Research on Detecting Bad Smells in Code

Joseph Fair
Computer Science Department
North Carolina State University
Raleigh, NC
jefair@ncsu.edu

1. Abstract

This paper reviews the recent research into the idea of code smells or anti-patterns. These are common mistakes that software developers make in their designs. This concept assumes that the code smells cause extra confusion, brittleness, and fault-proneness. [11] We will look at code smells, the assumption that they have an effect, and the most common methods of detection. Finally, we conclude with some suggestions for further study.

2. Index Terms:

Code Smells, Anti Patterns, Patterns, Code Quality

3. Introduction

Code Smells or anti-patterns are common problems that appear in code regardless of the domain. Just as patterns are reusable ideas that are tested by time [12], anti-patterns are traps that a developer falls into that make the code more complex or prone to bugs. Consensus is that code smells develop over time, as software maintenance evolves the code base to cover new features and fix bugs. Sometimes, instead of refactoring a whole section of code, it's easier and faster to patch up the code in a non-optimal way. As these patches build up over time, they exhibit characteristics of a code smell. Bugs in software are least expensive to fix directly after they occur [13], so we have developed tools that are integrated into the build cycle to alert us of problems early. Code smells can be mitigated by developer training and awareness. This paper will review the recent advances in code smell detection, along with new information about how much impact code smells have in real life.

4. Hypothesis

In most of these papers, the hypothesis assumes that code smells are bad, and need to be removed. Long lists of attributes, large classes, and classes that are overly intimate with each other make the code more complicated than it should be. Complex code is difficult to refactor and more likely to break than simple code that is easily understood and tested. However, not all code smells are equally odorous. Some code smells have negligible effects on the code's complexity or fault-proneness, and in rare cases we'll see where code smells make code less likely to break.

5. Motivation

The process of generating software is complex and time consuming. Fixing bugs in this process is much more inexpensive early in the process rather than later. Mistakes can cost a company time, customers, and revenue if the software is unstable. With real-time software now running airplanes, cars, and ships, bugs can be deadly[14]. By researching ways to fix bugs early in the software process, we decrease the expense of fixing code smells potentially years after they are introduced, and lessen the likelihood of bugs getting into production.

6. Related Work

Where do code smells come from, and why? That is the topic of Tufano et. al. [8] paper from this year. They looked at the records of 20 different open source projects, and a half-million code commits. After evaluating over nine thousand of these by hand, they concluded that apart from Bob and Complex Class, most smells are not introduced as a maintenance effort, but they are present in the initial commit of code. This is encouraging, because it means fixing these problems is a matter of training the developers to watch for smells during new code. As to why they are introduced, the researchers used the tags that were added during the commit. By analyzing these commits, they found 80% of smells are introduced during enhancements and new features, instead of bug fixes.

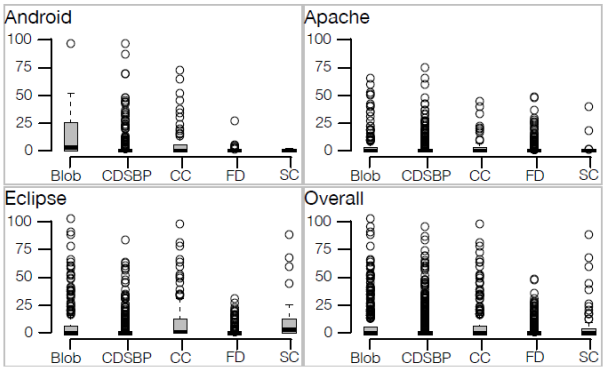
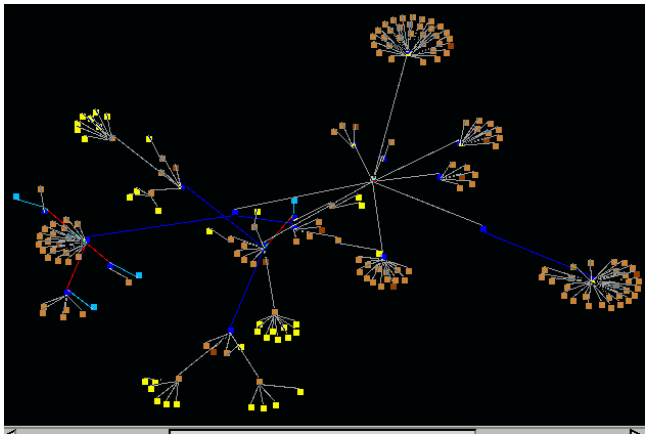


Fig. 2. Number of commits required by a smell to manifest itself.

