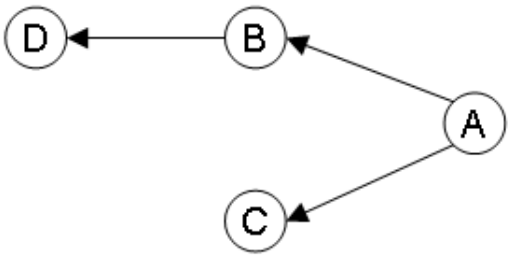


Figure 10: Example of a dependency graph

The refactoring dependency graph was used as a basis for analyzing 7 open source systems (OSS). The head version of each system was analyzed in addition to several previous versions. They found that refactorings inducing long chains tended to be utilized less by developers than refactorings with short chains. *Encapsulate Downcast* (0 occurrences), *Extract Subclass* (6 occurrences) and *Extract Superclass* (23 occurrences) did not occur often. *Hide method* had also few (9) occurrences despite of being a relatively simple refactoring. Results also suggest that the complexity of long refactoring chains may be a real consideration prior to refactoring.

Size and Frequency of Class Change from a Refactoring Perspective

In [S36], Counsell and Mendes investigate whether the number of changes to a class is a sufficiently good indicator that the class needs refactoring. The data material consisted of code changes from 161 Java classes from the Gnu GCC libjava library. The baseline for this research was the assumption that the number of changes to classes is a good indicator that a class might need refactoring. Counsell and Mendes claimed that combining this aspect with the number of lines added, would improve this approach. They concluded that an increase in *Lines of Code* is a better indicator than number of changes. They also investigated the claim that large classes are more change-prone than smaller classes, but found only limited support for this.

The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph

Counsell et al. [S37] investigate how refactorings affect the repeatability of tests. They identified four categories of refactorings, ranging from semantic-preserving to incompatible. These groups were applied to the empirical data from the 7 OSS previously mentioned in S37. The most relevant result shows that 4 out of the 5 most used refactorings all belong to the same group – refactorings that change the old interface, but can be made compatible by adding the old interface as a wrapper. They also conclude that semantic-preserving refactorings can have hidden ramifications despite their advantages. Counsell et al. postulate that the choice of refactorings must take into consideration the inter-relatedness of refactorings. Developers should not pick refactorings based on superficial characteristics, but look into the in-depth mechanics of the different refactorings.

4.2.2 Evidence for Supporting Detection/Analysis of Code Smells

Two contributions present empirical knowledge that could support the process of detecting and analyzing code smells. While SQ3 focuses on actual tools and methods to help with detecting and analyzing code smells, results from empirical studies could be used in order to assist in the creation of such tools and methods.

Leveraging Code Smell Detection With Inter-Smell Relations

Pietrzak et al. [S3] present different viewpoints on how code smells affect each other. Instead of looking into the resolution order, they investigate how already detected and rejected smells can be used as a factor (in addition to already utilized sources as metrics, code behavior or changes in code) for detecting new smells. Six inter-smell relations are identified as useful for smell detection. This theory is supported with empirical evidence from an experiment performed on classes from the Apache Tomcat codebase [22]. One example of such a relation is *Data Class* and *Feature Envy.* Of the 26 *Data Classes* found, 24 of them were referenced by methods identified as *Feature Envious*.

Relation of Code Clones and Change Couplings

Geiger et al. [S33] use the concept of *Change Couplings*, which is defined as "*files which are committed at the same time, by the same author, and with the same modification*", and the more familiar *Code Clones*. It is, however, reasonable to assume that they are related to Fowler's concepts of *Shotgun Surgery and Duplicated Code*. Geiger et al. examine the relation between these smells and try to validate and quantify this relation. The data is drawn from the Mozilla project [23]. Regression analysis was applied on the clone coverage and

coupling coverage data. They did, however, find that the correlation is too complex to be expressed easily and that the judgment of the software engineer is still needed. In addition to these results, they presented a framework to examine this relationship further on, which consists of a set of metrics and visualization techniques to spot where the correlation between cloning and change couplings exists.

4.2.3 Effects of Refactorings

Only one empirical contribution was identified that investigates the direct effect refactorings have on software quality.

Impact of Metrics Based Refactoring on the Software Quality: A Case Study

Shrivastava and Shrivastava [S40] report a case study in which an inventory application was considered and efforts were made to improve the quality of the system by refactoring. Code metrics were used before and after sets of refactorings to describe the impact. The code in question was from the open source application Inventor Deluxe and the Eclipse plug-in Metrics 1.3.6 was used to assess the code. The following metrics were used to measure quality: Number Of Attributes in class (NOA), Number Of Classes (NOC), Number of Methods in class (NOM), Depth of Inheritance Tree (DIT), Cyclomatic Complexity (CC) and Total Lines of Code in class (TLOC). The refactorings were applied sequentially (Extract Class, Extract Method and Extract Subclass). Average NOA, NOM, and CC were reduced throughout the refactorings while NOC, TLOC and DIT increased. They concluded that refactoring was found to have a positive impact on the software quality.

4.2.4 Effects of Code Smells

Two contributions that focus on the effects of code smells were identified, but they have different perspectives. Li and Shatnawi [S45] investigate how the presence of code smells affects error rate while Hamza et al. [S14] investigate the effort required to remove code smells from code. They are both interesting in terms of answering SQ1: *What is the state of art in SE research with respect to investigating empirically the effects of code smells in development and maintenance projects?*

An Empirical Study of the Bad Smells and Class Error Probability in The Post-Release Object-Oriented System Evolution

In [S45], Li and Shatnawi present results from an empirical study that investigates the relationship between six code smells (*Data Class, God Class, God Method, Refused Bequest, Shotgun Surgery* and *Feature Envy*), and class error probability in an industrial-strength

system. The code base studied was the Eclipse project. In addition to data extracted from the code base, relevant bugs and errors were extracted from Bugzilla [24] and divided into three error-severity levels. Code smells were detected in the code using Borland Together and the connection between code smells and errors were investigated. Multivariate Logistic Regression and Multinomial Multivariate Logistic Regression were used to study the association between code smells, error proneness and error severity. The results showed a (significant) positive linkage between the Shotgun Surgery, God Class and God Method code smells and class error probability. They also suggest that refactoring may reduce the chance that a class will have errors in the future.

Code Smell Eradication and Associated Refactoring

Hamza et al. [S14] look at the dependencies between the refactorings of Kerievsky [11] and Fowler [2] in the context of Fowler's 22 code smells. This was done to analyze the difference in effort required for each code smell in order to eradicate them. Extract Class (required to eradicate 6 code smells). Move Method (6 code smells), Extract Method (4 code smells), and Move Field (4 code smells) are the refactorings used to remedy the most smells. Only 5 code smells did not require one of these four refactorings. The results also suggest several code smells would be relatively expensive to eradicate and that Fowler's code smells are less complex to eradicate and induces fewer refactorings on average, compared to those of Kerievsky. This because Kerievsky's code smells often induces a set of design pattern refactorings as well as relatively large numbers of refactorings. Primitive Obsession can be seen as the most complex code smell because it induces a total of 200 Fowler refactorings. Large Class and Duplicated Code each induce a total of 163 refactorings.

4.2.5 Subjective Evaluation

The three articles on subjective evaluations all share the focus on how the subjective nature of code smells may affect the results of code evaluations. The two contributions by Mäntylä ([S23] and [S24]) both target code smells specifically, whiles Counsell et al. [S41], target the code attribute of cohesion – which in turn is commonly thought to signal that a code might need refactoring – much like a code smell [25].

