Evaluating the Lifespan of Code Smells using Software Repository Mining

Ralph Peters
Delft University of Technology
The Netherlands
Email: ralphpeters85@gmail.com

Andy Zaidman
Delft University of Technology
The Netherlands
Email: a.e.zaidman@tudelft.nl

***Abstract*—An anti-pattern is a commonly occurring solution to a recurring problem that will typically negatively impact code quality. Code smells are considered to be symptoms of anti-patterns and occur at source code level. The lifespan of code smells in a software system can be determined by mining the software repository on which the system is stored. This provides insight into the behaviour of software developers with regard to resolving code smells and anti-patterns. In a case study, we investigate the lifespan of code smells and the refactoring behaviour of developers in seven open source systems. The results of this study indicate that engineers are aware of code smells, but are not very concerned with their impact, given the low refactoring activity.**

I. Introduction

Software evolution can be loosely defined as the study and management of the process of repeatedly making changes to software over time for various reasons [1]. In this context Lehman [1] has observed that change is inevitable if a software system wants to remain successful. Furthermore, the *successful* evolution of software is becoming increasingly critical, given the growing dependence on software at all levels of society and economy [2].

Unfortunately, changes to a software system sometimes introduce inconsistencies in its design, thereby invalidating the original design [2] and causing the structure of the software to degrade. This structural degradation makes subsequent software evolution harder, thereby standing in the way of a successful software product.

While many types of inconsistencies can possibly be introduced into the design of a system (e.g., unforeseen exception cases and conflicting naming conventions), this study focuses on a particular type of inconsistency called an anti-pattern. An *anti-pattern* is defined by Brown et al. [3] as a commonly occurring solution that will always generate negative consequences when it is applied to a recurring problem. Detection of anti-patterns typically happens through code smells, which are symptoms of anti-patterns [4]. Examples include god classes, large methods, long parameter lists and code duplication [5].

In this study we investigate the lifespan of several code smells. In order to do so, we follow a software repository mining approach, i.e., we extract (implicit) information from version control systems about how developers work on a system [6]. In particular, for each code smell we determine when the infection takes place, i.e., when the code smell is introduced and when the underlying cause is refactored.

Having knowledge of the lifespans of code smells, and thus which code smells tend to stay in the source code for a long time, provides insight into the perspective and awareness of software developers on code smells. Our research is steered by the following research questions:

- RQ1 Are some types of code smells refactored more and quicker than other smell types?
- RQ2 Are relatively more code smells being refactored at an early or later stage of a system's life cycle?
- RQ3 Do some developers refactor more code smells than others and to what extent?
- RQ4 What refactoring rationales for code smells can be identified?

The structure of this paper is as follows: Section II provides some background, after which Section III provides details of the implementation of our tooling. Section IV presents our case study and its results. Section V discusses threats to validity, while Section VI introduces related work. Section VII concludes this paper.

II. Background

This section provides theoretical background information on the subjects related to this study.

A. Code Smells

There is no widely accepted definition of code smells. In the introduction, we described code smells as symptoms of a deeper problem, also known as an anti-pattern. In fact, code smells can be considered anti-patterns at programming level rather than design level. Smells such as large classes and methods, poor information hiding and redundant message passing are regarded as bad practices by many software engineers. However, there is some subjectivity to this determination. What developer A sees as a code smell may be considered by developer B as a valuable solution with acceptable negative side effects. Naturally, this also depends on the context, the programming language and the development methodology.

The interpretation most widely used in literature is the one by Fowler [5]. He sees a code smell as a structure that needs

to be removed from the source code through refactoring to improve the maintainability of the software. He also claims that informed human intuition is the best tool to label a piece of source code as a code smell and measure its intensity. This is a plausible statement, but it does not render automatic measurement impossible or redundant. Most code smells can be measured by using *software metrics*.

In this study we focus on the following code smells:

- God Class [7] — abbreviated as *GC*
- Feature Envy [8] — *FEM*
- Data Class [5] — *DC*
- Message Chain Class [5] — *MCC*
- Long Parameter List Class [5] — *LPLC*

B. Code Smell Detection Tools

Code smells can be detected manually, but this requires the software engineer to have an experienced eye and usually a deeper knowledge of the system. Extensive research has been devoted to develop techniques to do this automatically. Most code smell detection tools depend on the use of software metrics and corresponding thresholds for these metrics. Some important detection tools are *JDeodorant*, developed by Tsantalis [9], and *Ptidej*, developed by Guéhéneuc and his team [10].

