JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2011; **23**:179–202
Published online 20 October 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.521

Code Bad Smells: a review of current knowledge

Min Zhang[1,*,†], Tracy Hall[2] and Nathan Baddoo[1]

[1]*School of Computer Science, University of Hertfordshire, Hertfordshire, U.K.*
[2]*Information Systems and Computing, Brunel University, Uxbridge, U.K.*

SUMMARY

Fowler *et al.* identified 22 Code Bad Smells to direct the effective refactoring of code. These are increasingly being taken up by software engineers. However, the empirical basis of using Code Bad Smells to direct refactoring and to address 'trouble' in code is not clear, i.e., we do not know whether using Code Bad Smells to target code improvement is effective. This paper aims to identify what is currently known about Code Bad Smells. We have performed a systematic literature review of 319 papers published since Fowler *et al.* identified Code Bad Smells (2000 to June 2009). We analysed in detail 39 of the most relevant papers. Our findings indicate that Duplicated Code receives most research attention, whereas some Code Bad Smells, e.g., Message Chains, receive little. This suggests that our knowledge of some Code Bad Smells remains insufficient. Our findings also show that very few studies report on the impact of using Code Bad Smells, with most studies instead focused on developing tools and methods to automatically detect Code Bad Smells. This indicates an important gap in the current knowledge of Code Bad Smells. Overall this review suggests that there is little evidence currently available to justify using Code Bad Smells. Copyright © 2010 John Wiley & Sons, Ltd.

Received 9 February 2010; Revised 11 June 2010; Accepted 21 July 2010

KEY WORDS: Systematic Literature Review; Code Bad Smells; refactoring; empirical software engineering

1. INTRODUCTION

The 22 Code Bad Smells identified by Fowler *et al.* [1] aim to indicate software refactoring opportunities. Fowler *et al.* [1] suggest that Code Bad Smells can 'give you indications that there is trouble that can be solved by a refactoring'. They are widely used for detecting refactoring opportunities in software [2], and few people argue about their usefulness. However, two recent studies [3, 4] show that not all Code Bad Smells lead to reduced software reliability. Consequently, in this paper we ask: Do we know enough about Code Bad Smells to justify their use? And what is the current state of knowledge on Code Bad Smells? This paper presents our investigation reviewing the current state of published research on Code Bad Smells.

Fowler and Beck [1] informally introduce 22 Code Bad Smells. These Code Bad Smells range from straightforward software coding problems, such as 'Switch Statement', to complicated software structure problems, such as 'Shotgun Surgery' (see Appendix A for a summary of Code Bad Smells). It is likely that our knowledge varies from one Code Bad Smell to the next. In this paper we examine the current evidence for using each Code Bad Smell.

We review all studies of Code Bad Smells published in leading software engineering journals and proceedings since Fowler *et al.* [1] identified Code Bad Smells (Our review spans 2000 to

*Correspondence to: Min Zhang, School of Computer Science, University of Hertfordshire, Hertfordshire, U.K.
†E-mail: m.1.zhang@herts.ac.uk

June 2009). We select papers and extract data using a systematic literature review protocol which follows Kitchenham's [5] literature review guidelines, and statistically analyse the results to answer the following research questions.

1.1. RQ1: Which Code Bad Smells have attracted the most research attention?

The 22 Code Bad Smells introduced by Fowler *et al.* [1] target different coding problems. Indeed it is likely that some Code Bad Smells receive more research attention than others. Consequently, we investigate the intensity of previous studies of each Code Bad Smell. This will indicate whether some Code Bad Smells are popular whereas others lack attention. This will identify gaps in the current research of Code Bad Smells and provide directions for further study on Code Bad Smells.

1.2. RQ2: What are the aims of studies on Code Bad Smells?

Investigating why researchers are studying Code Bad Smells can provide insight into what aspects of Code Bad Smells are most commonly studied, i.e., we will capture whether the current studies of Code Bad Smells mainly aim to introduce tools/methods to detect Code Bad Smells or whether studies explore methods to refactor Code Bad Smells from code. Furthermore, we will also identify whether the aims of studies for different Code Bad Smells vary. This will tell us whether the different aspects of Code Bad Smells are studied in all Code Bad Smells, and provide information about which aspects lack attention.

1.3. RQ3: What methods have been used to study Code Bad Smells?

This analysis will identify whether empirical as opposed to theoretical approaches have been used to investigate Code Bad Smells. We will also identify the source of data collected in studies, in particular whether the data used are real world, open source, commercial or academic. This will indicate the balance of empirical evidence and theoretical underpinnings presented by the published studies, and suggest what data and methods should be applied in future studies of Code Bad Smells.

1.4. RQ4: What evidence is there that Code Bad Smells indicate problems in code?

Fowler *et al.* [1] suggest that Code Bad Smells cause detrimental effects on software and provide useful indicators of opportunities for refactoring. However, they do not provide any empirical evidence to support this claim. We explore whether any published studies provide empirical evidence to support Fowler *et al.*'s claims.

The remainder of this paper is structured: Section 2 describes our systematic literature review protocol. Section 3 provides the results of this systematic literature review. Our results are discussed and the research questions are answered in Section 4. Conclusions are drawn in Section 5. Section 6 describes the limitation of our studies, and suggests further investigations to follow-up this study.

2. REVIEW PROTOCOL

According to Kitchenham [5]:

> 'A systematic literature review is a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest'.

Kitchenham [5] also indicates that a systematic literature review is a useful approach to 'summarize the existing evidence concerning a treatment or technology'.

We adopted our own systematic literature review protocol based on Kitchenham's [5] systematic review guidelines. Four researchers are involved in our systematic literature review. This section briefly describes our review protocol, more details of which can be found in [6].

Table I. Targeted journals and proceedings.

Publication name	Abbreviation
IEEE Xplore (http://ieeexplore.ieee.org)	IEEE
Journal of Systems and Software	JSS
Empirical Software Engineering Journal	EMSE
Information and Software Technology Journal	IST
Journal of Software Maintenance and Evolution: Research and Practice	JSME
ACM Transactions on Software Engineering Methodology	TOSEM
Software: Practice and Experience Journal	SP&E

IEEE Xplore is an online database of the IEEE Computer Society. It contains all journals and conference proceedings published by IEEE. Appendix D lists the IEEE journals and conference proceedings included in this study.

2.1. *Define search strategy*

Our search strategy addresses the scope of our search for published studies, the search terms we used and the inclusion/exclusion criteria adopted for studies.

2.1.1. Scope. This systematic literature reviews used papers and publications from seven different sources (Table I).

There are two reasons for choosing IEEE Xplore. First, IEEE Computer Society is said to be one of the main platforms for the publication of software engineering research results [7]. Second, IEEE Xplore contains a wide range of journals and proceedings which means that a variety of studies on Code Bad Smells are likely to be contained in this database. In addition important joint IEEE/ACM proceedings are also included in this portal (for example, IEEE/ACM International Conference on Automated Software Engineering).

As including papers from one single publisher may be a bias in our study, we also consider papers from six software engineering journals. These journals have been used in Sjoeberg *et al.*'s [8] study and are thought to be the most representative leading journals in software engineering. We review all Code Bad Smell papers published in these software engineering publications since Fowler *et al.* [1] identified Code Bad Smells (2000 to June 2009 inclusive).

2.1.2. Search terms. Our search terms are constructed using the following strategy:

1. Derive major terms from research questions.
2. Identify alternative spellings or synonyms for major terms.
3. Check the search terms in relevant papers we already have.
4. Use the Boolean OR to combine alternative spellings and synonyms. Use the Boolean AND to link major terms.

The details about how we apply this strategy can be found in [6]. Our search terms are attached in Appendix B of this paper. In total, 319 papers were returned from our selected databases using these search terms.

2.1.3. Inclusion/exclusion criteria. Papers from the 319 identified are included for further analysis by applying the inclusion/exclusion criteria presented in Table II.