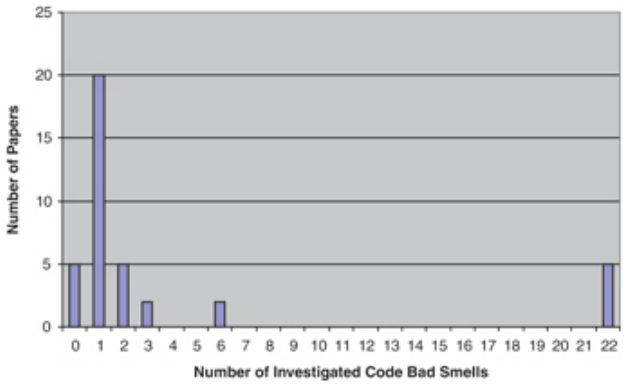
Back in 2002, Moonen [23] wrote about a smell detection tool they developed called jCosmo. They analyzed a Java program called CHARTOON, a tool for "2 ½ dimensional animations of facial expressions." As a part of their effort, they took bytecode from the unreleased commercial product and used that to research code smells. This is different from the other papers because it didn't have the source code to start with.

This team put some effort into visualizing the data they had to work with. Using a Rigi model [24] (a tool for software comprehension) they extended the displays to include not only the package and class structures, but also to add additional nodes for code smells.



Code smells have been around for a while, but Zhang et. Al. [18] researched the state of the art in code smell research back in 2008. They concluded that the research so far was subjective and focused on a few smells, not considering the entire array of possible problems. Interestingly, he quotes Pfleeger[19] "software development is as much an art as a science". We will see throughout this paper that although we make every effort to quantify our findings, this thread of development having a subjective component runs throughout [23].

Zhang et.al. [17] revisited the topic to update his findings in 2011, and found that better coverage, but still an uneven distribution of work. Most of the effort had still been focused on finding bad code smells via automation, and comparing those results to manual inspections. More code smells had been thoroughly researched, and some had hardly been touched. They assumed that this was due in part to the nebulous nature of the original specifications of the code smells.



Zhang concluded at the time that since the research was spotty, technical, and less focused on how code smells impact software development, they found little evidence that code smells are a useful tool for solving software problems.

As an example of that, Shatnawi [26] provides an early (2006) study of code smells in one release of the Eclipse platform. At the time (version 2.1), it had 6,221 classes and 451 files infected with bad smells for a total of 1,681 incidents of bad code smells.

Distribution of the Bad Smells

Bad Smell	Eclipse 2.1
Data Class	29
Feature Envy	18
God Class	31
God Method	650
Refused Bequest	219
Shotgun Surgery	734
Total	1681

Comparing big reports from Bugzilla, the team did confirm a correlation between code smells and fault-proneness. However, this was a very small sample (one version of code) and 1,380 instances were of God Method and Shotgun Surgery. Later research will show that the effect of God Class is somewhat mitigated by the fact that it is a bigger class, with more lines, and therefore more likely to have bugs.

Olbrich et. al [25] studied God Class and Shotgun Surgery in Eclipse over the period of years. This allowed them to see if the density of code smells increased or decreased over time. They also wanted to find out if fault-proneness and code churn are affected by these code smells.

As with the other research, they determined that God Classes were in fact more prone to code churn than the normal class. Although they didn't normalize for size, they did acknowledge that the size of the class may have an effect. Shotgun surgery also increased churn and fault-prone behavior.