C. Mining Software Repositories

The term *Mining Software Repositories* (MSR) has been coined to describe a broad class of investigations into the examination of software repositories. This includes sources such as the information stored in source code version control systems (e.g., SVN), requirements and bug-tracking systems (e.g., Bugzilla) and communication archives (e.g., e-mail) [6]. Such repositories contain a wealth of information and provide a unique view of the actual evolutionary path taken to realize a software system. It is this information that we intend to use for determining the lifespan of code smells.

III. SACSEA Implementation

SACSEA, which is an acronym for *Semi-Automatic Code Smell Evolution Assistant*, is our tool for studying code smells. It follows a repository mining approach to determine the lifespans of code smells and it reuses the aforementioned code smell detectors JDeodorant and Ptidej to detect smells in individual revisions of the software project. SACSEA generates a textual report containing the lifespans of each code smell instance. Also, metadata of their beginning and end revisions are printed in text, such as the developer responsible for the commit that introduced or removed the code smell and the commit date. Figure 1 shows an overview of the approach that we implemented in SACSEA.

The SACSEA approach can be seen as a 4-step process:

Step 1 – Initialization. The user specifies the URL of the SVN repository and selects the code smell to be analysed in the run (currently, SACSEA can only analyse one code smell per run). The user should also specify a start and end revision.

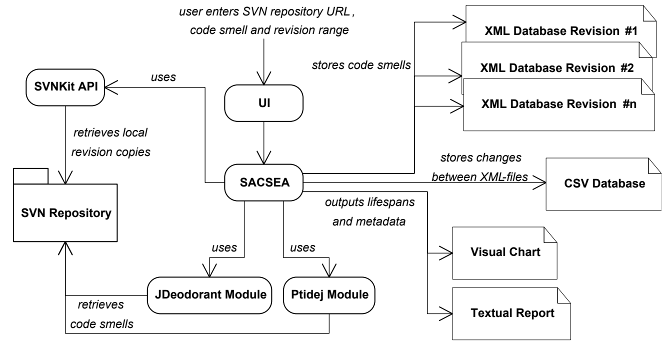

Figure 1. Overview of the operation of SACSEA.

Step 2 – Detection. SACSEA checks out the first revision (specified by the user) from the SVN repository as a working copy. This copy is imported into Eclipse as a Java project using the *.classpath* and *.project* files inside the root directory. If these files are not present, then the file *pom.xml* must be present. The project is then built by *Maven* [11]. If the root directory does not contain a *pom.xml* file or a combination of *.classpath* and *.project* files, then the Eclipse instance cannot import this as a Java project and the working copy is immediately deleted from the local disk and not considered for code smell detection. SACSEA will continue with the next revision. If the revision can be built, then either JDeodorant or Ptidej will start to detect code smells in the current revision. After the detection on a single revision is complete, the results are stored in an XML-database. The detection process now repeats itself by checking out the subsequent revision (if available).

Step 3 – Difference Computation. When all revisions have been examined, there exists an XML-database for each of them, containing the code smells that were found. Subsequently, every pair of consecutive revisions is compared for differences in code smell instances, thus enabling to determine the lifespans of code smell instances. The results of this analysis are stored in a CSV-file that contains all the introductions and removals of code smells in classes and methods in certain revisions, which respectively correspond with the beginning and end revisions of the lifespan of an infected instance.

Step 4 – Output Generation. A textual report is made that contains each smell instance and its lifespan, along with metadata, such as the commit date and the developer who made the commit. Optionally, a graphical representation of the lifespans of code smells is generated (see [12, p. 33]).

IV. Case Study

In order to answer the research questions that we introduced in Section I, we use SACSEA to examining the lifespans of five types of code smells in seven projects.

System	Strength	Development process	Revisions analysed	Active developers	# classes	Investigation period
CalDAV4j	Industrial	Strict	318	5	125	10/2007–03/2011
Evolution Chamber	OS	Strict	282	11	481	10/2010–03/2011
JDiveLog	OS	Loose	872	13	331	03/2005–03/2011
jGnash	OS	Loose	1493	1	466	12/2007–02/2011
Saros	Industrial	Strict	2482	26	821	09/2006–09/2010
VLCJ	Industrial	Loose	1502	1	241	05/2009–04/2011
Vrapper (Base)	OS	Loose	115	2	119	12/2008–04/2009
Vrapper (Core branch)	OS	Loose	231	2	229	04/2009–04/2010

Table I
OVERVIEW OF THE SYSTEMS UNDER INVESTIGATION.