An Experiment on Subjective Evolvability Evaluation of Object-Oriented Software: Explaining Factors and Interrater Agreement

Mäntylä [S23] reports two experiments on software evolvability evaluations where agreement of evaluators was studied. The participants were 88 MSc students and the code analyzed consisted of 1000LOC of Java code. The participants were asked to answer whether certain

code smells (Long method, Long parameter list or Feature Envy) existed in the code and whether it should be refactored or not. In experiment 2, participants were simply asked whether the code should be refactored or not. The results show that the interrater agreement was high for simple code smells, but low for refactoring decisions. Demographics and source code metrics were analyzed to account for the different evaluations. Code metrics could explain over 70% of the variations regarding simple code smells, but only about 30% for the refactoring decisions. Demographics did not seem to be useful predictors, neither for evaluating code smells nor refactoring decisions. Mäntylä states that the low agreement for the refactoring decisions may indicate difficulty in building tool support simulating real-life subjective refactoring decisions. He adds that code metric tools, however, should be effective in highlighting straightforward problems, as simple code smells.

Object-Oriented Cohesion Subjectivity amongst Experienced and Novice Developers: An Empirical Study

Counsell et al. report results on how software engineers rates cohesion in [S41]. They had a group of twenty-four subjects from IT-experienced and novice groups and asked them to rate ten classes sampled from two industrial systems in terms of their overall cohesiveness. The subjects were presented the classes in random order and asked to rate them on a scale of 1 – 10 how cohesive they thought that class was. They were also asked to give some comments on why they had given the various cohesion ratings. The time frame for this task was 15 minutes. The cohesive values were then grouped by experience level and by metrics as Number of Method in Class, Number of Associations (defined as the number of *unique* classes to which the class under consideration is coupled), Coupling Between Objects and Number of Comment Lines. The results suggest that class size (by number of methods) only influenced the perception of cohesion by novice subjects. Well-commented classes were rated more cohesive amongst IT experienced than novice subjects. Thirdly, results suggest strongly that cohesion comprises a combination of various class factors including low coupling, small number of attributes and well-commented methods, rather than any single, individual class feature *per se*.

Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study

Mäntylä et al. [S24] report the result of an empirical study on the subjective evaluation of code smells. They propose to use the word *software evolvability* to describe the ease of further developing a piece of software. Furthermore, they elaborate thoroughly the differences between human evaluations and program analysis based on metrics. The empirical evidence

presented is drawn from a case study performed in a Finnish software product company. The authors asked 12 participants for the degree of presence of 23 code smells in 16KLOC to 83KLOC-sized modules. The same code was also analyzed programmatically. Results show that lead developers reported more Parallel Inheritance Hierarchies, while regular developers reported more Duplicated Code. Developers with less experience with the modules reported more code smells than developers more familiar with the modules. Yet another result shows that Large Class, Long Parameter List and Duplicated Code finds conflicted with the program analysis. The authors conclude that organizations should combine subjective evaluations and metrics when making decisions on improving the code. The results also suggest that experienced developers are better at detecting more advanced code smells.

4.3 Design Research Contributions

As previously stated, design contributions add up to the majority of the overall contributions. The design contributions are here divided in three main categories according to their purpose, namely: *detecting code smells, performing refactorings* and *refactoring decisions*. Figure 11 shows the total number of contributions for each category. In this figure, contributions that have several purposes, will count towards the total for each of its purposes. Of the 28 design research contributions, only 10 of them could show to an *available* tool. Some tool contributions were using novel or prototype-tools, while other contributions did not state whether the tool presented was available or not. Another trend is that very few contributions seem to have been tested or validated in an industrial setting. Only 4 of the 28 design contributions were industrially validated. These few contributions also includes a tool contribution which were academic, but were going to be released commercially and had been tested for some time in this context [S7] and another tool contribution that was validated by one independent designer [S30]. The rest of the 28 design contributions were not validated at all, validated on example code from books, or validated by the researchers on code taken from various open source projects.

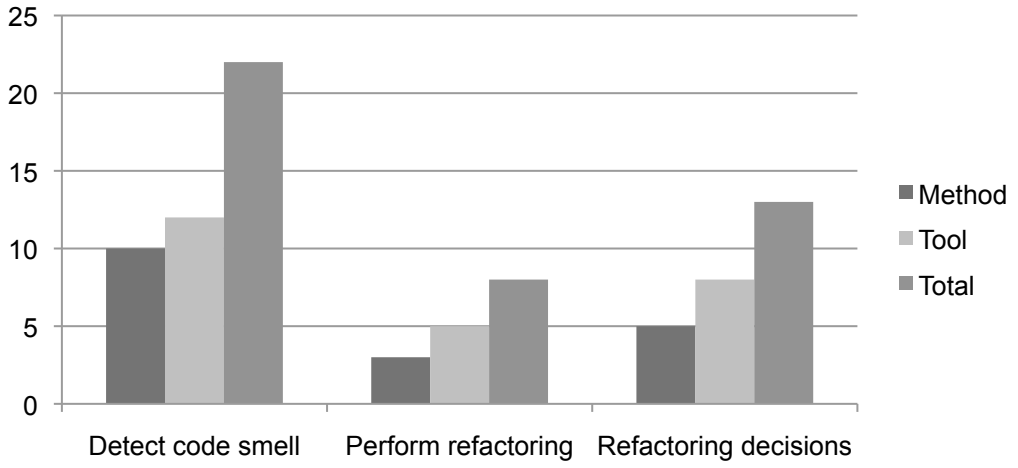


Figure 11: Summary of design research contributions by purpose

4.3.1 Detecting And Analyzing Code Smells

A total of 22 design contributions to fully or partially focus on detecting code smells were identified. Some methods and tools have code smell detection as their only focus, while other contributions both detect smells and try to remove them. This section includes every contribution related to detecting code smells and categorizes them into 7 groups: (1) Visualization tools, (2) Metrics, (3) Specification of code smells, (4) Various detection methods, (5) Pinpoint methods/tools (6) Early code smell detection tools and (7) Generic defect detection. In Table 5, the list of contributions is presented, where *Type* denotes the type of design contribution where T = tool, M = method and M/T = a combination of tool and method.

Table 5: List of contributions for detecting/analyzing code smells

Nr.	Name of contribution	Approach	Type
S6	*A Catalogue of Lightweight Visualizations to Support Code Smell Inspection*	*Visualization*	*T*
S20	*Comprehensive Software Understanding with SEXTANT*	*Visualization*	*T*
S22	*Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code*	*Metrics*	*M/T*
S11	*Metrics Based Refactoring*	*Metrics*	*M/T*
S18	*Dependency Oriented Complexity Metrics to Detect Rippling Related Design Defects*	*Metrics*	*M*
S26	*A Domain Analysis to Specify Design Defects and Generate Detection Algorithms*	*Specification*	*M*
S27	*Automatic Generation of Detection Algorithms for Design Defects*	*Specification*	*M/T*
S28	*DECOR: A Method for the Specification and Detection of Code and Design Smells*	*Specification*	*M/T*
S43	*Identifying Refactoring Opportunities Using Logic Meta Programming*	*Various detection*	*M*

S1	*Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix*	*Various detection*	*M*
S3	*Leveraging Code Smell Detection with Inter-Smell Relations*	*Various detection*	*M*
S13	*Discovering Unanticipated Dependency Schemas in Class Hierarchies*	*Various detection*	*M*
S4	*Multi-Criteria Detection of Bad Smells in Code with UTA Method*	*Various detection*	*M*
S21	*JDeodorant: Identification and Removal of Feature Envy Bad Smells*	*Pinpoint*	*T*
S31	*JDeodorant: Identification and Removal of Type-Checking Bad Smells*	*Pinpoint*	*T*
S39	*Source Code Enhancement Using Reduction Of Duplicated Code*	*Pinpoint*	*M*
S44	*Using Concept Analysis to Detect Co-Change Patterns*	*Pinpoint*	*M*
S7	*Java Quality Assurance by Detecting Code Smells*	*Early code smell detection*	*M/T*
S4	*Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring*	*Early code smell detection*	T
S16	*A Flexible Framework for Quality Assurance of Software Artefacts with Applications to Java, UML, and TTCN-3 Test Specifications*	*Generic defect detection*	*M/T*
S5	*Mining Software Repositories with iSPARQL and a Software Evolution Ontology*	*Generic defect detection*	*M/T*

Visualization tools: Parnin et al. [S6] propose specific visualizations that would benefit the process of inspecting code – both in terms of assessing code in general and for the detection of code smells. Eichberg et al. [S20] on the other hand use visualizations as one of several comprehension techniques and argues that a combination of this technique plus several other techniques is required to successfully understand software systems.