This systematic literature review excludes studies using secondary data. This is because although some studies using secondary data could improve the understanding of Code Bad Smells, these studies do not add new empirical evidence. As a consequence, we do not consider studies using secondary data in this systematic literature review.

2.2. *Define classification schemas*

In order to provide quantitative evidence to answer our research questions, defining classification schemas to accurately classify data collected from investigated papers is important in this literature

Table II. Inclusion/exclusion criteria.

Inclusion criteria	Exclusion criteria
1. A paper must be on the topic of software engineering. Consequently, it must be categorized into one or more of the 10 Knowledge Areas (KAs) defined by SWEBOK [9].	1. A paper just mentions the names of Code Bad Smells but does not further discuss or investigate.
2. A paper answers at least one of our research questions directly.	2. A paper uses secondary data (e.g., a literature review paper).
3. A paper is a complete paper rather than an abstract of an unfinished paper.	3. A paper is about software engineering of particular computer hardware.
4. A paper is written in English.	4. A paper is about methods of software engineering education.

Table III. Aims of studies classification schema.

Name	Code	Definition
Methods/tools to detect Code Bad Smells	DC	The aim of a study is to enhance tools/methods for detecting Code Bad Smells in source code.
Improve understanding of Code Bad Smells	IC	The aim of a study is to enhance the body of knowledge of Code Bad Smells, such as examining the effectiveness of Code Bad Smells, creating taxonomy to classify Code Bad Smells or formalizing the definitions of Code Bad Smells.
Refactor Code Bad Smells	RC	The aim of a study is to enhance the tools/methods for refactoring Code Bad Smells.
Others	OT	The aim of a study cannot be classified into the above categories.

review. For example, to answer research question 3, we need to classify the types of research data used in different studies so that we can quantitatively analyse which type is the most popularly used research data. We apply the following strategy for defining our classification scheme:

1. Identify any existing classification schemes relevant to our research questions.
2. Use existing classification schemes if appropriate to our research questions.
3. Otherwise build our own classification schemes.

The details about this can be found in [6].

2.2.1. Classifying the aims of studies. In this literature review, a classification schema is needed to classify the aims of studies on Code Bad Smells. As a consequence of finding no relevant existing classification schema, we used a bottom-up approach[‡] to build our own classification schema. Hence, our classification schema for the aims of studies is defined in Table III. The details about how we defined this classification schema can be found in [6].

2.2.2. Classifying empirical study methods. A classification schema is also needed to classify the empirical methods applied in the previous research of Code Bad Smells. In this literature review, an 'empirical study' is defined as a study using experiments, questionnaires/surveys, interviews, observations or case studies [11]. Using this definition study methods are classified in this review according to Table IV.

This systematic literature review counts studies according to the object of studies. If two or more studies in the same paper are conducted with the same goal, they are treated as a single empirical study, even if different sets of data are used in these studies. This is because the same

[‡]The bottom-up approach is based on grounded theory [10]. This approach is designed to build customized schemas based on collected research data.

Table IV. Study methods classification schema.

Name
Experiment
Questionnaire/survey
Interview
Observation
Case study
Non-empirical methods

Table V. Source of data classification schema.

Data type	Name	Definition
Subjective data	Student opinion	The data come from questionnaires, surveys or interviews using students.
	Professional opinion	The data come from questionnaires, surveys or interviews using professionals, such as software engineers and academics.
	Combination	The data combine information from students and professionals.
Objective data	Self-constructed project	The data are constructed for the purpose of the experiment.
	Academic project	The data come from projects developed by scientists, PhD students or research institutes.
	Commercial project	The data come from commercial projects.
	Student project	The data come from student projects.
	Open source	The data come from open source communities.
	Unclear	Unclear source.

object may be investigated through different dimensions. As a consequence, a paper may conduct several studies to explore a single object. If these studies are counted as different studies, the big investigation will be overrepresented, and the small investigation will be overlooked.

2.2.3. Classifying the source of data. To classify different sources of research data, a modified version of Sjoeberg *et al.*'s [8] classification schema of research data is used in our study. Our classification schema is shown in Table V.

2.3. Refinement of included papers

We performed two phases refinement of the included papers on our search results. These processes aim to eliminate papers that are not related to our research.

2.3.1. Phase One. This phase of refinement aims to eliminate papers easily identified as not related to our research. One researcher reads the title and abstract of each paper. Using our inclusion/exclusion criteria (Table II) this researcher judges whether a selected paper is an 'accept', 'reject' or 'not sure' paper. The 'accept' and 'not sure' papers are included in the next phase of our literature review, whereas the 'reject' papers are excluded. As a consequence, 89 out of 319 papers are selected into the next phase of our study. This means that 72% of the papers are rejected in this first phase of refinement.

An agreement test was conducted to measure the reliability of the judgements made by the first researcher. Ten papers are selected from the 319 papers in our search results using the systematic sampling strategy suggested by Saunders *et al.* [11]. Details about how we applied this strategy can be found in [6]. Each of these selected papers is reviewed by three researchers, who judge whether to classify each paper as 'accept', 'reject' or 'not sure'. The Kappa inter-rater test is used to test classification agreements between these three researchers (Appendix C). Comparing our Kappa test results with Landis and Koch [12] benchmarks, our refinement process is shown to be reliable.

2.3.2. Phase Two. This phase of the refinement process aims to exclude papers from the remaining set which do not answer our research questions. However, unlike the first phase of refinement, in the second phase, researchers read the whole research paper in detail. After this phase 39 out of 89 papers remain included in this systematic literature review as papers that address our research questions and conform to our inclusion criteria. A full list of these 39 papers is listed in Appendix E of this paper.

3. RESULTS

Taking a general look at the papers that investigate Fowler *et al.*'s Code Bad Smells, we found that most papers either focus on one or two Code Bad Smells or else all 22 Code Bad Smells together. Figure 1 shows the distribution of the investigated Code Bad Smells.

Figure 1 shows that 20 out of the total 39 papers investigate only one of Fowler *et al.*'s Code Bad Smells. Five papers investigate all of Fowler *et al.*'s 22 Code Bad Smells together, for example, Mantyla *et al.* (SLR 17. see Appendix E for the details of each paper) provide a taxonomy to classify Fowler *et al.*'s Code Bad Smells, and Mantyla *et al.* (SLR 16) investigate the opinions of developers on the existence of each Code Bad Smell. Figure 1 also shows that there are five papers that do not include any of Fowler *et al.*'s Code Bad Smells. Instead, they introduce new Code Bad Smells based on Fowler *et al.*'s [1] ideas. These papers extend the body of knowledge of Code Bad Smells and are therefore included in our literature review.

Figure 2 presents the number of papers for each year that this review covers. This figure shows that Code Bad Smell papers started to appear in 2001, peaked at 2004, slowly dropped between 2005 and 2007 and greatly dropped after 2008.

3.1. RQ 1: Which Code Bad Smells have attracted the most research attention?

Table VI presents the number of papers focusing on each Code Bad Smell. The details of which Code Bad Smells are investigated in each paper are presented in Appendix E.

Table VI shows that the Duplicated Code Bad Smell is most studied. This supports Fowler *et al.*'s [1] idea that 'Number one in the stink parade is duplicated code'. Besides the Duplicated Code Bad Smell, Feature Envy, Refused Bequest, Data Class, Long Method and Large Class Bad Smells are each studied in more than 20% of the research papers.

Furthermore, the Duplicated Code Bad Smell tends to be studied alone. This may mean that the Duplicated Code Bad Smell is different in nature from other Code Bad Smells. To further investigate this, we separated the research papers into two groups, the papers that include the Duplicated Code Bad Smell and the papers that do not include the Duplicated Code Bad Smell.