A. Experimental Setup

The seven software projects that we used in our case study are listed in Table I. The selection of these projects was subject to multiple criteria. First, the systems need to be written in Java, because the detection modules of SACSEA cannot process other programming languages. Moreover, they should allow free access to their repository on Subversion. A project also has to be in a mature development stage, meaning that it has to contain enough analysable revisions from which code smell lifespans of significant size can be determined. Furthermore, we made sure that they were sufficiently diverse in terms of domain, strength and strictness of the development process[1].

SACSEA determines the lifespans of any code smell instances found in each of these software systems. Subsequently, we derive statistics from these data, which are then used to help answer the research questions.

B. Results

Based on the results that we obtained from running SACSEA on the seven software systems, we are now in a position to answer the research questions.

RQ1 *Are some types of code smells refactored more and quicker than other smell types?*

In order to answer this question, we compare the average lifespans of different code smell types found in the subject systems. Table II shows the average lifespan, or the sum of the lifespans of all code smell instances detected in each system divided by the total number of these instances, in italic. All values are expressed in number of revisions and have been rounded to the nearest integer. Because the analysis of some smell types included a different number of revisions than other types in the same subject system, we also present this average lifespan expressed as a percentage. This discrepancy in the number of revisions analysed is due to the use of multiple detection tools, some of which were unable to process all revisions. This percentage is the average lifespan in revisions divided by the total number of analysed revisions per smell type, multiplied by 100.

[1]Strictness of the development process relates to whether developers have to abide by development rules, such as guidelines for programming and committing. Usually, this is reflected by development manuals.

	GC	FEM	DC	MCC	LPLC
CalDAV4j	42.38%	22.39%	54.55%	52.52%	66.52%
	135	*71*	*173*	*167*	*212*
Evolution Chamber	51.40%	30.27%	27.52%	29.68%	46.08%
	145	*85*	*78*	*84*	*130*
JDiveLog	54.37%	36.71%	51.28%	35.91%	42.67%
	419	*320*	*447*	*313*	*372*
jGnash	67.81%	60.83%	81.97%	70.78%	68.66%
	883	*792*	*810*	*669*	*1025*
Saros	32.56%	24.31%	24.24%	26%	34.29%
	680	*566*	*602*	*643*	*851*
VLCJ	35.46%	29.11%	67.03%	31.58%	35.98%
	533	*421*	*1007*	*474*	*541*
Vrapper (Base)	60.82%	47.58%	72.92%	72.73%	53.48%
	70	*55*	*84*	*84*	*62*
Vrapper (Core branch)	48.79%	71.90%	66.02%	57.14%	47.62%
	113	*160*	*153*	*132*	*110*
Total average	49.20%	40.39%	55.69%	47.04%	49.41%
Standard deviation	12.10%	18.08%	20.79%	18.76%	12.81%

Table II
AVERAGE LIFESPANS IN PERCENTAGE.

The differences between the average lifespans over all subject systems are small. Nevertheless, something can still be said about the results. The Feature Envy Method smell instances have the shortest lifespan on average. The highest lifespan can be found in the core branch of Vrapper, however it has only three Feature Envy Methods. Looking a bit deeper, we also see that most infected methods start suffering from this smell after several revisions. Also, the majority of the instances are removed relatively quickly.

RQ2 *Are relatively more code smells being refactored at an early or later stage of a system's life cycle?*

Here, the same approach as mentioned for RQ1 applies. The only difference is that just the first 20% and the last 20% of the examined revisions of the software projects are considered. Thus, each system will have two percentages per smell type, representing the average lifespan of smell instances in the earliest and latest investigated revisions. The average of the percentages of all systems is calculated, resulting in two percentages per smell type: the average lifespan of the earliest revisions and the average lifespan of the latest revisions over all subject systems.

	Youngest 20%					Latest 20%				
	GC	FEM	DC	MCC	LPLC	GC	FEM	DC	MCC	LPLC
CalDAV4j	52	0	0	0	0	56	45	64	64	64
Evolution Chamber	49	32	1	25	26	54	51	55	57	52
JDiveLog	102	94	73	73	99	132	151	165	154	166
jGnash	134	138	72	60	173	250	249	197	176	280
Saros	226	208	246	192	298	274	343	401	417	414
VLCJ	107	115	194	173	0	230	205	301	203	301
Vrapper (Base)	18	14	20	19	12	20	23	21	23	23
Vrapper (Core branch)	27	0	40	38	47	35	45	46	46	39

Table III
AVERAGE LIFESPANS WITHIN THE YOUNGEST/LATEST 20% IN TERMS OF REVISIONS.