Metrics: Munro [S22], Simon et al. [S11] and Reddy and Rao [S18] use existing and new metrics to identify code smells and opportunities for refactoring. They are similar in their approaches since they rely on various metrics to detect code smells, but differ in which types of smells they focus on and how they use metrics to detect them. The study reported in [S22] is from 2001 and the earliest contribution in this review. It focuses on 4 refactorings: *Move Method, Move Attribute, Extract Class* and *Inline Class,* which are all based on use relations. They calculate the distance between methods and classes by investigating the usage of attributes and methods. Simon et al. in [S11] investigate 2 smells, namely *Lazy Class* and *Temporary Field.* These smells are then characterized and described more informally and interpreted in terms of logic and metrics, as can be seen in Figure 12 (NOM = Number Of Methods, LOC = Lines Of Code, WMC = Weighted Methods per Class, CBO = Coupling Between Objects. DIT = Depth of Inheritance Tree, IVMC = number of methods that reference each instance variable defined in a class).

Interpretation of *Lazy Class*

if
$NOM = 0$
then PRINT YES
else if
$(LOC < LOC_{Median}) AND (\frac{WMC}{NOM} <= 2)$
then PRINT YES
else if
$(CBO < CBO_{Median}) AND (DIT > 1)$
then PRINT YES
else
PRINT NO.

Interpretation of *Temporary Field*

if
$(IVMC <= 1)$
then PRINT YES
else
PRINT NO

Figure 12: Formal interpretation of the code smells *Lazy Class* and *Temporary Field* [S11].

Reddy and Rao [S18] introduce new metrics to describe changes over time, in order to detect *Shotgun Surgery* and *Divergent Change.* This contribution is different from [S22] and [S11] because it does not only look at one specific state of the code. *Shotgun Surgery* and *Divergent Change* both need data on changes over time to be detected. The metrics introduced in this contribution by Reddy and Rao are tailored for this purpose – to describe dependencies between objects over time.

Specification of code smells: The studies reported by Moha et al. [S26, S27, S28], all address the issue of lack of formal descriptions of code smells. They work towards the goal of a unified way to formally describe them. They demonstrate how software engineers can specify code smells at a high-level of abstraction using a domain-specific language for automatically generating detection algorithms based on these specifications.

Various detection methods: [S43], [S1], [S3], [S13], and [S4] explore different techniques to detect code smells. Using code metrics to detect code smells has been one of the major approaches for refactoring as seen in [S11], [S22], [S18]. The contributions presented here all use different approaches for detecting code smells. One example is the approach by Pietrzak and Walter [S3], which uses data from previously deleted and rejected code smells to help detect other code smells. They also document how code smells relate to each other in other situations. Arévalo et al. [S13] use concept analysis to detect hidden dependencies in class hierarchies. They focus on detecting *bad smells* in design, but do not refer to any of the code smells defined in Fowler [2]. Reddy and Rao [S1] investigate an approach called Change

Propagation Probability Matrix, as a technique to detect the *Shotgun Surgery* and *Divergent Change* code smells. This technique investigates how change propagates between objects – making it suit the requirements of a tool to detect these dynamic smells. Tourwé and Mens [S43] report preliminary but promising results on using logic meta-programming to detect smells. This is one way of performing logical computation on code in another language. They propose the SOUL meta-language and show how the *Obsolete Parameter* and *Inappropriate Interface* smells can be detected using this technique. Walter and Pietrzak [S4] try to combine several approaches by using several data sources to detect code smells. They identify the following six distinct sources useful for smell detection:

a) Programmer's intuition and experience
b) Metrics values
c) Analysis of a source code syntax tree
d) History of changes made in code
e) Dynamic behavior of code
f) Existence of other smells

The programmers subjective intuition and metrics values are probably the most used sources as of now. But at least 5 of the 6 sources are represented in contributions present in this review. Walter and Pietrzak describe *dynamic behavior of code* as smells that are hard to detect. The authors claim that these smells require the code to run to be detected. They suggest writing unit tests to better be able to detect these smells. They also mention *Data Class* as an example of a code smell of this type. *Data Class* should, however, be a relatively easy code smell to detect [S3, S22]. The other 5 sources for detecting code smells are represented in this review. Walter and Pietrzak evaluate their approach by verifying that they are able to detect instances of *Large Class* in an open source system by using several sources as input for the detection algorithm.

Pinpoint methods/tools: Tsantalis et al. [S21], Fokaefs et al. [S31], Nasehi et al. [S39], and Gîrba et al. [S44] are all relatively new design contributions (all published in 2007 or 2008) that target one specific code smell or design problem. This can be described as a depth-first approach on refactoring research. As previously showed, several other contributions investigate techniques for detecting several code smells with more general approaches. By narrowing the scope to one specific code smell, the contributions in this category are able to deliver tool support in a much greater extent than the average contribution. The two tools described in [S21] and [S31] are released and available as plug-ins to the popular Eclipse

IDE, while the studies reported in [S39] and [S44] investigate troublesome aspects of programming – duplication of code and the closely related topic of co-change (parts of code that change at the same time), respectively. The main contributions from these articles will obviously have to be the specific *method* to remove the chosen code smell, because using a new tool for each code smell is not very efficient.

Early code smell detection tools: Van Emden and Moonen [S7] give an early contribution (from 2002), a code smell detection tool in Java. Another early article [S42] (from 2003) suggest and demonstrate that refactoring tools should use code smells to suggest refactorings. These two contributions are historically interesting because they present early efforts to use code smells in refactoring tools. They are, however, not very relevant for developers of today except for as background information. The approaches can be seen as *proof of concept* tools, concepts that already are further developed into available and refined tools.

Generic defect detection: Nodler et al. [S16] present a framework for analyzing code, presenting metrics and detecting code smells. It is, however, not only applicable to software code, but also to other languages like UML-diagrams and the TTCN-3 test specification. Kiefer et al. [S5] present several tools and frameworks for mining software repositories. This could quite possibly be used to improve code smell detections, but the code smell aspect is not evaluated in the contribution.

4.3.2 Performing refactoring

Tools and methods that perform the actual refactoring are the least represented ones with only 6 contributions. Nevertheless, this category is interesting because 5 of the 6 contributions have available tools. [S30], [S29], [S21] and [S31] by Tsantalis et al. have present tool support in the JDeodorant plug-in for Eclipse. Steimann et al. [S12] also present tool support in an Eclipse plug-in and present a new refactoring technique to help with creating small context-specific interfaces to lower the coupling between classes. The study reported by Pérez and Crespo [S17] both consists of a survey on existing tools and a presentation of a method for supporting behavior-preserving refactoring.