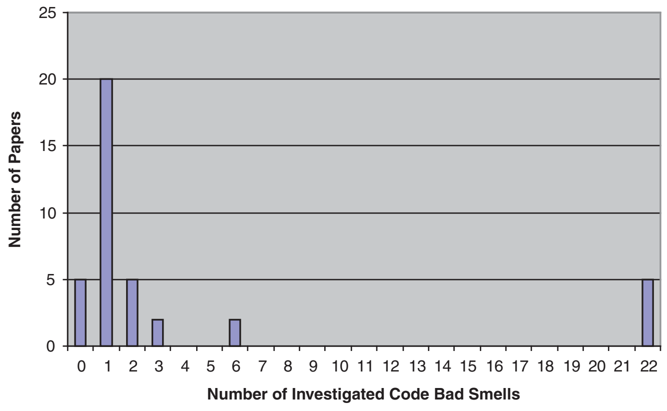

Figure 1. Distribution of Code Bad Smells in research papers.

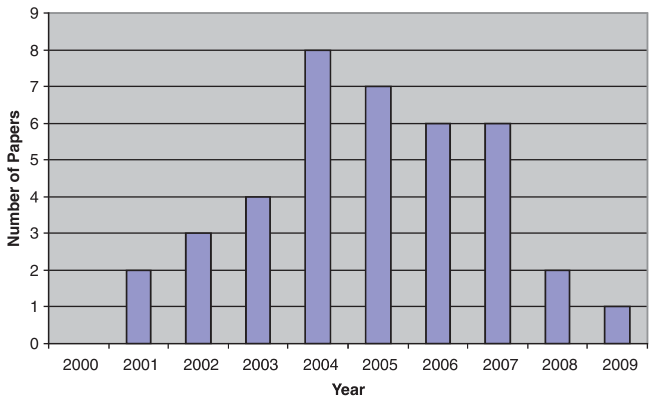

Figure 2. Number of research papers by year.

Table VI. Attention on Code Bad Smell.

Bad Smell name	No. of papers (Total number of papers = 39)	Percentage (%)
DUC—Duplicated Code	21	54
FE—Feature Envy	12	31
RB—Refused Bequest	11	28
DC—Data Class	10	26
LM—Long Method	8	21
LC—Large Class	8	21
LPL—Long Parameter List	7	18
SS—Shotgun Surgery	7	18
LAZ—Lazy Class	6	15
TF—Temporary Field	6	15
COM—Comments	6	15
SW—Switch Statements	6	15
DCP—Divergent Change	5	13
DAC—Data Clumps	5	13
PO—Primitive Obsession	5	13
PIH—Parallel Inheritance Hierarchies	5	13
SG—Speculative Generality	5	13
MC—Message Chains	5	13
MM—Middle Man	5	13
II—Inappropriate Intimacy	5	13
ACDI—Alternative Classes with Different Interfaces	5	13
ILC—Incomplete Library Class	5	13
OTHERS—Other Code Bad Smell	13	33

The percentage may not add up to 100% in this table, because one paper may investigate more than one Code Bad Smell.

We compare the proportion of papers that only investigate one Code Bad Smell between these two groups of papers. The results are summarized in Table VII.

Table VII suggests that the Duplicated Code Bad Smell is likely to be studied alone and is generally a single research area. The implications of the Duplicated Code Bad Smell having a different research profile will be discussed in Section 4.

In addition, we also notice that ten Code Bad Smells (Divergent Change, Data Clumps, Primitive Obsession, Parallel Inheritance Hierarchies, Speculative Generality, Message Chains, Middle Man, Inappropriate Intimacy, Alternative Classes with Different Interfaces, and Incomplete Library Class) were only studied in five papers. These five research papers considered all 22 Code Bad Smells together. This indicates that these Code Bad Smells have not been studied individually in previous studies. We will discuss the implications of this in Section 4.

Table VII. Numbers of investigated Code Bad Smells between Duplicated Code related and non-Duplicated Code related papers.

	Papers investigated Duplicated Code Bad Smell	Paper not investigated Duplicated Code Smell
One Code Bad Smell	13	7
More than one Code Bad Smells	8	11
Total	21	18

Table VIII. Aims of papers.

Code	Aim of studies	Number of papers	Percentage % (Number of papers/ total number of papers)
DC	Method/tools detect Code Bad Smells	19	49
IC	Improve understanding of Code Bad Smells	13	33
RC	Refactor Code Bad Smells	6	15
OT	Others	1	3
Total		39	100

3.2. *RQ 2*: *What are the aims of studies on Code Bad Smells*?

This research question intends to provide insight into the current focus of research in Code Bad Smells and investigate what aspects of Code Bad Smells are most commonly studied. The results of this study are provided in Table VIII. The aims of each of the 39 investigated papers are summarized in Appendix E.

Table VIII shows that

- Nearly half (49%) of the papers aim to enhance tools/methods to detect Code Bad Smells, for example, Munro (SLR 25) proposes a methodology to identify Code Bad Smells from source code using a combination of software metrics, whereas Tourwe and Mens (SLR 33) suggest identifying Code Bad Smells using logic meta programming.
- One third of the papers (13) aim to improve the current understanding of Code Bad Smells. However, only five investigate the impact of Code Bad Smells. These include Shatnawi *et al.*'s (SLR 28) empirical experiment to investigate the association between Code Bad Smells and software faults (these five papers are discussed in more detail in Section 3.4). Other papers include, for example, Mantyla *et al.*'s (SLR 17) taxonomy to classify Fowler and Beck's Code Bad Smells and Mantyla *et al.*'s (SLR 16) investigation of developers' agreement on the existence of Code Bad Smells in software applications.
- Only 15% of the papers (6) focus on enhancing the current knowledge of refactoring Code Bad Smells. These include Counsell *et al.*'s (SLR 3) method to reduce refactoring effects by prioritizing the refactoring order of each of the 22 Code Bad Smells.

Overall, these results suggest that enhancing the tools/methods to detect Code Bad Smells is the most popular aim of research into Code Bad Smells. We will discuss the implications of this skewed research focus in Section 4.

We investigated whether the profile of the research aims are different for different Code Bad Smells. We concentrated on those Code Bad Smells that were studied in more than 20% papers[§] (see Table VI). As a consequence, six Code Bad Smells are investigated. They are the Duplicated Code, Feature Envy, Refused Bequest, Data Class, Long Method and Large Class. In addition, not all the research papers were included in this analysis; we excluded the five papers that analyse

[§]The reason for us only investigating the six Code Bad Smells studied in more than 20% papers is that the numbers were too small for the rest of the Code Bad Smells to make analysis of them meaningful.

Table IX. Distribution of aims of studies for different Code Bad Smells.

	Code Bad Smells											
	Duplicated code		Feature envy		Refused bequest		Data class		Long method		Large class	
Aim of studies	%	*n*	%	*n*	%	*n*	%	*n*	%	*n*	%	*n*
Methods/tools	56	9	29	2	50	3	60	3	0	0	0	0
Improve understanding	25	4	57	4	50	3	40	2	100	3	100	3
Refactor	13	2	14	1	0	0	0	0	0	0	0	0
Others	6	1	0	0	0	0	0	0	0	0	0	0

Table X. Research methods used in studies of Code Bad Smells.

Empirical methods	Number of studies	Percentage (%)
Case study	22	52
Experiment	14	33
Questionnaire/survey	5	12
Non-empirical methods	1	2
Interview	0	0
Observation	0	0
Total	42	100

all 22 Code Bad Smells together because these papers focus on the general features of Code Bad Smells and do not look in detail at the features of individual Code Bad Smells. Consequently, the 27 papers that investigate six different Code Bad Smells are analysed here. These results are presented in Table IX.

Table IX suggests significant variation in the aims of studies of different Code Bad Smells. The studies of the Duplicated Code Bad Smell mainly aim to improve detection methods/tools (56%), far fewer studies aim to improve the understanding of Duplicated Code (25%), and only 13% of studies aim to refactor Duplicated Code. In contrast, over half (57%) of the studies of the Feature Envy Bad Smell aim to improve understanding of it. Our data also show that there are a balanced number of studies (50% each) on the Refused Bequest Bad Smell aiming to improve methods/tools for detecting Refused Bequest and to improve the understanding of Refused Bequest. The main aim of the Data Class Bad Smell studies is to improve the methods/tools for detecting Data Class (60%); also many studies aim to improve the understanding of Data Class (40%), and no study aims to refactor Data Class. The studies of Long Method and Large Class Bad Smells only aim to improve the understanding of these Code Bad Smells, and no other types of studies are reported. The above results show significant differences in the research profiles between different Code Bad Smells. We will discuss this further in Section 4.