Yamashita et al. [20] found more empirical evidence using a study of 6 developers behavior over the course of a month. By looking at 4 different projects, they could study what effects code smells had on the problems that developers coded around. Overall, the code smells didn't really seem to be that big a deal: "Within the limits established by the context of this study, it was clear that the proportion of problems that were linked to source code and more specifically, to code smells was not as large as expected. A corollary from our observations is that code smells are just partial indicators of maintenance difficulties." In the end, about 15% of the problems resulted from code smells.

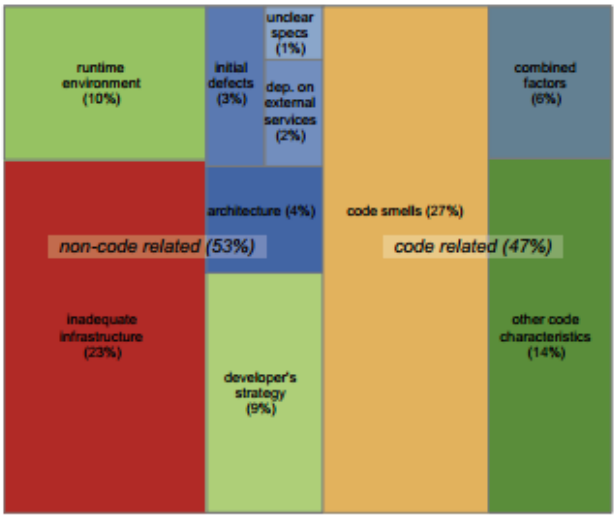


Fig. 2. Distribution of maintenance problems classified by their origins.

How big an impact are code smells to fault-proneness? Hall [5] researched a set of code smells that they determined are the least-researched. The five they picked (Data Clumps, Middle Man, Speculative Generality, and Switch Statements) are ones that had not been studied before. Also, they seem to have put a good big of effort in to filter out the least-well-defined smells. Of the remaining smells there were six, but one (Parallel Inheritance Hierarchies) required analysis over multiple versions of the software, so they eliminated this, too.

They hypothesized that none of these smells would have an impact on the code they reviewed. They had a very clever method of measuring smells. For method-level smells, they added up each method in the class that had that smell, but for class-level smells, the measurement was binary for whether the class contained that smell or not. Although there were cases of smells in each of the code bases, some of the code smells had no effect on fault-proneness, and others were inconsistent. None of the smells tested had more than a 10% effect on faults. Middle Man caused fewer effects in one project, and a different smell prevented faults in Eclipse. A more important contributor to faults seems to be code size.

Table XXIII. Simpler Negative Binomial Regression Model on the Apache Data

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	0.3471	0.1296	2.68	**0.0074**
DataClumps	−4.0870	0.8374	−4.88	**0.0000**
DataClumps:LOC	0.0080	0.0019	4.11	**0.0000**

This seemed to be the most well-researched paper of the initial research papers. There are references to earlier works throughout, a solid review of the literature, and explanations for each facet of the problem or decision that needed to be made.

Looking at the general concept of comparing static analysis of a single version of code to the dynamic analysis of mutipule versions over time is a source of interest for H. Kagdi et. al. [4] They differentiate the Information Retrieval (IR) approach of static analysis with Mining Software Repositiories (MSR) approach from using multiple revisions over time. The evolution of the software is more evident when we view it in context of not only what it does, but how one change can later cause other changes. These are called Ripple Effects, and can cause what is initially a small change to force other changes in many parts of the software. This is an attempt to use both types of research in order to detect the impact of code changes.

For comparison, they used KOffice, which uses a static code analyzer, and iBatis, which uses a MSR analyzer. By adding the analysis not normally done to each code base they hypothesized that better accuracy would result. They found that using a combination of both approaches allowed a 20% increase in accuracy over IR methods in kOffice and up to 45% improvement compared to the MSR technique in iBatis.