According to Table III, CalDAV4j seems to have no smell instances for almost all smell types in its early revisions. This is due to the fact that those revisions were unexpectedly not analysable. Nevertheless, the overall results show a clear pattern for all smell types. In particular, for all subject systems and smell types, the first 20% of the examined revisions reveal a substantially lower average lifespan than the last 20%. Closer inspection revealed that (1) infected instances at the beginning of a system's life cycle are bound to be refactored within a few revisions and (2) the number of long-living code smell instances increases over time. This can be explained by the fact that as a system expands, it will contain more classes and methods that may or may not get infected. The results hint towards the lack of concern or awareness of the developers regarding code smells in general. A more thorough investigation is needed in order to strengthen the validity of this presumption.

RQ3 *Do some developers refactor more code smells than others and to what extent?*

The goal is to count the number of times a code smell instance of a certain smell type is resolved. The name of the responsible developer is stored whenever he or she performs a corresponding activity. This will result in a list of developers and per smell type the number of instances they removed. However, there are various reasons for removing a smell. Intentional refactorings must be distinguished from removals that were the side effect of bug fixes or the renaming or deletion of entire classes or methods.

The approach here is to count the number of removed code smell instances per developer. Table IV shows the names of the developers participating in the subject systems with the number of infected instances that they removed according to SACSEA. Note that one commit can contain multiple removed smell instances, which can be the consequence of moving functionality across classes and methods.

However, not all removals are the result of intentional, dedicated refactoring activities. The number of resolved instances in the left side of Table IV must be reduced by the number of times a coincidental removal occurred. For this, the log messages of all commits responsible for a smell removal have been manually examined and categorized based on the cause of the removal. Signal words like "*Refactored*", "*Extracted class*" and "*Clean-up*" are usually indications of true refactoring activities. Naturally, these words are no guarantee and a developer's perspective on the manifestation of code smells may differ from the ones assumed by the detection tools we used in our study. The right side of Table IV shows the names of the developers with the amount of resolved instances that were the consequence of deliberate refactoring per code smell. Also, the last column in both tables shows the total number of commits per developer within the range of analysed revisions.

		All					Intentional					
	Developer	GC	FEM	DC	MCC	LPLC	GC	FEM	DC	MCC	LPLC	Commits
CalDAV4j	bobbyrullo	7	1	1	1							*78*
	robipolli	18	17	2	1	1	2			1		*111*
Evolution Chamber	nafets.st	8	61		6		1					*15*
	bdurrer	2	8	2	1		2	7	2			*15*
	domagala.lukas	1	4	1	52	4					1	*36*
	mike.angstadt		3									*41*
	fritley		3	3	1			1				*70*
	brendan.speer		1	1		1						*4*
	netprobe		1									*16*
JDiveLog	onlinervolker	13	49		2							*93*
	andre_schenk	5	8	4		1						*107*
	vkorecky	12	37		3							*48*
	sjomik	1										*2*
	szdavid1	3	1	1								*52*
	pellmont	46	53	3	3	1						*456*
	Levtraru		2									*3*
jGnash	ccavanaugh	196	246	2	26	17		1				*1355*
Saros	sotitas	6	4	1	1	1						*7*
	chris_fu	13	15	3	4	7	1					*141*
	coezbek	108	128	4	19	13	3	1			1	*784*
	Arbosh	1										*6*
	k_beecher	2	7		1	2		7			1	*31*
	wojtus	1	13			6						*1*
	hstaib	1										*9*
	marrin	33	58	3	25	4	1					*130*
	orieger	19	20	2	7	3						*119*
	ahaferburg		10	55		1						*69*
	s-ziller	8	5					1				*63*
	starkmann	1										*3*
	testvogel	1										*4*
	szuecs	5	2		1	1						*14*
	ldohrmann	7	2									*15*
	djemili	1	1									*43*
	marcus-fu		1									*3*
	ornis			2		1						*27*
VLCJ	wm.mark.lee	39	15	13	4	2	3	3		1		*878*
Vrapper (Base)	weissi	2										*6*
	waweee	9	2	3	5	1	1			2		*102*
Vrapper (Core)	kgoj	6		4	1	1			1			*75*
	waweee	15		2	7							*81*

Table IV
NUMBER OF CODE SMELL REMOVALS, AS FOUND BY SACSEA.