Table 6: List of design research contributions for performing refactoring

Nr.	Name of contribution	Type of contribution
S30	*Identification of Extract Method Refactoring Opportunities*	*T*
S29	*Identification of Move Method Refactoring Opportunities*	*T*
S21	*JDeodorant: Identification and Removal of Feature Envy Bad Smells*	*T*

S31	*JDeodorant: Identification and Removal of Type-Checking Bad Smells*	*T*
S12	*Decoupling classes with inferred interfaces*	*T*
S17	*Perspectives on automated correction of bad smells*	*M*

4.3.3 Making refactoring decisions

It was found through the review that design contributions for supporting refactoring decisions follow very different strategies. The approach used in [S21] and [S29] calculates a "score" for each refactoring by summing up relevant metrics and presents a list of proposed refactorings rated by this score. This score is meant to rate the state of the code *after* each suggested refactoring. The tool presented in [S30] uses the same approach, but as demonstrated by the researchers, the resulting order does not correlate well with a developer's opinion. Trifu and Reupkes [S2] do not only recommend refactorings. They use a combination of several *symptoms* to identify targets for refactoring. They state that "*by conception, design flaws can be put in a one-to-one relation with semantically meaningful refactoring solutions"* [S2].

Table 7: List of design research contributions to help with making refactoring decisions

Nr.	Name of contribution
S21	*JDeodorant: Identification and Removal of Feature Envy Bad Smells*
S29	*Identification of Move Method Refactoring Opportunities*
S30	*Identification of Extract Method Refactoring Opportunities*
S2	*Towards Automated Restructuring of Object Oriented Systems*
S20	*Comprehensive software understanding with SEXTANT*
S16	*A Flexible Framework for Quality Assurance of Software Artefacts with Applications to Java, UML, and TTCN-3 Test Specifications*
S46	*Code Evaluation Using Fuzzy Logic*
S15	*Facilitating Software Refactoring With Appropriate Resolution Order of Bad Smells*
S8	*Representing Refactoring Opportunities*

The tools presented by Eichberg et al. [S20], Nodler et al. [S16] and Avdagic et al. [S46] support refactoring decision making since their contribution attempts to analyze and visualize the code to give a broader understanding of its properties. Code smells are vaguely described and thus it can be fruitful to be able to look at many different aspects of the code through analyzing and visualizing it. Liu et al. [S15] investigate the relationship between different bad smells and propose a resolution order based on this. They demonstrate that resolving a set of code smells could have different results, depending on the resolution order. Piveta et al. [S8] propose a unified way of describing conditions of which applications of refactoring can be advantageous.

4.4 Summarizing and Theoretical Research Findings

In addition to the empirical research and design contributions, there are three summarizing and four theoretical contributions amongst the primary articles. They are not relevant to any of the specific sub research questions, but are included to provide input for the main research question on *state of the art* in the research.

4.4.1 Summarizing Contributions

The three summarizing contributions stem from language specific tool surveys to a historical review on code smells. Lerner [S34] presents "metric_fu", which is a tool suite for analyzing code in the Ruby programming language. The contribution is summarizing the different tools present in the suite. [S19] is a survey on java refactoring tools for the big IDEs, namely: Eclipse (version 3.1) IntelliJ IDEA (version 4.5) and Netbeans (version 4.0); and focuses on refactoring as a way of developing, not only maintaining code. The results from this survey show that Netbeans severely lacks refactoring abilities compared to the other IDEs, such as Eclipse and IntelliJ, which both support around 30 refactorings. It is, however, noted that IntelliJ 5.0 is a better refactoring tool because it includes code analysis that can be used to detect code smells. The conclusion of the survey is that there are very good tools that could help developers to perform refactorings rather painlessly

Table 8: List of summarizing contributions

Nr.	Name of contribution
S34	*At the Forge: Checking Your Ruby Code With Metric_Fu*
S19	*Refactoring Tools and Complementary Techniques*
S17	*Perspectives on Automated Correction of Bad Smells*

Pérez and Crespo [S17] present a historical review on the field of refactoring and code smells. They summarize the origin of code smells and refactorings and investigate some of the most prominent contributions in the field. The article also presents a short section on *Evolution of research on bad smells' related activities*. The material studied in this contribution seems to be exclusively based on books and PhD theses. The bottom line of the article is that one can separate the work on code smells and refactoring in three parts as illustrated in Figure 13. The bad smell catalogs and specifications have corresponding work in refactoring research, while the effort to automate smell detection has yet to be mirrored by research on refactoring.

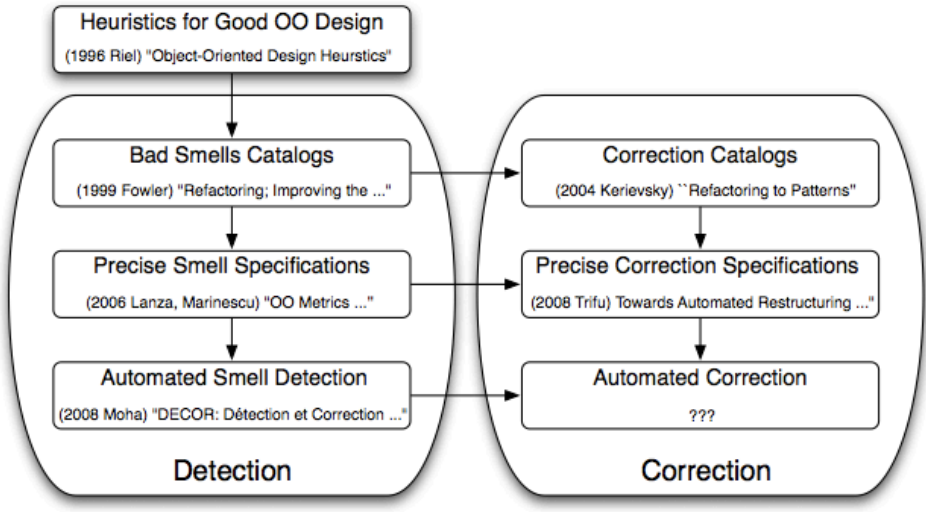


Figure 13: History of research on code smells and refactoring as presented by Pérez and Crespo [S17].

4.4.2 Theoretical Contributions

Murphy-Hill [S9] and Murphy-Hill et al. [S10] discuss properties of code smell detection tools. The focus of these contributions is a top-down approach where attributes of the tool, and not implementation details, are discussed. The contributions are very similar to each other, and since [S10] resulted the most recent and elaborated contribution, this is was the article investigated further in the review.

Table 9: List of theoretical contributions

Nr.	Name of contribution
S10	*Seven Habits of a Highly Effective Smell Detector*
S9	*Scalable, Expressive, and Context-Sensitive Code Smell Display*
S25	*Towards a Catalog of Aspect-Oriented Refactorings*
S15	*Facilitating Software Refactoring With Appropriate Resolution Order of Bad Smells*

Murphy-Hill et al. [S10] postulate 7 habits that code smell detection tools should pay extra attention to:

Availability – *Rather than forcing the programmer to frequently go through a series of steps in order to see if a tool finds any code smells, a smell detector should make smell information as available as soon as possible, with little effort on the part of the programmer.*

Relativity *– A tool should place more emphasis on smells that are more difficult to recognize without a tool.*

Scalability *– A proliferation of smells in code should not cause the tool to overload the programmer with smell information.*

Unobtrusiveness *– A smell detection tool should not block the programmer while gathering, analyzing and displaying information about smells.*

Expressiveness – *A smell detector should go further than simply telling the programmer that a smell exists; it should help the programmer find the source(s) of the problem by explaining why the smell exists.*

Context-Sensitivity *– A smell detector should first and foremost point out smells relevant to the current programming context. Fixing smells in a context-insensitive manner may be a premature optimization.*

Relationality *– A smell detection tool should be capable of showing relationships between code fragments that give rise to smells.*

Monteiro and Fernandes [S25] reviewed code smells from an aspect-oriented programming perspective. New code smells concerning aspect orientation were also introduced in this contribution. Investigating how code smells and refactoring relate to the paradigm of aspect oriented programming is interesting, but out of scope for this thesis, so it will not be discussed further. Liu et al. [S15] propose a resolution order for when removing code smell. They postulate the necessity to arrange resolution order of code smells and describes how both the result and the cost of removing code smells can vary depending on resolution order. The evaluated code smells are: *Duplicate Code, Divergent change, Long Method, Large Class, Long Parameter List, Feature Envy, Useless Field, Useless Method, Useless Class* and *Primitive Obsession*. For each pair of these smells the following situations are investigated:

a) Will different resolution orders lead to different resulting systems?
b) Will the resolution of a bad smell ease the detection of the other?
c) Will the resolution of a bad smell complicate the detection of the other?
d) Will the resolution of a bad smell ease the modification (resolution) on the other bad smell?
e) Will the resolution of a bad smell complicate the modification on the other?