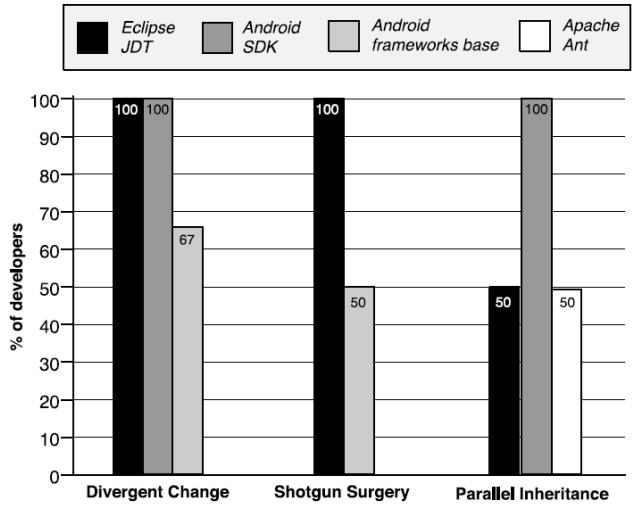
In 2013 F. Palombota [1] with Moonen [23] published more research on using MSR to detect code smells. Some code smells are difficult to find by just looking at one version of the code, but they can be discovered if you use multiple versions of code and compare how it changes over time. This allows us to see the effects of changing one feature on related functionality. The approach they propose is HIST (Historical Information for Smell deTection) and applied it to 8 open source Java projects.

Similarly, a later paper (Palomba et.al. [6]) hired 34 developers and had them evaluate code for smells. They used code from three open source projects, and interestingly, they used 10 developers from those projects, some from industry, and some from the masters program. This study was not about automated software to detect smells, but about how important developers think code smells impact the project. Instead of evaluating how many errors a code smell caused, it asked the question the other way around: How bad do the developers think this smell is.

This survey showed that from a developer's perspective some of these code smells aren't really all that bad. The most commonly perceived smells were also rated the most serious (Complex Code, God Class, Spaghetti Code), but others were minor problems, or representing coding in bad form, but would not cause problems. The explanations of each code smell and quotes from the developers were succinct and instructive.

Following up on his earlier work, Palomba[7] investigated several common code smells, using the history data from 20 open source SCM repositories to point out code smells over time. Their approach is named Historical Information for Smell deTection (HIST), and it was effective for detecting smells. Then they compared HIST to the results of 12 open source developers over the same code base, to see how they compare. All of the projects were sizable (twenty-five thousand to one million lines of code) and were in development between 3 months and thirteen years.

The results of the survey show that 75% of developers are able to manually find the code smells found by HIST. Of those, they tended to rate the severity of design problems as high to very high.



The researchers also did a comparison of HIST to DECOR, an approach mentioned earlier in the paper [2]. HIST did better than DECOR on Blob detection.

This paper focuses on 3 smells that can be more easily detected via change history: Divergent Change, Shotgun Surgery, and Parallel Inheritance, along with Blob and Feature Envy. The idea is that the first 3 are better suited to this approach, while the others may have some effect.

Extracting change data was one challenge from the beginning. SCM systems keep track of code at a file-commit level, but in order to see what smells lurk in these code bases a more finely-grained approach. Just knowing that a pair of class files changed is not enough to know that there is a case of Divergent Change, you need to be able to check at the method level.

One concern of anti-patterns is that they make the code more difficult to read. This concept is examined by Kohm et al [3] with a controlled experiment on Blob and Spaghetti code. The idea here is that given a set of developers and three sets of code (one with no code smells, one with one code smell, and one with two code smells) we should be able to show the code to developers and ask them to modify and explain it, then generate results based on their answers.

They find that the presence of one anti-pattern is not, in fact, that much of a barrier to understanding the code. However, having more than one code smell added more cognitive load and decreased understanding. The time it took the developers to find the problems and fix the code were both increased, and the number of incorrect answers on the post-study questionnaire also increased. This justifies the effort to dig out code smells where we find them, but tempers the view that every code smell is bad. Remediation should be concentrated on code with an overlap of code smells, not just fix each one.

This paper seemed to suffer from a narrow pool of participants. There were only 24 subjects, most of which were masters and PhD students. There are a large majority of developers that will not attempt this level of learning, so I would be interested in seeing a wider study with less academic focus. Also, since Blob is directly related to size, it would be interesting to see if that had an effect. By controlling for that in some other studies [2] it would seem the answer is yes.