3.3. *RQ 3*: *What methods have been used to study Code Bad Smells*?

This analysis will identify whether empirical as opposed to theoretical approaches have been used to investigate Code Bad Smells.

3.3.1. Research methods.

First we indicate the methods used in the studies. In this systematic literature review we define that if two or more studies are conducted with the same goal in a same paper they are treated as a single study. Table X summarizes the results of this analysis. The study methods applied in the 39 investigated papers are presented in Appendix E.

Our results show that in the 42 studies reported in the 39 included papers, 41 have used empirical study methods. These results indicate that empirical methods play an important role in the study of Code Bad Smells. The only non-empirical studies we found were from Counsell *et al.* (SLR 3) which presents a categorization schema for Code Bad Smells based on the features of

Table XI. Source of data in studies of Code Bad Smells.

Source		Number	Percentage % (Number of data/total number of data)
Subjective data	Student opinion	4	8
	Professional opinion	4	8
	Combination	0	0
Objective data	Self-constructed project	4	8
	Academic project	4	8
	Commercial project	13	25
	Student project	2	4
	Open source project	21	40
	Unclear	1	2
Total		53	100
			Total number of empirical studies: 42

their refactoring solutions. Their theoretical taxonomy is introduced but no empirical methods are used to validate it in this paper.

Table X also shows that a high proportion of empirical studies on Code Bad Smells are case studies (52%) or experiments (33%), relatively few are questionnaires/surveys (12%), and none uses interview or observation methods. These results suggest that objective methods are commonly used in the empirical study of Code Bad Smells, with only a few current studies using subjective methods.

3.3.2. Source of data. We review whether the data used in published studies are real world, open source, commercial or academic. Table XI presents the results of this analysis. The types of research data used in each paper are presented in Appendix E.

Table XI shows that 53 data sets[¶] are used in the 42 studies from the 39 included papers. Objective data, such as those from open source projects (40%) and commercial projects (25%), is the main source of research data used in previous studies. Subjective data are seldom used in previous studies (student opinion 8%, professional opinion 8% and no combination opinion data). This confirms our finding that objective methods are the most popularly used methods in empirical studies of Code Bad Smells.

We further investigated the sources of data across the individual Code Bad Smells. Again we focused only on those Code Bad Smells investigated in more than 20% of papers and excluded the five papers that analyse all 22 Code Bad Smells together. Table XII shows the results of this analysis.

Table XII tells us that the studies of the Duplicated Code Bad Smell mainly focus on using open source project data (48%) and commercial project data (26%), in far fewer student opinion data (4%) was used. The studies of the Feature Envy Bad Smell also used a high proportion of open source project data (36%) and commercial project data (18%). However, student opinion (18%) also played an important role in the studies on the Feature Envy Bad Smell. The studies on Refused Bequest Bad Smell are similar to the studies on Duplicated Code Bad Smell. High proportions of open source project data (50%) and commercial project data (25%) were investigated, but no student opinion was used. The studies on Data Class Bad Smell investigated many open source project (43%) and commercial project (29%), but at the same time student opinion (14%) also played an important role. The studies on Long Method Bad Smell used three types of research data, the student opinion data, self-constructed project data and open source project data (33% each). Studies on the Large Class Bad Smell only used open source project data. This is different from other Code Bad Smells. The above results show significant variation between different Code Bad Smells. We will discuss the reasons and implications of this in Section 4.

[¶]Some of these 53 sets of data overlap. For example, data from the Eclipse open source project has been used in several research papers.

Table XII. Distribution of the source of data used in studies of Code Bad Smells with most research attention.

		Code Bad Smell											
		Duplicated code		Feature envy		Refused bequest		Data class		Long method		Large class	
Data type	Data source	%	*n*	%	*n*	%	*n*	%	*n*	%	*n*	%	*n*
Subjective data	Student opinion	4	1	18	2	0	0	14	1	33	2	0	0
	Professional opinion	0	0	0	0	0	0	0	0	0	0	0	0
	Combination	0	0	0	0	0	0	0	0	0	0	0	0
Objective data	Self-constructed project	4	1	18	2	13	1	0	0	33	2	0	0
	Academic project	13	3	9	1	13	1	0	0	0	0	0	0
	Commercial project	26	6	18	2	25	2	29	2	0	0	0	0
	Student project	4	1	0	0	0	0	14	1	0	0	0	0
	Open source project	48	11	36	4	50	4	43	3	33	2	100	3
	unclear	0	0	0	0	0	0	0	0	0	0	0	0

Table XIII. Empirical events of impacts of Code Bad Smells.

Code Bad Smells	Support Fowler *et al.* suggested impact	Against Fowler *et al.* suggested impact
Duplicated Code	Reducing software maintainability (SLR 24)	Some Duplicated Code increases software reliability (SLR 24) Non-trivial number of Duplicated Code can be categorized as good (SLR 9)
Large Class	Statistically associated with software faults (SLR 13, SLR 28)	
Long Method	Statistically associated with software faults (SLR 13, SLR 28)	
Shotgun Surgery	Statistically associated with software faults (SLR 13, SLR 28)	
Data Class		No significant relationship with software faults (SLR 13, SLR 28)
Refused Bequest		No significant relationship with software faults (SLR 13, SLR 28)
Feature Envy		No significant relationship with software faults (SLR 13, SLR 28)

3.4. RQ 4: What evidence is there that Code Bad Smells indicate problems in code?

We investigate the evidence for Fowler *et al.*'s [1] claimed impact of Code Bad Smells. Only five out of the total 39 papers investigated the impact of Fowler *et al.*'s Code Bad Smells. The other 34 papers either investigate methods/tools to detect Code Bad Smells, to refactor Code Bad Smells or investigate other aspects of Code Bad Smells. We summarize the results of papers that investigate the impact of Code Bad Smells in Table XIII.

Overall these five papers examine seven out of the 22 Fowler *et al.*'s [1] Code Bad Smells. Two of these papers are from Shatnawi and Li's research group. Li and Shatnawi's (SLR 13) paper is an extension of Shatnawi and Li (SLR 28) paper, hence we treated these as one study. Furthermore, Srivisut and Muenchaisri's (SLR 29) paper only theoretically identifies the impact of Fowler *et al.*'s Code Bad Smells, and they do not provide any empirical evidence to support their results, as a consequence, we will not analyse this paper in detail.

Monden *et al.* (SLR 24) investigated the impact of the Duplicated Code Bad Smell on software maintainability and reliability. They used maintenance data from an industrial legacy system that has been maintained for 20 years as their research data. Their results show that the Duplicated Code Bad Smell reduces the maintainability of software. This finding supports Fowler and Beck's claim for Code Bad Smells. However Monden *et al.* [3] also found that, in some situations, Duplicated Code can increase the reliability of software. This is an addition to Fowler *et al.*'s claimed impacts.

Kapser and Godfrey (SLR 9) investigated how different types of Duplicated Code Bad Smells affect the quality of software. They identified 11 Duplicated Code patterns in open source projects: Apache and Gnumeric. They invited academic experts to judge whether they considered each piece of Duplicated Code harmful to software. This paper reports that non-trivial amounts of Duplicated Code are a good form of cloning in both the Apache and in Gnumeric projects. As a consequence, the authors suggest that not all Duplicated Code require refactoring. In some situations, the effect of the costs of refactoring, such as effects on program comprehension and exposure to risk, should be measured against the expected gain in maintainability or extendibility of the system. This result indicates that Fowler *et al.*'s claim for Code Bad Smells deserves further investigation.