5 Discussion

This section discusses the main findings from the literature review as presented in Section 4. Initially the 3 sub-research questions will be recaptured and results will be summarized in order to get a more clear view of the state of the art in the field. Subsequently, general tendencies within the research will be discussed. Finally, the knowledge gap identified in the state of the art will be discussed.

5.1 The Effects of Code Smells (SQ1)

The results relevant for **SQ1** are the contributions summarized under the topic *Effects of code smells* in the section on empirical contributions in Chapter 4. The section does, however, only include two contributions. Hamza et al. [S14] investigates the effect of code smells in the context of removing them and the cost related to that. Li and Shatnawi [S45] have positively linked Shotgun Surgery, God Class and God Method to increased error rate in software projects in their 2007-analysis of the Eclipse code base and bug reports. Using Fowler's 22 initial code smells as reference, it leaves us with 19 code smells lacking of evidence that their presence have a negative and perceivable impact on error rate or other quality related properties. The overall goal of empirical articles on *effects of code smells* should be to have several sources documenting the effects in several contexts relevant for practitioners. Furthermore, an increase in error rate is only *one* interesting aspect to investigate within a wide range of possible aspects within a maintenance project. Some other quality aspects that the presence of code smells could affect are:

- Time/cost needed to implement additional functionality in an existing system
- Time/cost needed to make changes or to fix bugs to an existing code
- Time/cost needed to understand the code for people unfamiliar with it

Moreover, practitioners also need additional information such as the impact size of code smells if any tradeoffs need to be taken during development or maintenance. How worried should I be about the presence of certain code smells and under which circumstances?

5.2 The Effects of Refactoring (SQ2)

The only result directly relevant to answer this question is the single contribution under the topic *Effects of refactoring.* Shrivastava and Shrivastava [S40] presented a case study in which several steps of refactorings were applied to an application and metrics were used to measure the quality between and after each step. They present metrics that show an increase in some values and a decrease in other values. However, it is possible to see that there is no attempt to operationalize any relevant maintainability concept or other software quality by using these metrics. It could be true that most developers would find the application in question to have a higher quality after the refactorings were applied, but empirical evidence such as reduction in effort or defect rate is not present in the study. It would be more helpful to other developers and researchers if the work could illustrate the advantages of refactoring by associating them to measures more directly relevant to a maintenance project (e.g., a decrease in defect rates or effort needed to add features to the product). Showing changes in code metrics is only valuable if you can further link those metrics directly to more concrete quality aspects.

This leads us to the conclusion that empirical evidence of the effects of refactoring is a wide-open research area. There is an increasing interest in refactoring research, there is considerable support for refactoring in the most popular IDEs, and there seems to be much interest in this topic within the major practitioner conferences in combination with new practices and processes such as TDD and XP. Could putting significant effort into refactoring activities be justified from a manager's point of view with this provided information? [S10] states that *root canal refactoring* (the process of intensively refactoring code) is left to managers' decisions, as opposed to *floss refactoring* [S10], which is a decision left to the individual developer. In both contexts, neither the manager will pay for work that doesn't add any value (empirically validated) nor a developer would find it easy to justify the time spent in refactoring if they are just required to deliver code in time. Nonetheless, Martin Fowler suggests how to handle schedule driven managers:

"Of course, many people say they are driven by quality but are more driven by schedule. In these cases I give my more controversial advice: Don't tell! Subversive? I don't think so. Software developers are professionals. Our job is to build effective software as rapidly as we can. My experience is that refactoring is a big aid to building software quickly. If I need to add a new function and the design does not suit the change, I find it's quicker to refactor first and then add the function. If I need to fix a bug, I need to understand how the software works – and I find refactoring is the fastest way to do this. A schedule-driven manager wants me to

do things the fastest way I can; how I do it is my business. The fastest way is to refactor; therefore I refactor." [2]

Developers are refactoring, and will probably continue to do this, regardless of empirical evidence. It would, however, be of vital importance to validate its effects in order to enable managers and customers to perform tradeoffs during product maintenance and acceptance, whether it is in the form of *floss refactoring* or time-consuming *root canal refactoring.*

5.3 State of the art in methods and tool availability (SQ3)

Section 4 displayed a wide set of tools and methods for detecting code smells. Of the 28 design contributions, 22 related to code smells detection, 8 related to performing refactorings, and 13 related to the process of making refactoring decisions (each article can span several purposes). The review found 15 contributions presenting tools, but only 10 of these contributions could show to an available tool (a tool that was available for developers and researchers to test out, either commercially or open source). Some contributions stated that the tool used in the contribution was only a prototype or a novel tool, while other contributions never mention whether the tool in question was available or not. This could be an indicator that the implementation of the methods into tools is not getting enough attention. Within the available tools, it is still required to use a number of different tools since they are tailored to one or a few code smells. This again means that there might be a lack of integration with respect to tools supporting different code smells as well as amongst different refactoring activities that belong together.

The state of the art when it comes to commercial tool support for refactoring is, however, not as bad as one can get the impression of from the numbers and summaries in this review. This claim is partly based on my experience as developer, but also backed by the 2006-survey of refactoring tools presented by Drozdz et al. [S19]. The most used IDEs have advanced refactoring support and can perform many different refactorings including: *Extract Method, Move Method, Extract Superclass, Encapsulate Field, Introduce Parameter Object, Remove Middle Man, Replace Inheritance with Delegation,* and *Replace Constructor With Factory Method* [26-28]. This shows well-developed refactoring support for developers, as opposed to code smell detection tools. The reasons for this could be many, but the lack of formalized descriptions of code smells is most likely to be the cause of this.

This literature review has identified several methods for detecting several code smells, but many of these techniques lack tool support. Others have tool support but as independent tools

or IDE plug-ins. A comprehensive and functional smell detector seems far away, and thus the existing IDE refactoring support is expected to be mainly usable within floss refactoring, as explained by Murphy-Hill and Black [S10]:

"It appears that programmers refactor frequently to maintain healthy code, interleaving refactoring with other tasks such as adding features. We call this floss refactoring*, and contrast it with* root canal refactoring*, where a programmer refactors code intensively and exclusively once it has become unhealthy"*

5.4 Gap between Refactoring Tools and Code Smell Detection Tools

This review found that there is a large compendium on code smell tools research, but nearly no tools. The opposite is true for refactoring: There exist a lot of available tools, but hardly any research on such tools. The nature of refactoring and code smells, as described by Fowler [2], could account for this difference. Refactorings are well formalized and procedural in its form. They have a step-by-step explanation on how to proceed forth and the relatively early tool support in the popular IDEs might also indicate that the work of making these tools were simpler than those for detecting code smells. The process of presenting refactoring opportunities is, however, not as straightforward and this is also a subject with more research efforts, although maybe not as much as could be desired. It is understandable that industrial IDEs as IntelliJ and Visual Studio aren't broadcasting their algorithms. But it is potentially dubious that researchers are spending a lot of time researching on the detection of code smells and IDE developers are implementing refactoring support, without any evident communication or cooperation between each other.