One big assumption is the code that smells is worse than code that doesn't. Khomh et al [2] expanded on their previous paper[3] to examine this premise in their 2011 paper. They confirm the idea that code smells are a good indication of bugs, but interestingly, they compare whether size of the class has an effect. A large class would be more likely to have bugs than a small class. It would also be more likely to have code smells, just by the fact that there's more code in it. They conclude that size is a much less important factor in finding fault-prone code than the presence or absence of anti-patterns. They also look at which kinds of change are the most likely to trip up code with smells in it. Structural changes seem to be the biggest culprit, which makes sense. If your code is brittle, the most likely way to expose that is to try refactoring it to expand the functionality.

Sjoberg et. al. [9] did research by hiring developers and giving them tasks to fix bugs in code provided. They studied 12 smells over 4 different projects. Measurements were taken from a plug-in to their Eclipse IDE to measure churn (additions, deletions, and changes in the LOC) and how long it took to change it. This was summarized into effort.

They found that none of the smells they introduced were significantly associated with an increase in bugs, and one (Refused Bequest) was associated with decreased effort. Another interesting point is that the only factor they found in effecting fault rates was code size. This is a contradiction of the findings of another study [2], where class size did not have any effect. However, that study was only looking at Blog and Spaghetti, so the focus may have been too nary to account for class size.

One question that did not get asked very often is "When do code smells end?" The best paper I found about that topic is Peters et.al [22] where they not only found code smells, but then monitored to see when they were refactored from the code base. They developed a tool named SACSEA to find smells in code bases, then keep track of them until the smell disappeared.

This paper looked at God Class (GC), Feature Envy(FEM) , Data Class (DC), Message Chain Class (MCC), and Long Parameter List Class (LPLC) across 7 software projects.

	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.

This study found that Feature Envy classes are refactored slightly more than God Classes or Data classes. However, they point out that these classes may also be refactored accidentally as opposed to conscious recognition and removal.

It also found that if a code smell is going to be revised, it will happen within a few commits of its implementation. The last 20% of revisions do not contain nearly as many code smells as the first 20%.

Another telling point: One developer generally will refactor code smells at about the same rate as the other developers on his team. This indicates that the different developers all value the same. However, looking at the log messages, it does not appear that the developers were necessarily focused on refactoring code smells. The authors seem to think that the refactoring happened as a part of general bug fixing and feature development.

7. Study Instruments

The papers that were suggesting a new approach to detecting code smells generally compared their results to either another code analysis tool. This is a good comparison for a baseline, because it's an objective measure of how each algorithm is doing.

Other researchers spent time surveying developers to compare with their results. This introduces a more subjective component into the analysis. It's important to know, because developers can provide feedback on whether a code smell is or isn't valid, and can also support what kind of impact a code smell can have.

Several studies [1] [6] [22] built custom detectors for the code smells they are interested in. Although static code analyzers are common, analysis across commits is where the new research lies. These new tools can help identify problems in the evolution of a body of software that would go unnoticed by looking at a single snapshot in time.

8. Key Words

8.1 Specific Code Smells

These code smells are from Zhang et. al. [17], which is the most succinct list of smells I've found. All of these smells are referenced in more than one paper, with the exception of 3 (Alternative Classes with Different Interfaces, Incomplete Library Class, and Comments).

8.1.1 Duplicated Code

Same code structure happens in more than one place.

8.1.2 Long Method

A method is too long.

8.1.3 Large Class

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

8.1.4 Long Parameter List

A method needs passing too many parameters.

8.1.5 Divergent Change

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

8.1.6 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.

8.1.7 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.

8.1.8 Data Clumps

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

8.1.9 Primitive Obsession

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

8.1.10 Switch Statements

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

8.1.11 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.

8.1.12 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.

8.1.13 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.

8.1.14 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.

8.1.15 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.

8.1.16 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.

8.1.17 Inappropriate Intimacy

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

8.1.18 Alternative Classes with Different Interfaces

Classes are doing similar things but with different signatures.

8.1.19 Incomplete Library Class

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

8.1.20 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.

8.1.21 Refused Bequest

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

8.1.22 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.