Shatnawi and Li's (SLR 13, SLR 28) studies examine the association between Code Bad Smells and software faults. They also investigated Code Bad Smells' association with different severity levels of software faults. In these two studies, they use data from an open source project, Eclipse. Their results indicate that the Large Class, Large Method and Shotgun Surgery Bad Smells show significant association with all severity levels of software faults. This supports Fowler *et al.*'s claimed impacts of Code Bad Smells. However Shatnawi *et al.* also found that the other Code Bad Smells, Data Class, Refused Bequest and Feature Envy Bad Smells are not associated significantly with software faults or particular severity levels of software faults. The results of these studies imply that some Code Bad Smells may not lead to faults in software. As a consequence, Fowler *et al.*'s claim for Code Bad Smells deserves further scrutiny.

4. DISCUSSION

In this section we address our research questions and discuss our results in terms of the overall landscape of current Code Bad Smell knowledge. We also discuss the implications of our research for individual Code Bad Smells. In particular, we look at the maturity of the current knowledge about each Code Bad Smell as indicated by our analysis of the literature.

4.1. *RQ1*: *Which Code Bad Smells have attracted the most research attention*?

Our results indicate that the Duplicated Code Bad Smell has attracted the most research attention in the period between 2000 and June 2009. This supports Fowler *et al.*'s [1] view that Duplicated Code is the 'Number One' of all Code Bad Smells. Moreover, our results also suggest that the Duplicated Code Bad Smell tends to be studied alone. We think that the reason for this could be that research on Duplicated Code has been carried out for longer than for other Code Bad Smells. Indeed a lot of Duplicated Code research was published before the identification of Fowler *et al.*'s [1] Code Bad Smells. Consequently, it is not surprising to see that the investigation of Duplicated Code has a unique profile, but it is difficult to know whether this research focus on Duplicated Code means that it is a very important Code Bad Smell in terms of its negative impact on code or whether duplicated code is easy to understand and therefore to study.

However, many Code Bad Smells have received very little research attention. This may be because they are of marginal interest, but it is not clear how common some of these smells are in code. Without more research on these smells it is difficult to know how important they are and how they impact on code. Research is needed even if only to dismiss some Code Bad Smells as not worth worrying about.

4.2. *RQ2*: *What are the aims of studies on Code Bad Smells*?

Our results indicate that studies on Code Bad Smells mainly aim to enhance the tools or methods to detect Code Bad Smells. With only a third of studies aiming to improve the understanding of Code Bad Smells, very few studies look at Code Bad Smells in terms of refactoring. This is surprising because it has been largely argued that refactoring is the main reason for using Code Bad Smells [1, 2]. The overall aim profile of studies suggests that less than expected is known about the impact of Code Bad Smells and their use in refactoring. Instead most of the effort has gone in developing tools and methods to identify Code Bad Smells. Such tools are, of course, a pre-requisite to the research aimed at studying the impact of Code Bad Smells. Thus it may be that this profile of studies will change as tools are developed and made available to researchers wanting to look in more detail at Code Bad Smells. Although given we show that studies on Code Bad Smells have been tailing off in the recent years, it is not obvious that there is a resurgence of research reporting empirical evidence of the impacts of Code Bad Smells.

The aim profile of studies on individual Code Bad Smells varies considerably. For example Duplicated Code has a profile of aims focused on the presentation of methods and tools. Only 25% of studies on Duplicated Code improve the understanding of this smell. Given Duplicated Code is a smell that was recognized before Fowler *et al.*'s work in 1999, it may be that studies improving the understanding of Duplicated Code were carried out before this review. However, it may be that the impact of Duplicated Code is still not properly understood. But that many tools and methods have been developed despite Duplicated Code not being well understood.

Feature Envy, Refused Bequest and Data Class all have a reasonably balanced division of studies aiming to develop methods/tools and to improve understanding. Each has between 40 and 57% of studies focused on improving understanding. This suggests that there is now a basic level of understanding of these smells and that tools and methods are starting to be available to identify them in code. Long Method and Large Class both have all of their studies focused on improving the understanding of these smells. However, there have only been a small number of studies of each of these smells (3 each). This suggests that there is still a long way to go in terms of understanding these smells better and being able to identify them using tools.

4.3. RQ 3: What methods have been used to study Code Bad Smells?

Our findings suggest that nearly all studies on Code Bad Smells are empirical and a high proportion of these are experiments or case studies. Relatively few studies are questionnaires or surveys. This suggests that the focus has been on objective studies with few subjective studies. Although this could be considered a strength in studies, Cushman and Rosenberg [13] suggest that a combination of objective methods and subjective methods are ideally used in studies, as subjective methods provide insights that are not captured by objective methods alone, for example, some software properties cannot be quantified, as a consequence, these properties cannot be captured by using objective methods. This is a gap in the current research of Code Bad Smells.

Our results on the source of research data suggest that commercial projects and open source projects are the two main sources of objective data used in studies of Code Bad Smells. This suggests that, because commercial projects and open source projects exist in the 'real world', findings from these studies are relevant to real projects and are a strength of the work done to date on Code Bad Smells.

Data used to study individual Code Bad Smells vary. Studies on Duplicated Code focus on using open source project data and commercial project data. This suggests that the results of these studies are highly relevant to real-world situations. Similarly, studies on Feature Envy, Refused Bequest and Data Class also predominately use real world open source and commercial project data. Feature Envy and Data Class Bad Smells also have studies using some student opinion data. These subjective data add another dimension to the current knowledge of these smells. Although these data are student based rather than practitioner based, studies on the Long Method Bad Smell use three types of research data: student opinion data, self-constructed project data and open source project data. Given the lack of commercial data current knowledge of this Code Bad Smell is likely to be immature. Studies on the Large Class Bad Smell only use open source project data. Although this is real-world data it is a narrow focus for all studies and reveals another gap in the existing knowledge.

4.4. RQ 4: What evidence is there that Code Bad Smells indicate problems in code?

Mens and Tourwe [2] conclude that Code Bad Smells are the most widely used method of identifying refactoring opportunities. However, only five papers in our sample investigate the impact of using Code Bad Smells, and only four of these provide empirical evidence of Fowler *et al.*'s [1] claimed impact of Code Bad Smells. This indicates that the impact of Code Bad Smells is not empirically understood. However, the reasons for this lack of evidence are hard to explain. It may be because Fowler *et al.*'s [1] claims about the impact of Code Bad Smells are common sense and researchers do not believe there is any value in collecting evidence. Consequently, researchers concentrate on investigating how to identify Code Bad Smells rather than examining their empirical impact. It is also difficult to investigate the impact of Code Bad Smells without mature tools to identify Bad Smells in code. Thus there is a dependency on tools to study the impact and those tools may not be available.

The results of the four papers (SLR 9, SLR 13, SLR 24, SLR 28) investigating the impact of Fowler *et al.*'s [1] Code Bad Smells indicate that not all Code Bad Smells lead to a negative impact on software, e.g., Duplicated Code can increase the reliability of software, and Data Class, Refused Bequest and Feature Envy Bad Smells are not associated significantly with software faults or the particular severity level of software faults, but the reason for this is still unclear. Consequently, further examination of the impact of using Code Bad Smells is necessary.

5. CONCLUSION

In this paper, a systematic literature review has been conducted to investigate the current status of knowledge about Code Bad Smells. Our results indicate that the current level of knowledge for different Code Bad Smells varies. Our findings show that the Duplicated Code Bad Smell has attracted the most attention. This may mean that the Duplicated Code Bad Smell is one of the most important Code Bad Smells and it needs this intensity of investigation. Alternatively, it may be that Duplicated Code has been over-studied as it is easy to understand and therefore to investigate. Duplicated Code also has a research profile different from other Code Bad Smells, in particular it has a smaller proportion of studies that aim to improve the understanding of that smell. Again this could mean that Duplicated Code is easy to understand and needs fewer studies to improve understanding, or alternatively that we know less about Duplicated Code than we should.