5.5 The Current Focus on Design Contributions

As shown in Section 4, 61% of the contributions were methods and tools. Improvements of existing tools and methods and the addition of new tools are valuable. These tools are, however, only helping people that already are refactoring. If every developer were refactoring and waiting for such tools, they would be very beneficial. If nobody were refactoring, these tools would be useless. The truth is obviously somewhere in-between. Some developers are using refactoring tools and actively looking for code smells either manually or with existing tools, other developers are not. The bottom line is that as long as there is little evidence for the effects of code smells and refactoring, a substantial part of developers (and researchers) will await with the embracement of these techniques. Some effort (24%) has been focused on conducting empirical studies, but as discussed, only a small segment addresses the effects of code smells and refactoring. When looking on the contributions from this perspective, the

field might have been better off with less emphasis on finding new innovating methods and tools. It is not to say that the work is uninteresting or not valuable, it might, however, be a bit early to focus on the details until the most basic evidence is available.

There is no obvious answer to why research has been so heavily focused towards making design contributions, but there are some elements that might contribute to this. Empirically validating the impact of code smells and refactoring is difficult. You would need an industrial relevant software project and be able to separate the impacts of code smells and refactoring from all the other influencing factors. Proving that it was indeed the refactoring effort that improved the quality of the system and not one of the near endless other factors is both time consuming and difficult. Another possible factor when doing research in the field is economy. Comprehensive empirical studies could be more expensive than design research. Development of tools on the other hand, would imply a greater commercial gain than empirical research. Producers of commercial IDEs could integrate them into their own product and companies that spend a lot of effort on maintaining software systems could potentially save a lot of money on such tools and could also be interesting in paying for it.

5.6 General Tendencies within the Current research on this Topic

It is worth noting that although there has been written nearly 50 articles in the field relevant to this topic, rather few groups of researchers have been the main contributors. Mäntylä et al. is responsible for most of the material on the subjective evaluations, and the detection of code smells is, at least for now, heavily dependant on these subjective evaluations. Groups of authors leaded by Counsell, Moha and Tsantalis are important contributors within this topic. They have contributed with a total of 15 articles. Without those teams, only 31 contributions would be left in the review. However, of the 31 contributions then left, only four would be empirical. Counsell is the author or co-author of six contributions, all empirical, while Mäntylä has authored two empirical contributions. It might be a weakness for the field that one group of researchers is behind half of the empirical contributions. Another trend worth mentioning is the geographic variations. Several contributions have authors from different countries and continents, and this kind of information is rarely explicitly mentioned. Research institutions are, however, usually mentioned and e-mail addresses are also usually a good indicator as the top domains of these addresses usually correspond with the country of origin for a given researcher or research center. The one observation I would like to address based on the impression I've got from the review process, is what seems to be a lack of

contributions from America. The clear impression I've got after the review work is that majority of the contributions stems from Europe, with Asia as the second most productive continent.

5.7 Limitations Found in the Current State of Art

Throughout the review it was possible to observe that very few contributions are evaluated or used in an industrial setting. The vast majority of the methods are either validated on test code crafted by the researchers or on open source projects. The four contributions noted as being industrial validated include the study reported by Tsantalis in [S30], where one independent designer assessed "*the soundness of the […] refactoring opportunities*" and the study reported by Van Emden and Moonen in [S7], in which the studied code was a tool that was developed in an academic setting, but was being prepared to be released as commercial software. Experiences from using these methods in an industrial setting certainly would improve the quality of the evaluations performed in the tools and methods, which in turn may identify the usefulness or feasibility of such techniques or practices in different industrial settings.

Of all collected articles, very few of them were empirical studies, whereas the focus seems to be on extending/confirming existing theories/claims and to create new methods and tools based on these. As scientific research in the field of software engineering is fairly young, it has probably not come as far as in fields like medicine or social sciences. It is for example not trivial to sort the empirically based articles into groups of case studies, surveys or experiments. Research method is rarely explicitly mentioned and is often "something in between". Glass et al. [29] describe software engineering researchers' approach to research methods as: *"SE researchers tend to analyze and implement new concepts, and they do very little of anything else"*. When researching on the properties of code smells and refactoring (i.e. refactoring chains or consequences of code smells), most of the data will come from looking into code repositories, code metrics and output from mining tools. This could be one of the reasons for not stating a specific research method. Analyzing different sets of data to draw conclusions lacks the variables of experiments but will often not fully qualify as a case study.

5.8 Potential Avenues for Future Research

What kind of empirical validated knowledge should researchers in the field aim for? Research on refactoring needs to validate the effect of refactoring. Some research sub-areas the community could address are: More concrete and visible effects of different refactorings on different product/process quality aspects, Measurement and evaluation of the impact (type and size) of refactoring/code smells in a realistic yet feasible way. Studies aiming at investigating the costs and benefits of refactoring, Studies aiming at investigating/identifying potential risks represented by the presence of certain code smells,

The development of unified, flexible and extensible frameworks for defining code smells and detecting them in a variety of contexts could constitute another area of work. Furthermore, the development of more concise evaluation frameworks could aid comparable evaluations of tool contributions and improve their quality and increase their adoption within industry.

6 Threats to Validity

This chapter discusses the most important threats to the validity of the results obtained from this literature review.

6.1 Choice of Research Databases

The selection of ACM Digital Library, IEEE Xplore and ISI Web of Knowledge as the only sources for contributions could be seen as a limitation. However, by going beyond the three major research databases in the SE field, it would be required to spend a lot of effort in identifying the most relevant additional databases

6.2 Construction of Queries

The query used for searching the research databases for contributions could be more comprehensive. It could have included more keywords as *refactoring* and *restructuring.* The query was focused towards code smells because the thesis was mainly interested in refactoring as part of a strategy to remove code smells. Refactoring in this sense is usually motivated by noticing a code smell [2] and thus the absence of *refactoring* in the query would help to remove irrelevant articles.

6.3 Application of the Inclusion- and Exclusion Criteria

The criteria for inclusion and exclusion were well formalized, but the process of applying these criteria was still subjective and performed only by one person. In order to reduce threats to internal validity when categorizing the contributions, several persons should ideally perform the categorization individually and the inter-rater agreement should be observed. Inter-rater agreement is the extent to which evaluators agree. High inter-rater agreement is a positive indication of the reliability of the subjective evaluations. Lack of inter-rater agreement can mean that some of the evaluators are mistaken in their evaluation [30].

6.4 Data Extraction

During the final data extraction, 46 contributions were analyzed and different pieces of information were gathered. Due to the high variability on the quality, location and level of detail of the information that was provided in the contribution, (findings, research methods and other type of information) identifying the different pieces of data within the documents was highly difficult. This situation may have lead to missing information due to the fact that only one person performed the data extraction.

7 Conclusions

This thesis reported a literature review conducted on the topic of code smells and refactoring, using most of the techniques of a systematic literature review.

The review found a significant growth of publications on the topic of code smells and refactoring in the last five years, which indicates the increasing popularity of the topic within the research community. Nevertheless, the review found in general a lack of empirically sound evidence that could be used to guide practitioners on choosing the best refactoring strategies for improving maintainability.

Relatively few articles reported empirical studies (24%) as opposed to design research contributions (61%). Only 13.8% of these design contributions reported any type of experiences from its usage in industry. In addition to empirical and design contributions, a few articles summarized and provided theoretical contributions. One of the summarizing articles presented an overview of the state of the art of refactoring support for the most popular IDEs.

The *direct* effects of code smells were only investigated in one article. A total of six code smells were investigated. Among those, *Shotgun Surgery, God Class,* and *God Method* were positively correlated with an increase in the error probability in the code. In addition to this, one study investigated another effect of code smells: The number of refactorings required to eradicate the 22 code smells described by Fowler [2]. The review also only found one study that empirically validated the effects of refactoring. *Extract Class, Extract Method,* and *Extract Subclass* were applied sequentially and metrics were used to measure the quality of the code before and after each stage of applying refactorings. The authors found several changes in the metrics and claimed that the code had improved. They did however not state how much the code had improved or for which quality aspects. Nor did they provide sound evidence for the claim.