8.2 Common Keywords

These keywords are common to the literature:

Anti-Pattern: A common design choice that is a poor choice, but is often seen in various code bases. Patterns are reusable best practices. Anti-patterns are opposite: common thing that should not be done.

Bad Code Smells - particular structures in code that can cause detrimental effects on software and should be refactored.

Blob Class (God Class): a large and complex class that centralises the behavior of a portion of a system and only uses other classes as data holders, i.e., data classes. The main characteristic of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields and–or accessors to their fields.

Change-Prone (or fault-prone): A section of software that is more common to change over time. Well-written software can change as it is improved or expanded, but poorly written software must be revisited more often to work out the bugs that come up later.

Commit: Change to the code base for a project. Can be multiple files, but only one user and one comment. The goal of the project is to get more complete data on commits.

Churn: An overall description of the amount of code that is added, deleted, or updated in a commit.

Confounding Factor - In statistics, a confounding variable (also confounding factor, a confound, a lurking variable or a confounder) is an extraneous variable in a statistical model that correlates (directly or inversely) with both the dependent variable and the independent variable, in a way that "explains away" some or all of the correlation between these two variables.

Conceptual couplings - capture the extent to which domain concepts and software artifacts are related to each other. This information is derived using Information Retrieval based analysis of textual software artifacts that are found in a single version of software (e.g., comments and identifiers in a single snapshot of source code). This analysis focused on a single version.

Context: When characterizing a software project, the context looks at attributes like: size of project, likely number of developers, target environment (mobile, plugin, or standalone app).

Construct Validity: concerns the relation between theory and observation. An invalid construct will not represent the actual facts gathered. The map is not the terrain.

Dependent Variable: Measurable results from the test. For this paper, they are: Effort (measured by NASA Task Load Index (TLX)), time taken, and percent correct.

Event Stream: The set of events that describe changed to repositories on gitHub. This is an information-rich source of data for projects that are looking at changes in code over time.

Historical Information for Smell deTection (HIST) - an approach exploiting change history information to detect instances of five different code smells - this study's proprietary software to detect Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy.[1]

Information Retrieval (IR) - analysis of textual software artifacts that are found in a single version of software (e.g., comments and identifiers in a single snapshot of source code).

Mining Software Repositories (MSR) - a broad class of investigations into the examination of software repositories. The premise of MSR is that empirical and systematic investigations of repositories will shed new light on the process of software evolution, and the changes that occur over time, by uncovering pertinent information, relationships, or trends about a particular evolutionary characteristic of the system.

Odds Ratio (OR): The likelihood of an event to occur in two samples. OR<1 means it's more likely to occur in the first same, OR>1 means it's more likely in the second sample, and OR=1 means it's equally likely in both.

Spaghetti Code: classes that have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming. They do not exploit and may prevent the use of object-orientation mechanisms: polymorphism and inheritance.

Independent Variable: Variable that the tester has control over. In this case, the presence of Blob, Spaghetti Code, or both anti patterns.

Impact Analysis (IA) - the determination of potential effects to a subject system resulting from a proposed software change.

Ripple Effects - a phenomenon that affects other parts of a system on account of a proposed change.

Repository: A set of software and all its versions. One repository is usually related to one project. A repo will usually have one owner and one to many contributors.

Schema: In this context, a schema is the description of the objects, attributes, and relationships underlying github data. Until recently the github schema was in use by the github code base, but it was undocumented, so providing details is one of the contributions of [1]. Entities : users, organizations, teams, commits, issues, repositories, pull requests, issue comments, milestones, and pull request comments.

Workload: This is the cognitive load introduced by code smells. It measures how busy a developer was when introducing the bad smell. In particular, the workload of each developer involved in a project using time windows of one month, starting from the date in which the developer joined the project (i.e., performed the first commit). The Workload of a developer during one month is measured in terms of the number of commits she performed in that month.[8]

9. Patterns in the Research

The literature seemed to focus heavily on Java. It may very well be that dynamic languages by their very nature are ill-suited to this kind of research, but after scratching the surface of what python can do, wouldn't it make sense to see if lambda structures, list comprehension, or custom iterators are more fault prone? The limitations of Java are potentially hiding big code smells

Each paper on finding code smell followed a similar pattern:

Find a particular subset of code smells. With over 30 to choose from, it makes sense to concentrate on a few. Most of these studies looked for code smells that would only be visible via the method they employed (IR vs. MWS), with the exception of [4], which found two code bases that used one of each, and applied the other approach to that code base.