Some Code Bad Smells have attracted very few studies, for example, the Message Chains Bad Smell, and consequently, we know very little about these Code Bad Smells. This suggests that our knowledge of some Code Bad Smells remains insufficient and that there is a lack of evidence available on which developers can decide whether to use Code Bad Smells to direct refactoring.

Our results show that we have knowledge based on different types of research data for different Code Bad Smells. Studies are based mainly on open source project data and commercial project data. Other data, such as expert opinion, are rarely used. This suggests that the current studies of Code Bad Smells mainly focus on objective data, and subjective data are rarely used. Since subjective methods provide insights that are not captured by objective methods alone, a more balanced use of objective and subjective data are needed in future studies of Code Bad Smells.

Our results suggest that only a few empirical studies have been conducted to examine the impact of Code Bad Smells. This suggests that the impact of Code Bad Smells remains far from being fully understood. Only five previous studies have directly investigated the impact of Code Bad Smells. Only seven Code Bad Smells were included in these five studies. More studies are needed to investigate the impact of Code Bad Smells, especially for those Smells whose impact has not been previously explored at all.

Overall, our review suggests that very little work has been reported investigating Fowler *et al.*'s claims about the impact of Code Bad Smells. Consequently, there is little evidence currently available justifying the effective and efficient use of Code Bad Smells to direct refactoring. At the moment it is largely unclear what impact smells really have on code and whether it is worth the effort of refactoring to eliminate them. This is an important omission from the current research as we do not know whether the elimination of some Code Bad Smells would lead to improved software quality. At the moment developers have very little evidence on which to effectively target their refactoring effort.

6. LIMITATIONS

There are several limitations in our literature review:

- Only the papers published by IEEE and six leading software engineering journals were investigated in this literature review. These papers may not include all types of studies on Code Bad Smells.
- The search term used in this systematic literature review is limited. Papers only indirectly related to Code Bad Smells are not included in this review. For example, papers about 'Code Clones' may relate to 'Duplicated Code', and 'God Class' studies could also be related to work on 'Large Classes'. However, because we investigate all 22 of Fowler *et al.*'s Code Bad Smells in this systematic literature review, if we consider all possible synonyms of each individual Code Bad Smell, the search space will be huge. It is only practical to investigate papers which address Code Bad Smells directly in this study.
- Some Code Bad Smells were investigated individually before Fowler *et al.* identified Code Bad Smells, such as the Duplicated Code Bad Smell and the Long Method Bad Smell. Because of the limitation of resources, in this literature review only papers after Fowler *et al.*'s identification of Code Bad Smells are investigated; as a result some studies on particular Code Bad Smells may be missed by this study.
- Some differences between Code Bad Smells may just be coincidental. In particular, in this study, the objective data and self-constructed project data may be overrepresented in the studies on Feature Envy and the Long Method Code Bad Smells. This is because we found that objective data and self-constructed project data samples actually are from a single research paper. Thus these results may not represent the common phenomena.

APPENDIX A: A SUMMARY OF DEFINITIONS OF CODE BAD SMELLS

- Duplicated Code

Same code structure happens in more than one place.

- Long Method

A method is too long.

- Large Class

A class is trying to do too much, it often shows up as too many instance variables.

- Long Parameter List

A method needs passing too many parameters.

- Divergent Change

Divergent change occurs when one class is commonly changed in different ways for different reasons.

- Shotgun Surgery

Shotgun surgery is similar to divergent change but is the opposite. Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

- Feature Envy

The whole point of objects is that they are a technique to package data with the processes used on that data. A Feature Envy is a method that seems more interested in a class other than the one it actually is in.

- Data Clumps

Some data items together in lots of places: fields in a couple of classes, parameters in many method signatures.

- Primitive Obsession

Primitive types are overused in software. Small classes should be used in place of primitive types in some situations.

- Switch Statements

Switch statements often lead to duplication. Most times you see a switch statement which you should consider as polymorphism.

- Parallel Inheritance Hierarchies

Parallel inheritance hierarchies is really a special case of shotgun surgery. In this case, every time you make a subclass of one class, you also have to make a subclass of another. You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy.

- Lazy Class

Each class you create costs money to maintain and understand. A class that is not doing enough to pay for itself should be eliminated.

- Speculative Generality

If a machinery was being used, it would be worth it. But if it is not, it is not. The machinery just gets in the way, so get rid of it.

- Temporary Field

Sometimes you see an object in which an instance variable is set only in certain circumstances. Such code is difficult to understand, because you expect an object to need all of its variables.

- Message Chains

You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. Navigating in this way means that the client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change.

- Middle Man

You look at a class's interface and find that half the methods are delegating to this other class. It may mean problems.

- Inappropriate Intimacy

Sometimes classes become far too intimate and spend too much time delving in each others' private parts.

- Alternative Classes with Different Interfaces

Classes are doing similar things but with different signatures.

- Incomplete Library Class

Library classes should be used carefully, especially we do not know whether a library is completed.

- Data Class

These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes.

- Refused Bequest

Subclasses get to inherit the methods and data of their parents, but they just use a few of them.

- Comments

Do not write comments when it is unnecessary. When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

APPENDIX B: SEARCH TERMS

('code bad smell' *OR* 'code smell' *OR* 'bad smell' *OR* 'duplicated code' *OR* 'long method' *OR* 'large class' *OR* 'long parameter list' *OR* 'divergent change' *OR* 'shotgun surgery' *OR* 'feature envy' *OR* 'data clumps' *OR* 'primitive obsession' *OR* 'switch statements' *OR* 'parallel inheritance hierarchies' *OR* 'lazy class' *OR* 'speculative generality' *OR* 'temporary field' *OR* 'message chains' *OR* 'middle man' *OR* 'inappropriate intimacy' *OR* 'alternative classes with different interfaces' *OR* 'incomplete library class' *OR* 'data class' *OR* 'refused bequest' *OR* 'comments' *IN* Title/Abstract)
AND
('software' *IN* All Meta Data)
AND
('education' *OR* 'hardware' *NOT IN* All Meta Data)
AND
('comments on' *OR* 'reply to' *NOT IN* Title)

APPENDIX C: KAPPA INTER-RATER RELIABILITY TEST FOR PHASE ONE ESTIMATION OF INCLUDED PAPERS

Estimation agreement tables between Researchers 1 and 2 and Researchers 1 and 3 are given in Tables CI and CII.

Table CI. Estimation agreement table between Researcher 1 and Researcher 2.

		Researcher 1		
		Accept	Reject	Not sure
Researcher 2	Accept	2	0	0
	Reject	0	6	1
	Not sure	1	0	0

Kappa value = 0.608.

Table CII. Estimation agreement table between Researcher 1 and Researcher 3.

		Researcher 1		
		Accept	Reject	Not sure
Researcher 3	Accept	1	0	0
	Reject	1	6	0
	Not sure	1	0	1

Kappa value = 0.623.