RQ4 *What refactoring rationales for code smells can be identified?*

To answer this question, the log messages of all commits responsible for the removal of a smell are examined. Similar to the approach for RQ3, deliberate refactoring activities must be identified. Ideally, the log messages are clear and representative for the actual changes. Unfortunately, this is not always the case. Furthermore, due to the scarcity of intentional refactoring activities in the subject systems, finding rationales is difficult. Therefore, code styling rationales are also considered here, such as dead code elimination. The most common rationales for resolving the smells considered in this case study are listed below and have been derived from the log messages and source code inspection.

- *Cleaning up dead or obsolete code*: Many subject systems contain a few revisions in which duplicate, unused or old classes and methods are removed. Occasionally, this results in the removal of a smell instance, albeit accidental. Some of these activities may not be considered as true refactorings.

- *Dedicated refactoring*: Similar to dead code elimination, there are some cases in which developers refactor for the sole purpose of refactoring. This often comes down to restructuring libraries (Data Classes) or generalizing large classes through the use of interfaces. The question arises whether the developers are aware of the specific code smell that infects a certain software entity.
- *Maintenance activities*: The majority of the refactorings are coincidental, as a side effect of intentional bug fixes or implementing new functionality. This causes many classes and methods to be removed, including the infected instances.

V. Threats to Validity

This section describes the aspects that may threaten the validity of this study.

Internal Validity. The two code smell detection tools that we incorporated in SACSEA may identify code smell instances that are not considered as such by a human expert. As a result, code smells may not be subject to a refactoring. False negatives may also occur for similar reasons, in which case the code smell instance is not shown in the results at all. This threat is slightly reduced by using two tools with different detection approaches.

JDeodorant, one of the code smell detectors that we used, uses an AST-based approach, which needs parsable Java source files. There are some revisions in the subject systems that do not compile. Depending on the responsible detection utility, the source file or the entire revision is skipped for detection. Consequently, there is no useful data available for some revisions of four case study systems. The percentage of unparsable revisions varies from 1% to 25%. This affects the results and in order to keep the effect to a minimum we carefully selected software systems with the smallest number of unparsable revisions.

SACSEA is unable to determine if an entity in revision n has been renamed in revision n+1. This will usually be shown in the output as a smell instance ceasing to exist in one revision and another smell instance being introduced in the subsequent revision. This phenomenon has to be recognized manually and not mistaken for a refactoring.

External Validity. The most obvious threats are in this case the number of subject systems and their scope. A great amount of effort has been spent on achieving diversity among the subject systems. Still, the case study was performed with seven open source projects written in Java. Therefore, it is possible that the results will not fully hold for other similar projects, industrial systems or applications developed in other programming languages or paradigms. The investigation of this issue is proposed as future work.

Construct Validity. Threats to construct validity concern the relation between theory and observation. A serious threat lies in the identification of refactorings, which is based on the commit logs of the VCS. Indeed, the commit log of the revision in which a smell disappears can be retrieved. However, they may not accurately reflect the commits related to a smell removal, because developers show different behaviour for committing their changes, e.g., periodically or task-based. Also, deliberate refactorings must be distinguished from other coding activities that coincidentally result in the removal of a code smell. Log messages have to be inspected manually to make this distinction, which is usually clear, taking the aforementioned threat into account.

VI. Related Work

This section highlights some important contributions in the area of code smells and software evolution.

Several studies, among them the work of Fowler [5], show that code smells and anti-patterns have a negative influence on software quality. If no action is taken in a timely manner, then a software system will deteriorate over time. There are numerous examples available of the study and development of code smell detection techniques (e.g., [8]). There are also contributions in the area of the evolution of code smells and anti-patterns [13], [14].

Zazworka and Ackerman developed a framework called *CodeVizard* [15], which can mine data from source code repositories at source file level. The tool focuses on areas of risk, such as increasing software complexity, degrading architectures, process violations and also code smells.

Khomh et al. performed a study [16] that investigated whether classes with code smells are more change-prone than classes without smells. They provided empirical evidence that classes with code smells are more subject to change than others and that specific smells are more correlated than others to change-proneness.