This review identified several methods that target the detection of one or more code smells. However, there is no unified way of describing code smells or the eradication of code smells. This is probably the main reason for the number of different, often overlapping, approaches to detecting code smells. At some point, work must be done to establish a form of standardization or formalization of the process of detecting and analyzing code smells.

According to all the various contributions on code smell detection methods and tools, the best approaches for detecting the different smells varies greatly depending on smell and context. If the effort needed to create effective and efficient detection tools is as significant as the presence of this plethora of contributions suggests, it might not be feasible for any team of developers and researchers to create one comprehensive tool. A formalized way of gathering and analyzing and presenting information on code smells and a framework for displaying the different code smells is one flexible solution to this problem.

The research on refactoring tools is limited. The review found six contributions related to performing refactorings, but five of them also focused on either the detection of code smells or providing help with the refactoring decision, or both of these, in addition to performing the refactoring. Nonetheless, the main IDEs have heavy refactoring support [S19]. The reason for this could be that while code smells are informally described, there exist more precise formalizations of the various refactorings [2].

8 Future work

This section elaborates on potential work that could be done to achieve a deeper understanding on the research questions presented in this thesis.

8.1 Elaborating on the Research Questions

Identifying relevant work from the references found in the primary studies could possibly increase the result set of the data. This involves going through the references in the primary studies and applying the exclusion and inclusion criteria on the referenced articles. As a relevant article references these articles, the chance for them to also be relevant is fairly high. Most of the relevant articles should hopefully already be present in the result set, but articles not present in any of the chosen databases, or not returned by the queries could exist. Increasing the result set would benefit the review in that it would increase the body of knowledge contained in the review and could return more interesting findings.

Another way to increase the amount of data would be to look into the area of the grey literature. There is most probably not a great amount of scientific empirical research within the grey contributions, but tools and methods would be present. One example of this is the refactoring tools embedded in modern IDEs. Eclipse has had refactoring support for quite some time. Details on the implementation and possible empirical support for these would be interesting in an extended review. The inclusion of grey literature in the review would, however, bring implications both in terms of internal validity and considerations regarding the weighing of contributions. Should grey research impact aggregated results in the same way as validated and empirically sound evidence?

8.2 Research Method Suggestions

This research has shown that there are great differences in what information are being provided with respect to validation and availability of tools. Making this information easily available to developers and researchers should increase the possibility of taking the given research into account. Elaborating on evaluation frameworks that can rank or categorize the usefulness and applicability of methods and tools in an industrial setting constitutes another area for future work.

Bibliography

1. Cooling, J., *Software maintenance concepts and practice : Armstrong A. Takang, Penny A. Grubb, International Thompson Publishing Inc., 1996, Softbound £22.95. ISBN 1-85032-192-2.* Microprocessors and Microsystems, 1996. 20(5): p. 311-311.
2. Fowler, M., et al., *Refactoring: Improving the Design of Existing Code*. 1999: Addison-Wesley Professional.
3. Kitchenham, B., *Procedures for performing systematic reviews*, in *Technical Report TR/SE-0401*. 2004.
4. Fowler, M. *MF Bliki: EtymologyOfRefactoring*. [cited 2010 01 March]; Available from: http://martinfowler.com/bliki/EtymologyOfRefactoring.html.
5. William, F.O., *Refactoring object-oriented frameworks*. 1992, University of Illinois at Urbana-Champaign.
6. *DevWeek | Conference Sessions*. [cited 2010 17 March]; Available from: http://www.devweek.com/sessions/conference2.asp.
7. *My JavaOne Talk on Advanced Refactoring | Java.net*. [cited 2010 28 Feb.]; Available from: http://weblogs.java.net/blog/tball/archive/2007/03/my_javaone_talk_1.html.
8. *JavaZone 2008 - Agenda*. [cited 2010 28 Feb]; Available from: http://javazone.no/incogito/session/Clean+Code+III%3A+Functions.html.
9. Van Emden, E. and K. Moonen, *Java quality assurance by detecting code smells*, in *Working Conf. on Reverse Engineering*. 2002. p. 97-106.
10. Marinescu, R. *Measurement and quality in object-oriented design*. in *Intl. Conf. on Softw. Maint.* 2005.
11. Kerievsky, J., *Refactoring to Patterns*. 2004: Pearson Higher Education.
12. Mens, T. and T. Tourwé, *A Survey of Software Refactoring.* IEEE Trans. Softw. Eng., 2004. 30(2): p. 126-139.
13. Brereton, P., et al., *Lessons from applying the systematic literature review process within the software engineering domain.* J. Syst. Softw., 2007. 80(4): p. 571-583.
14. Holt, N.E., *A systematic review of case studies in Software engineering*. 2006, University of Oslo: Oslo.
15. Dybå, T., et al., *Empirical studies of agile software development: A systematic review.* Inf. Softw. Technol., 2008. 50(9-10): p. 833-859.
16. *ACM Digital Library*. [cited 2010 01 March]; Available from: http://portal.acm.org/portal.cfm.
17. *IEEE Xplore*. [cited 2010 01 March]; Available from: http://ieeexplore.ieee.org/.
18. *ISI Web of Knowledge*. [cited 2010 01 March]; Available from: www.isiknowledge.com.
19. Service, T.G.L.N. *GreyNet*. 2010 [cited 2010 29.03]; Available from: http://www.greynet.org/.
20. *Zotero | Home*. [cited 2010 01 March]; Available from: http://www.zotero.org.
21. Martin, P.Y. and B.A. Turner, *Grounded theory and organizational research.* The Journal of Applied Behavioral Science, 1986. 22(2): p. 141-141.
22. *Apache Tomcat - Welcome!* [cited 2010 01 March]; Available from: http://tomcat.apache.org.
23. *Mozilla.org - Home of the Mozilla Project*. [cited 2009 09 Dec]; Available from: http://www.mozilla.org.
24. *Home :: Bugzilla :: bugzilla.org*. [cited 2010 01 March]; Available from: http://www.bugzilla.org.

25. Usha, K., N. Poonguzhali, and E. Kavitha, *A Quantitative Model for Improving the Effectiveness of the Software Development Process using Refactoring.* 2009.
26. *Help - Eclipse SDK*. [cited 2010 01 March]; Available from: http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm.
27. *Refactoring - NetBeans Wiki*. [cited 2010; Available from: http://wiki.netbeans.org/Refactoring.
28. *IntelliJ IDEA :: Java refactoring plus sophisticated code refactoring for JSP, XML, CSS, HTML, JavaScript* *http://www.jetbrains.com/idea/features/refactoring.html*. [cited 2010 01 March].
29. Glass, R.L., I. Vessey, and V. Ramesh, *Research in software engineering: an analysis of the literature.* Information and Software Technology, 2002. 44(8): p. 491-506.
30. Mantyla, M.V., *An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement*, in *2005 International Symposium on Empirical Software Engineering*. 2005. p. 10 pp.

Appendix

Appendix A - Studies Included in the Review

S1 Rao, A.A. and K.N. Reddy, *Detecting bad smells in object oriented design using design change propagation probability matrix.* Imecs 2008: International Multiconference of Engineers and Computer Scientists, Vols I and Ii, 2008: p. 1001-1007.

S2 Trifu, A. and U. Reupke, *Towards Automated Restructuring of Object Oriented Systems*, in *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*. 2007. p. 39-48

S3 Pietrzak, B. and B. Walter, *Leveraging code smell detection with inter-smell relations.* Extreme Programming and Agile Processes in Software Engineering, Proceedings, 2006. 4044: p. 75-84.

S4 Walter, B. and B. Pietrzak, *Multi-criteria detection of bad smells in code with UTA method.* Extreme Programming and Agile Processes in Software Engineering, Proceedings, 2005. 3556: p. 154-161.