APPENDIX D: A LIST OF CONSIDERED JOURNALS AND CONFERENCE PROCEEDINGS

Index	Name of journals/conference proceedings	Number of investigated papers
IEEE journals/conference proceedings		
1	*IEEE/ACM International Conference on Automated Software Engineering*	4
2	*International Conference on BioMedical Engineering and Informatics*	0
3	*International Symposium on Code Generation and Optimization*	0
4	*IEEE/ACIS International Conference on Computer and Information Science*	1
5	*Annual IEEE International Computer Software and Applications Conference*	1
6	*Mediterranean Conference on Control and Automation*	0
7	*International Symposium on Empirical Software Engineering*	1
8	*IEEE Congress on Evolutionary Computation*	0
9	*International Conference on Information Technology*	1
10	*IEEE Instrumentation and Measurement Magazine*	0
11	*International Conference on Machine Learning and Cybernetics*	0
12	*IEEE International Workshop on Mining Software Repositories*	1
13	*IEEE International Symposium on Modern Computing*	0
14	*IEEE Transactions on Neural Networks*	0
15	*IEEE International Conference on Program Comprehension*	1
16	*International Conference on Research Challenges in Information Science*	0
17	*Working Conference on Reverse Engineering*	4
18	*IEEE International Conference on Software Engineering*	1
19	*IEEE Transactions on Software Engineering*	0
20	*International Workshop on Principles of Software Evolution*	3
21	*International IEEE Workshop on Software Evolvability*	0
22	*European Conference on Software Maintenance and Reengineering*	6
23	*IEEE International Conference on Software Maintenance*	5
24	*IEEE International Symposium on Software Metrics*	2
25	*International Conference on Software Testing Verification and Validation*	0
26	*IEEE Software*	0
27	*IEEE International Workshop on Source Code Analysis and Manipulation*	1
28	*International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*	1
29	*Testing*: *Academic and Industrial Conference—Practice And Research Techniques*	1
30	*IEEE Symposium on Visual Languages and Human Centric Computing*	1
Other journals		
31	*Journal of Systems and Software*	1
32	*Empirical Software Engineering Journal*	3
33	*Information and Software Technology Journal*	0
34	*Journal of Software Maintenance and Evolution*: *Research and Practice*	0
35	*ACM Transactions on Software Engineering Methodology*	0
36	*Software*: *Practice and Experience Journal*	0

APPENDIX E: A LIST OF INCLUDED PAPERS AFTER THE PHASE TWO REFINEMENT OF INCLUDED PAPERS

Ref	Paper reference	Code Bad Smells	Aims	Study method	Source of research data	Impacts
SLR 1	Arevalo G, Ducasse S and Nierstrasz O. Discovering unanticipated dependency schemas in class hierarchies. *Ninth European Conference on Software Maintenance and Reengineering* (*CSMR 2005*), 2005; 62–71.	Refused Bequest, others	DC	Case study x1	Commercial project x1	N
SLR 2	Casazza G, Antoniol G, Villano U, Merlo E and Di Penta M. Identifying clones in the Linux kernel. *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001; 90–97.	Duplicated Code	IC	Case study x1	Open source x1	N
SLR 3	Counsell S, Hierons RM, Najjar R, Loizou G and Hassoun Y. The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. *Proceedings of Testing*: *Academic and Industrial Conference—Practice and Research Techniques* (*TAIC PART 2006*), 2006; 181–192.	All	RC	Non-empirical x1	N/A	N
SLR 4	Eichberg M, Haupt M, Mezini M and Schafer T. Comprehensive software understanding with SEXTANT. *Proceedings 21st IEEE International Conference on Software Maintenance* (*ICSM'05*), 2005; 315–324.	Others	DC	Case study x1	Academic project x1, unclear x1	N
SLR 5	Elssamadisy A and Schalliol G. Recognizing and responding to 'bad smells' in extreme programming. *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), 2002; 617–622.	Others	IC	Case study x1	Commercial project x1	N
SLR 6	van Emden E and Moonen L. Java quality assurance by detecting code smells. *Proceedings of the Ninth Working Conference on Reverse Engineering*, 2002; 97–106.	Others	RC	Case study x1	Commercial project x1	N
SLR 7	Fokaefs M, Tsantalis N and Chatzigeorgiou A. JDeodorant: identification and removal of feature envy Bad Smells. *IEEE International Conference on Software Maintenance* (*ICSM 2007*), 2007; 519–520.	Feature Envy	RC	Experiment x1	Open source x1	N
SLR 8	Hill R and Rideout J. Automatic method completion. *Proceedings of the 19th International Conference on Automated Software Engineering*, 2004; 228-235.	Duplicated Code	DC	Case study x1	Commercial project x1, open source x1	N

SLR 9	Kapser C and Godfrey M. 'Cloning considered harmful' considered harmful: patterns of cloning in software. *Empirical Software Engineering* 2008; 13(6):645–692.	Duplicated Code	IC	Case study x1	Open source x1	Y
SLR 10	Kapser C and Godfrey MW. Aiding comprehension of cloning through categorization. *Proceedings of the Seventh International Workshop on Principles of Software Evolution*, 2004; 85–94.	Duplicated Code	DC	Case study x1	Open source x1	N
SLR 11	Kapser C and Godfrey MW. 'Cloning Considered Harmful' considered harmful. *13th Working Conference on Reverse Engineering* (*WCRE '06*), 2006; 19–28.	Duplicated Code	IC	Case study x1	Open source x1	N
SLR 12	Kiefer C, Bernstein A and Tappolet J. Mining software repositories with iSPAROL and a software evolution ontology. *Fourth International Workshop on Mining Software Repositories* (*ICSE Workshops MSR '07*), 2007; 10.	Long Parameter List, others	DC	Case study x1	Open source x1	N
SLR 13	Li W and Shatnawi R. 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): 1120–1128.	Long Method, Large Class, Shotgun Surgery, Feature Envy, Data Class, Refused Bequest	IC	Experiment x1	Open source x1	Y
SLR 14	Mantyla MV. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. *2005 International Symposium on Empirical Software Engineering*, 2005; 10.	Long method, Long Parameter List, Feature Envy	IC	Questionnaires/ surveys x2	Student opinion x2, constructed project x2	N
SLR 15	Mantyla, M and Lassenius, C. Subjective evaluation of software evolvability using code smells: an empirical study. *Empirical Software Engineering* 2006; 11(3):395–431.	All	IC	Questionnaires/ surveys x1	Professional opinion x1	N
SLR 16	Mantyla MV, Vanhanen J and Lassenius C. Bad smells—humans as code critics. *Proceedings 20th IEEE International Conference on Software Maintenance*, 2004; 399–408.	All	IC	Experiment x1, questionnaires/ surveys x1	Professional opinion x2, commercial project x1	N
SLR 17	Mantyla M, Vanhanen J and Lassenius C. A taxonomy and an initial empirical study of bad smells in code. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2003*), 2003; 381–384.	All	IC	Questionnaires/ surveys x1	Professional opinion x1	N
SLR 18	Marcus A and Maletic JI. Identification of high-level concept clones in source code. *Proceedings of the 16th Annual International Conference on Automated Software Engineering* (*ASE 2001*), 2001; 107–114.	Duplicated Code, comments	DC	Experiment x1	Constructed project x1	N

SLR 19	Marinescu C. Identification of design roles for the assessment of design quality in enterprise applications. *14th IEEE International Conference on Program Comprehension* (*ICPC 2006*), 2006; 169–180.	Feature Envy, Data Class	DC	Experiment x2	Commercial project x2	N
SLR 20	Mens T, Tourwe T and Munoz F. Beyond the refactoring browser: advanced tool support for software refactoring. *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, 2003; 39–44.	Duplicated Code, Refused Bequest, others	DC	Case study x1	Academic project x1, commercial project x1, open source x1	N
SLR 21	Merlo E, Antoniol G, Di Penta M and Rollo VF. Linear complexity object-oriented similarity for clone detection and software evolution analyses. *Proceedings 20th IEEE International Conference on Software Maintenance*, 2004; 412–416.	Duplicated Code	DC	Case study x1	Open source x1	N
SLR 22	Mihancea PF and Marinescu R. Towards the optimization of automatic detection of design flaws in object-oriented software systems. *Ninth European Conference on Software Maintenance and Reengineering* (*CSMR 2005*), 2005; 92–101.	Data Class, others	DC	Experiment x1	Student opinion x1, student project x1	N
SLR 23	Moha N, Gueheneuc YG and Leduc P. Automatic generation of detection algorithms for design defects. *21st IEEE/ACM International Conference on Automated Software Engineering* (*ASE '06*), 2006; 297–300.	Others	DC	Case study x1	Open source x1	N
SLR 24	Monden A, Nakae D, Kamiya T, Sato S and Matsumoto K. Software quality analysis by code clones in industrial legacy software. *Proceedings Eighth IEEE Symposium on Software Metrics*, 2002; 87–94.	Duplicated Code	IC	Experiment x1	Commercial project x1	Y
SLR 25	Munro MJ. Product metrics for automatic identification of 'bad smell' design problems in Java source-code. *11th IEEE International Symposium on Software Metrics*, 2005; 9.	Lazy Class, Temporary Field	DC	Case study x1	Commercial project x1	N
SLR 26	Rapu D, Ducasse S, Girba T and Marinescu R. Using history information to improve design flaws detection. *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering* (*CSMR 2004*), 2004; 223–232.	Data Class, others	DC	Case study x1	Open source x1	N
SLR 27	Rieger M, Ducasse S and Lanza M. Insights into system-wide code duplication. *Proceedings of the11th Working Conference on Reverse Engineering*, 2004; 100–109.	Duplicated Code	RC	Case study x1	Academic project x1, commercial project x1, student project x1 open source x1	N