Create a method of finding those smells. Most researches used off-the-shelf detectors as a baseline, but the more interesting ones tackled the bigger problem of finding different versions of the code at different times, and measuring the impact of code changes then test the new method against old methods and report the results.

The papers that focused on the impact of code smells were similar

First, focus on a subset. Then, set up tests for users to see code with smells in it. Finally, run the results against an external measure and compare how they did.

Future Scope/Recommendations

One of the common problems in these papers is that code smells are subjective. Although the concepts are not hard to see, they are hard to quantify in terms that can be used from one study to another. Is a God Class smell set to a threshold of a hundred lines or a thousand? This was particularly evident in [5] where it quotes a developer explanation that the two classes that are Inappropriately Intimate are really one thing, it's just coded up into two classes. With that insight, the code smell didn't apply. A good research effort could be developing a standard for each of the major code smells, so that study result can be applied across different studies. Zhang et. al. at least made the attempt in [16]. It would be difficult to get everyone to agree, so the standards could have 'tweaks' to the setting so that for THIS code base a God Class is 500 lines, but THAT code base it's 1000. Then, your adjustments from the norm can be summarized all in one chart "We used these thresholds for smell detection." I propose we use the name "The Fair Olfactory Code Scale" after its creator. FOCS for short.

Another thing that jumped out is that there are inefficiencies in studying for bad code smells. One step that was repeated in several places was to create a set of code that had a known number of bugs in it. Then this was tested either against the novel approach to see how it worked, or against a group of developers to see what they thought. Instead, it would be more efficient if there were a non-trivial set of code that could be seeded with a known set of code smells. A good example is Yamashita [20], where the code used in their experiment was seeded with knows problems, including code smells. In that case, only about 30% of the problems were code smells. This would allow other researchers to use an objective standard for testing code and developers. With this concept, you can introduce a "clean" code base that has no smells as a baseline to find false positives. If deficiencies in the code base are found, new versions with additions and corrections can be proposed. A research paper for someone who wants to impact the field would be to create a curated set of code with known code smells. Future researchers are encouraged to use the name "Standard Testbed for Investigating Code Smells" STINCS, for short.

Finally, in most the papers, there was not much correlation between bad code smells and in increase in complexity. Only when you layer one code smell on top of the other or create large classes does confusion start to set in. As for fault-prone behavior, it seems more likely the driver problems is class size, such as Blob or God Class. These are relatively easy to detect with existing technology.

10. Conclusion

Overall, it seems like research in this area is pretty well covered. Here's a quote from [5] that summarizes most of the research:

> *There is no significant relationship in any of the three systems between faults and Switch Statements. There are significant relationships between faults and LOC and Message Chains across two systems.*
>
> *Where there is a relationship between faults and smells, it is not always a positive relationship. Some smells (e.g., Middle Man in ArgoUML) are associated with fewer rather than more faults. … no consistent relationships exist between particular smells and faults across all three systems.*

We can detect some code smells that seem to have some effect (God Class) and we know that some code smells don't have much effect. We can study how changes in code can effect later versions and measure those impacts. We can warn against potential problems with automation during the development process in a timely manner so that developers can fix the code shortly after they add their smells.

Making code more reliable is a noble goal, but I would suggest that future work be better spent in making developers in industry

aware of code smells and how to fix them. Yamashita [15] started this with a study of developers in industry to see how they evaluated the importance of code smells. We need better tools to be able to detect code smells. Murphy-Hill [21] seemed to be on the right track by testing developers with the open-source eclipse plugin called Stench Blossom. The research on code smell life cycles [22] was a step in the right direction. We need detection tools to be easy to integrate into automated build systems (ant, gradle, rake, make) and we need tools that are easy for industrial developers to use.