S5 Kiefer, C., A. Bernstein, and J. Tappolet, *Mining Software Repositories with iSPAROL and a Software Evolution Ontology*, in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. 2007, IEEE Computer Society. p. 10

S6 Parnin, C., C. Görg, and O. Nnadi, *A catalogue of lightweight visualizations to support code smell inspection*, in *Proceedings of the 4th ACM symposium on Software visualization*. 2008, ACM: Ammersee, Germany. p. 77-86

S7 van Emden, E. and L. Moonen, *Java quality assurance by detecting code smells*, in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. 2002. p. 97-106

S8 Piveta, E., et al., *Representing refactoring opportunities*, in *Proceedings of the 2009 ACM symposium on Applied Computing*. 2009, ACM: Honolulu, Hawaii. p. 1867-1872

S9 Murphy-Hill, E., *Scalable, expressive, and context-sensitive code smell display*, in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. 2008, ACM: Nashville, TN, USA. p. 771-772

S10 Murphy-Hill, E. and A.P. Black, *Seven habits of a highly effective smell detector*, in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*. 2008, ACM: Atlanta, Georgia. p. 36-40

S11 Simon, F., F. Steinbruckner, and C. Lewerentz, *Metrics based refactoring*, in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. 2001. p. 30-38.

S12 Steimann, F., P. Mayer, and A. Meißner, *Decoupling classes with inferred interfaces*, in *Proceedings of the 2006 ACM symposium on Applied computing*. 2006, ACM: Dijon, France. p. 1404-1408

S13 Arevalo, G., S. Ducasse, and O. Nierstrasz, *Discovering Unanticipated Dependency Schemas in Class Hierarchies*, in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*. 2005. p. 62-71

S14 Hamza, H., S. Counsell, and T. Hall, *Code Smell Eradication and Associated Refactoring.* Proceedings of the 2nd European Computing Conference - New Aspects on Computers Reseach, 2008: p. 102-107.

S15 Liu, H., et al., *Facilitating software refactoring with appropriate resolution order of bad smells*, in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*. 2009, ACM: Amsterdam, The Netherlands. p. 265-268.

S16 Nodler, J., H. Neukirchen, and J. Grabowski, *A Flexible Framework for Quality Assurance of Software Artefacts with Applications to Java, UML, and TTCN-3 Test Specifications*, in *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*. 2009. p. 101-110.

S17 Pérez, J. and Y. Crespo, *Perspectives on automated correction of bad smells*, in *Proceedings of*

the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. 2009, ACM: Amsterdam, The Netherlands. p. 99-108

S18 Reddy, K.R. and A.A. Rao, *Dependency oriented complexity metrics to detect rippling related design defects.* SIGSOFT Softw. Eng. Notes, 2009. 34(4): p. 1-7.

S19 Drozdz, M., et al., *Refactoring tools and complementary techniques.* 2006 Ieee International Conference on Computer Systems and Applications, Vols 1-3, 2006: p. 684-687.

S20 Eichberg, M., et al., *Comprehensive software understanding with SEXTANT*, in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. 2005. p. 315-324

S21 Fokaefs, M., N. Tsantalis, and A. Chatzigeorgiou, *JDeodorant: Identification and Removal of Feature Envy Bad Smells*, in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. 2007. p. 519-520

S22 Munro, M.J., *Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code*, in *Software Metrics, 2005. 11th IEEE International Symposium*. 2005. p. 15

S23 Mantyla, M.V., *An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement*, in *Empirical Software Engineering, 2005. 2005 International Symposium on*. 2005. p. 10 pp.

S24 Mantyla, M.V. and C. Lassenius, *Subjective evaluation of software evolvability using code smells: An empirical study.* Empirical Software Engineering, 2006. 11(3): p. 395-431.

S25 Monteiro, M.P. and J.M. Fernandes, *Towards a catalog of aspect-oriented refactorings*, in *Proceedings of the 4th international conference on Aspect-oriented software development*. 2005, ACM: Chicago, Illinois. p. 111-122

S26 Moha, N., et al., *A domain analysis to specify design defects and generate detection algorithms.* Fundamental Approaches to Software Engineering, Proceedings, 2008. 4961: p. 276-291.

S27 Moha, N., Y.G. Gueheneuc, and P. Leduc, *Automatic generation of detection algorithms for design defects.* ASE 2006: 21st IEEE International Conference on Automated Software Engineering, Proceedings, 2006: p. 297-300.

S28 Moha, N., et al., *DECOR: A Method for the Specification and Detection of Code and Design Smells.* Software Engineering, IEEE Transactions on, 2009. PP(99): p. 1.

S29 Tsantalis, N. and A. Chatzigeorgiou, *Identification of Move Method Refactoring Opportunities.* Software Engineering, IEEE Transactions on, 2009. 35(3): p. 347-367.

S30 Tsantalis, N. and A. Chatzigeorgiou, *Identification of Extract Method Refactoring Opportunities*, in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*. 2009. p. 119-128

S31 Tsantalis, N., T. Chaikalis, and A. Chatzigeorgiou, *JDeodorant: Identification and Removal of Type-Checking Bad Smells*, in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. 2008. p. 329-331

S32 Seng, O., J. Stammel, and D. Burkhart, *Search-based determination of refactorings for improving the class structure of object-oriented systems.* Proceedings of the 8th annual conference on Genetic and evolutionary computation, 2006: p. 1909-1916

S33 Geiger, R., et al., *Relation of code clones and change couplings.* Fundamental Approaches to Software Engineering, Proceedings, 2006. 3922: p. 411-425.

S34 Lerner, R.M., *At the forge: checking your ruby code with metric_fu.* Linux J., 2009. 2009(183): p. 5

S35 Counsell, S., *Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?*, in *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on*. 2008. p. 111-122.

S36 Counsell, S. and E. Mendes, *Size and Frequency of Class Change from a Refactoring Perspective*, in *Software Evolvability, 2007 Third International IEEE Workshop on*. 2007. p. 23-28.

S37 Counsell, S., et al., *The Effectiveness of Refactoring, Based on a Compatibility Testing*

S37 Counsell, S., et al., *The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph*, in *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*. 2006. p. 181-192.

S38 Counsell, S., et al., *Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS*, in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. 2006, ACM: Rio de Janeiro, Brazil. p. 288-296

S39 Nasehi, S.M., G.R. Sotudeh, and M. Gomrokchi, *Source code enhancement using reduction of duplicated code.* Proceedings of the IASTED International Conference on Software Engineering, 2007: p. 192-197.

S40 Vasudeva Shrivastava, S. and V. Shrivastava, *Impact of metrics based refactoring on the software quality: a case study*, in *TENCON 2008 - 2008, TENCON 2008. IEEE Region 10 Conference*. 2008. p. 1-6.

S41 Counsell, S., et al., *Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study.* SIGSOFT Softw. Eng. Notes, 2006. 31(5): p. 1-10.

S42 Mens, T., T. Tourwe, and F. Munoz, *Beyond the refactoring browser: advanced tool support for software refactoring*, in *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*. 2003. p. 39-44.

S43 Tourwe, T. and T. Mens, *Identifying refactoring opportunities using logic meta programming*, in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. 2003. p. 91-100

S44 Gîrba, T., et al., *Using concept analysis to detect co-change patterns*, in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. 2007, ACM: Dubrovnik, Croatia. p. 83-89

S45 Li, W. and R. Shatnawi, *An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution.* Journal of Systems and Software, 2007. 80(7): p. 1120-1128.

S46 Avdagic, Z., D. Boskovic, and A. Delic, *Code evaluation using fuzzy logic.* Proceedings of the 9th Wseas International Conference on Fuzzy Systems - Advanced Topics on Fuzzy Systems,