SLR 28	Shatnawi R and Wei L. An investigation of Bad Smells in object-oriented design. *Third International Conference on Information Technology*: *New Generations* (*ITNG 2006*), 2006; 161–165.	Long Method, Large Class, Shotgun Surgery, Feature Envy, Data Class, Refused Bequest	IC	Experiment x1	Open source x1	Y
SLR 29	Srivisut K and Muenchaisri P. Bad-Smell metrics for aspect-oriented software. *Sixth IEEE/ACIS International Conference on Computer and Information Science* (*ICIS 2007*), 2007; 1060–1065.	Duplicated Code, Feature Envy, others	DC	Case study x1	Academic project x1	Y
SLR 30	Srivisut K and Muenchaisri P. Defining and detecting Bad Smells of aspect-oriented software. *31st Annual International Computer Software and Applications Conference* (*COMPSAC 2007*), 2007; 65–70.	Others	IC	Experiment x1	Open source x1	N
SLR 31	Tairas R and Gray J. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering* 2009; 14(1):33–56.	Duplicated Code	DC	Case study x1	Commercial project x1	N
SLR 32	Toomim M, Begel A and Graham SL. Managing duplicated code with linked editing. *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, 2004; 173–180.	Duplicated Code	RC	Experiment x1	Student opinion x1	N
SLR 33	Tourwe T and Mens T. Identifying refactoring opportunities using logic meta programming. *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, 2003; 91–100.	Refused Bequest, others	DC	Case study x1	Constructed project x1	N
SLR 34	Trifu A and Marinescu R. Diagnosing design problems in object oriented systems. *12th Working Conference on Reverse Engineering*, 2005; 10.	Large Class, Feature Envy, Refused Bequest, others	IC	Case study x1	Open source x1	N
SLR 35	Trifu A and Reupke U. Towards automated restructuring of object oriented systems. *11th European Conference on Software Maintenance and Reengineering* (*CSMR '07*), 2007; 39–48.	All	RC	Experiment x1	Open source x1	N
SLR 36	Tsantalis N, Chaikalis T and Chatzigeorgiou A. JDeodorant: identification and removal of type-checking bad smells. *12th European Conference on Software Maintenance and Reengineering* (*CSMR 2008*), 2008; 329–331.	Switch statements	DC	Experiment x1	Open source x1	N
SLR 37	Van Rysselberghe F and Demeyer S. Evaluating clone detection techniques from a refactoring perspective. *Proceedings of the 19th International Conference on Automated Software Engineering*, 2004; 336–339.	Duplicated Code	DC	Experiment x1	Commercial project x1, open source x1	N

SLR 38	Van Rysselberghe F and Demeyer S. Reconstruction of successful software evolution using clone detection. *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, 2003; 126–130.	Duplicated Code	OT	Case study x1	Open source x1	N
SLR 39	Wettel R and Marinescu R. Archeology of code duplication: recovering duplication chains from small duplication fragments. *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (*SYNASC 2005*), 2005; 8.	Duplicated Code	DC	Case study x1	Open source x1	N

ACKNOWLEDGEMENTS

This systematic literature review could not be carried out without the kind help from Mrs Pam Green and Dr Paul Wernick from the School of Computer Science University of Hertfordshire. Pam helped us to conduct inter-rater reliability tests to define classification schemas for this systematic literature review. Paul gave us useful suggestions throughout this study.

REFERENCES

1. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring*: *Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999.
2. Mens T, Tourwe T. A survey of software refactoring. *IEEE Transactions on Software Engineering* 2004; **30**(2):126–139.
3. Monden A, Nakae D, Kamiya T, Sato S, Matsumoto K. Software quality analysis by code clones in industrial legacy software. *Proceedings Eighth IEEE Symposium on Software Metrics*, 2002.
4. Shatnawi R, Li W. An investigation of bad smells in object-oriented design. *Third International Conference on Information Technology*: *New Generations*, *ITNG*, 2006; 161–165.
5. Kitchenham B. Procedures for performing systematic reviews. *TR/SE-0401*, Keele University and National ICT Australia Ltd., 2004; 1–28.
6. Zhang M, Hall T, Baddoo N. A review protocol for the impact of code bad smells systematic literature review. *Technical Report 496*, STRI, University of Hertfordshire, Hatfield, 2009.
7. Glass RL, Vessey I, Ramesh V. Research in software engineering: an analysis of the literature. *Information and Software Technology* 2002; **44**(8):491–506.
8. Sjoeberg DIK, Hannay JE, Hansen O, Kampenes VB, Karahasanovic A, Liborg NK, Rekdal AC. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* 2005; **31**(9):733–753.
9. Bouqure P, Dupuis R, Albran A, Moore JW. *Guide to the Software Engineering Body of Knowledge 2004*. IEEE Computer Society: Los Alamitos, California, 2004.
10. Glaser BG, Strauss AL. *The Discovery of Grounded Theory*: *Strategies for Qualitative Research*. Aldine Publishing Co.: New York, 1967.
11. Saunders MNK, Lewis P, Thornhill A. *Research Methods for Business Students* (3rd edn). Financial Times Prentice Hall: Harlow, xvii, 2003; 15–504.
12. Landis J, Koch G. Measurement of observer agreement for categorical data. *Biometrics* 1977; **33**:159–174.
13. Cushman WH, Rosenberg DJ. *Human Factors in Product Design*. Elsevier: Amsterdam, 1991.

AUTHORS' BIOGRAPHIES

Mr Min Zhang is a PhD candidate in the School of Computer Science at the University of Hertfordshire, U.K. He is also a member of the Systems and Software Research (SSR) group at the University of Hertfordshire. His current PhD research investigates Code Bad Smells in relation to fault proneness. His other research interests include Design Patterns, Refactoring, software metrics and open source software. He received an MSc software engineering degree from the University of Hertfordshire. He worked as a senior system developer in PCCW China.

Dr Tracy Hall is a Reader in Software Engineering at Brunel University. Previously she was Head of the Systems & Software Research Group at the University of Hertfordshire and Adjunct Professor of Industrial Systems at the University of Oslo. Dr Hall's expertise is in Empirical Software Engineering research. Over the last 15 years she has conducted many empirical software engineering studies with a variety of industrial collaborators. Dr Hall has reported on the efficacy of various software engineering techniques and is currently investigating the use of program slicing metrics in relation to fault proneness and maintainability. She has particular interest in the human aspects of software engineering and has successfully led previous projects investigating the motivations of software engineers. She has published nearly 100 international peer reviewed journal and conference papers.

Dr Nathan Baddoo is a Principal Lecturer in the School of Computer Science at the University of Hertfordshire. He is also a member of the Systems and Software Research (SSR) group at the University of Hertfordshire. His research focuses on the relationship between developer motivation and software quality, Software Process Improvement (SPI) and software project performance. Nathan has expertise in focus group discussions and Repertory Grid Technique interviews, and has applied novel data collection and analysis techniques such as Multi Dimensional Scaling. Nathan is also exploring how to construct prediction models of software engineers' motivation by using concepts like agent-based simulation.