11. References

[1] Georgios Gousios and Diomidis Spinellis. GHTorrent: Github's data from a firehose. In Michele Lanza, Massimiliano Di Penta, and Tao Xie, editors, 9th IEEE Working Conference on Mining Software Repositories (MSR), pages 12–21. IEEE, June 2012. (doi:10.1109/MSR.2012.6224294)

[2] Khomh, Foutse, et al. "An exploratory study of the impact of antipatterns on class change-and fault-proneness." Empirical Software Engineering 17.3 (2012): 243-275

[3] An empirical study of the impact of two antipatterns blob and spaghetti code on program comprehension"15th European Conference on Software Maintenance and Reengineering CSMR 2011pp. 181-190 2011

[4] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, “Blending conceptual and evolutionary couplings to support change impact analysis in source code,” in Reverse Engineering (WCRE), 2010 17th Working Conference on, 2010, pp. 119–128.

[5] "Some code smells have a significant but small effect on faults" T Hall, M Zhang, D Bowes, Y Sun - ACM Transactions on Software , 2014

[6] Palomba, Fabio, et al. "Mining version histories for detecting code smells." *IEEE Transactions on Software Engineering* 41.5 (2015): 462-489.

[7] Walter, Bartosz, and Tarek Alkhaeir. "The relationship between design patterns and code smells: An exploratory study." Information and Software Technology 74 (2016): 127-142.

[8] Tufano, Michele, et al. "When and why your code starts to smell bad." Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 2015

[9] Sjøberg, Dag IK, et al. "Quantifying the effect of code smells on maintenance effort." IEEE Transactions on Software Engineering 39.8 (2013): 1144-1156.

[10] Fontana, Francesca Arcelli, et al. "Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains." Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE, 2013.

[11] Fowler, Martin. Refactoring: improving the design of existing code. Pearson Education India, 2009.

[12] Brown, William H., et al. AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc., 1998.

[13] Stecklein, Jonette M., et al. "Error cost escalation through the project life cycle." (2004).

[14] Dowson, Mark. "The Ariane 5 software failure." ACM SIGSOFT Software Engineering Notes 22.2 (1997): 84.

[15] Yamashita, Aiko, and Leon Moonen. "Surveying developer knowledge and interest in code smells through online freelance marketplaces." User Evaluations for Software Engineering Researchers (USER), 2013 2nd International Workshop on. IEEE, 2013.

[16]Min Zhang, N. Baddoo, P. Wernick, and T. Hall. 2008. Improving the precision of Fowler’s definitions of bad smells. In Proceedings of the 32nd Annual IEEE Software Engineering Workshop (SEW’08). 161-166.

[17] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: A review of current knowledge. J. Softw. Mainten. Evol. Res. Pract. 23, 3 (2011), 179-202.

[18] Min Zhang, Tracy Hall, Nathan Baddoo, and Paul Wernick. 2008. Do bad smells indicate "trouble" in code? In Proceedings of the Workshop on Defects in Large Software Systems (DEFECTS’08). ACM, New York, NY, USA, 43-44.

[19] Pfleeger, S.L. 2002. What software engineering can learn from soccer. Software, IEEE, 19 (6). 64-65

[20] Yamashita, Aiko, and Leon Moonen. "To what extent can maintenance problems be predicted by code smell detection?–An empirical study." *Information and Software Technology* 55.12 (2013): 2223-2242.

[21] Murphy-Hill, Emerson, and Andrew P. Black. "An interactive ambient visualization for code smells." *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010.

[22] Peters, Ralph, and Andy Zaidman. "Evaluating the lifespan of code smells using software repository mining." *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012.

[23] Van Emden, Eva, and Leon Moonen. "Java quality assurance by detecting code smells." *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002.

[24] Rigi (software) https://en.wikipedia.org/wiki/Rigi_(software)

[25] Olbrich, Steffen, et al. "The evolution and impact of code smells: A case study of two open source systems." Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement. IEEE Computer Society, 2009.

[26] Shatnawi, Raed, and Wei Li. "An investigation of bad smells in object-oriented design." Third International Conference on Information Technology: New Generations (ITNG'06). IEEE, 2006.