VII. Conclusions

In this study we created a tool called SACSEA that computes the lifespans of code smell instances in software projects. As a case study, SACSEA has been applied to seven software projects in order to answer four research questions regarding the lifespan of code smells and the refactoring behaviour of software engineers.

RQ1 *Are some types of code smells refactored more and quicker than other smell types?* During our investigation we noticed that, on average, code smell instances seem to have a lifespan of approximately 50% of the examined revisions. However, there were some small differences per code smell type, where Feature Envy Methods seem to be refactored more than God Classes, Data Classes and Long Parameter List Classes. On first sight, the cause of this phenomenon seems to lie in the fact that Feature Envy Methods are easier to refactor, either by accident or intentionally. Also, God Classes are proven to be difficult refactoring candidates [8], while Data Classes and Long Parameter List Classes do

not form a big threat in the eyes of many developers. Overall, this implies that software engineers are not very much interested in refactoring code smells most of the time.

RQ2 *Are relatively more code smells being refactored at an early or later stage of a system's life cycle?* The results show that the majority of the smell instances in the early revisions subset of any subject system are resolved within a few revisions. However, their numbers do not outweigh the increasing number of infected instances that exist for a long period of time. Relative to the first 20% of the revisions of a system, the latest revisions do not contain many smell removals.

RQ3 *Do some developers refactor more code smells than others and to what extent?* This research question deals with the behaviour of developers regarding code smells. Within each subject system, usually one or two developers refactor more than their colleagues. The differences are not large: most of the time they are either the only ones who resolve smell instances or either refactor just a few more infected instances than other developers. Most smell instances found in the case study were removed as a side effect of other maintenance activities or the implementation of new functionality. These results hint towards low awareness or concern among developers regarding code smells.

RQ4 *What refactoring rationales for code smells can be identified?* In order to answer this question, we have taken a closer look at the log messages left behind by the developers in order to find refactoring motives. Finding such motives in the case study was one of the expectations, but was eventually not fulfilled. Some of the rationales that were mentioned by the developers are cleaning up dead or redundant code and refactoring for the purpose of code smell resolution. More rationales were found regarding performance and readability improvement. However, these motives were derived from commits that were not marked by SACSEA as responsible for smell removals and therefore fall outside the scope of the experiments. This implies that developers are most certainly aware of code smells in their software projects, although they seem to resolve them for opportunistic reasons, which explains the relatively low refactoring effort in most subject systems.

Contributions. This study makes these contributions:

- *SACSEA.* An application that can analyse multiple versions of Java systems stored on SVN repositories in order to find different types of code smells.
- *Experiment results.* The study of seven open source Java projects showing that developers are aware of code smells, but do not consider them to be a high priority.

Future work. This study provides a small step in the field of code smell evolution and more research is required to strengthen or validate the claims made in this study or broaden the knowledge by performing more studies using different variables. These are our intentions for future work:

- The investigation of more code smells.
- Analysing industrial software systems.
- Establishing correlation between smells and developers using statistics.

References

[1] M. Lehman and J. Ramil, "Towards a Theory of Software Evolution - And its Practical Impact," in *Proc. Int'l Symposium on Principles of Software Evolution (IWPSE)*. IEEE, 2000, pp. 2–11.

[2] D. Parnas, "Software Aging," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1994, pp. 279–287.

[3] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[4] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] H. Kagdi, M. L. Collard, and J. I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[8] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Germany: Springer-Verlag, 2006.

[9] N. Tsantalis, "Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings," Ph.D. dissertation, University of Macedonia, 2010.

[10] "Ptidej," http://www.ptidej.net/, Last visited: June 2011.

[11] "Maven," http://maven.apache.org/, Last visited: June 2011.

[12] R. Peters, "Evaluating the lifespan of code smells in a software system using software repository mining," Master's thesis, Delft University of Technology, 2011. [Online]. Available: http://resolver.tudelft.nl/uuid:6e02be89-3d5a-4207-a449-ca14eff30231

[13] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the Impact of Bad Smells using Historical Information," in *Proc. of the Int'l Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2007, pp. 31–34.

[14] S. Olbrich, D. Cruzes, and D. Sjøberg, "Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems," in *Proc. Int'l Conf. on Softw. Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[15] N. Zazworka and C. Ackermann, "CodeVizard: A Tool to Aid the Analysis of Software Evolution," in *Proc. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2010, p. 63:1.

[16] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *Proc. Working Conf. on Reverse Engineering (WCRE)*. IEEE, 2009, pp. 75–84.