An Empirical Study of Bad Smell in Code on Maintenance Effort

Prabhu Chand Makkapati

Sai Chaitanya Tadikonda

Research Paper

Lewis University

12/16/2022

**ABSTRACT**

Software deteriorates in quality over time for various causes, including program aging, inconsistent design, and preliminary requirement analysis in the initial phases of software development. A deeper issue may be present when there is a bad code smell. Bad coding habits are neither defects nor technically incorrect; therefore, they do not obstruct the software's ability to perform as intended. Refactoring is the name given to the procedure of getting rid of the problematic code. This paper discusses available detection methods for offensive code smells. The automated code scent detection and refactoring tool we provide in this research can be used to calculate risk factors by identifying code smell and reduce risk factors by refactoring techniques. Refactoring is a procedure for reorganizing or enhancing a software's internal structure without altering its functionality. Also given is a novel code smell detection called "Lazy Catch." Declarative programming is used with object-oriented software metrics to accomplish this goal. We employed this tool to identify the foul odors in case of studies based on oops, such as those in C#, CPP, and Java. That implies that this tool is not language-dependent. Three categories are used to categorize the risk factor level (Hi, Low, and Medium).

**INTRODUCTION**

At the coding level, design flaws are nothing more than offensive odors. Refactoring involves making structural changes to the software without sacrificing any functionality. Understanding the system design will help identify likely mistakes and foresee potential failures. A collection of software measurements is used to find the code smells that could lead to frequent failures and related expenses. Eliminating code smell is suggested as a technique to raise software design standards. In large systems, faulty code detection is still time-consuming and error-prone. This might be caused by inadequate tool support. On the detection of offensive code odors, numerous researchers have worked extensively.

Bad code smells are warning signs of trouble, but they are simply suggestions and should not be taken as mandates. There are many tools available for spotting unclean code. A tool called Checkstyle can assist programmers in writing Java code that complies with a coding standard and can identify code smells like Large Class and Long Method. Due to a lack of tools at the time, CodeNose was created in 2005 as a prototype for automated code smell detection. It is implemented as an Eclipse plug-in that recognizes and displays code smells like how compilation errors and warnings are shown. JDeodorant, an eclipse plug-in, locates offensive odors and performs refactoring to eliminate them. Refactoring is described as "the process of altering a software system that complies with which it advances its internal structure but does not impact the external activities of the code." Refactoring is crucial in converting software systems into design patterns-based systems to expand a high-quality software system[3]

**THE EFFECT OF BAD SMELLS IN CODE**

Software with code smells will not necessarily stop working; it will still provide an output, but it may process more slowly, run at a higher risk of failure and mistakes, and be more susceptible to faults in the future. A code smell is a factor in poor code quality, which raises technical debt.

Code smells are a sign of a more serious issue, yet they are easy to detect or smell. The best smell is one that is simple to locate but will result in a fascinating issue, such as classes with only data and no behavior. Tools can be used to identify code smells quickly. According to the design principles established by an organization, code smell varies from project to project and developer to developer. We have listed typical sorts of odors as a cheat sheet for better code quality so you can quickly recognize and categorize them[6].

**Maintenance of Software**

Software engineering is changing a software product that has been delivered and needs to be fixed, improve performance, or change other factors. The expertise of the programmer, occurrence, system documentation, and the nature of the system itself all have an impact on the maintenance process. They frequently interact with intricate and challenging systems. Before altering a plan, they must first understand how it works[9]

**code smell**

Software's distinctiveness or code smells can indicate a programming or design challenge and make it challenging to evolve and maintain software. Refactoring techniques must be used to enhance the internal quality of software in order to eliminate code smells and improve maintainability and software development. Code smells can be eliminated in some circumstances; depending on the system, sometimes leaving them in place is the best course of action. The code smell Large Class serves as a good illustration of this. Smell detection takes time and money. Therefore, removing them as soon as possible is preferable when they need to be disconnected. Given that many code smells might be overlooked while programmers are at work, tools for their identification are especially helpful. The different detection strategies employed by the detection tools are often based on the computation of a specific set of shared metrics, standard object-oriented metrics, or metrics generated ad hoc for the smell detection purpose.

These are apparent and palpable signs that there is a problem with an application's core code that can potentially result in catastrophic failures and ruin the performance of an application.

Code smells frequently appear in the following situations:

* duplicate code: As a general rule, duplicate code is considered duplicate if another brief code explains the same functionality in a more concise way, such as blocked repeated codes[8]
* dead code: It is essential to remove dead code, which means removing code that is not being used by the organism. Our source control systems were created to address this problem, and that is why they exist.
* long methods: With refactoring rules like the extract method, an extended and composite method is broken up into dummy and well-named methods. An explanation of how various method components might be removed and used as new techniques. Typically, the extracted new techniques are called within the old ones in their original positions; as a result, the parameter list is not shortened by the extraction[7]
* long parameter list: If a developer creates a method with parameters, he or she should be aware that the more parameters there are, the more complex the method's maintenance requirements become. In object-oriented programming, this is different, and long parameter list methods can be restored by giving an object as a substitute for the parameters because long parameter list techniques are challenging to read and alter.
* Comment lines: Comment lines are usually used to make the resource code easier to read and understand and are usually unnoticed by compilers and line-by-line check code servers that analyze the code for broken links.
* Large class: Large classes are divided into smaller ones, each for a particular dependability, to increase their understandability and preserve.
* Lazy class: The lazy classes should put extra emphasis on requesting weight information from a specific source to ensure that they are accurate. In order for a project to become more complex, it is necessary to add more classes to it[6]

Code that is particularly "smelly" may be ineffective, unresponsive, convoluted, and challenging to modify and maintain. Code smells may not always be signs of a significant issue, but investigating them frequently reveals deteriorated code quality, resource drains, or even serious security flaws hidden within the application's code. It at least requires teams to do some thorough tests on the code, which frequently shows some crucial parts in the code that need to be fixed with remedial work.

**DETECTION TOOLS**

Different techniques are used by bad code smell detection technologies to identify bad code smells. Here, a few of them are discussed.

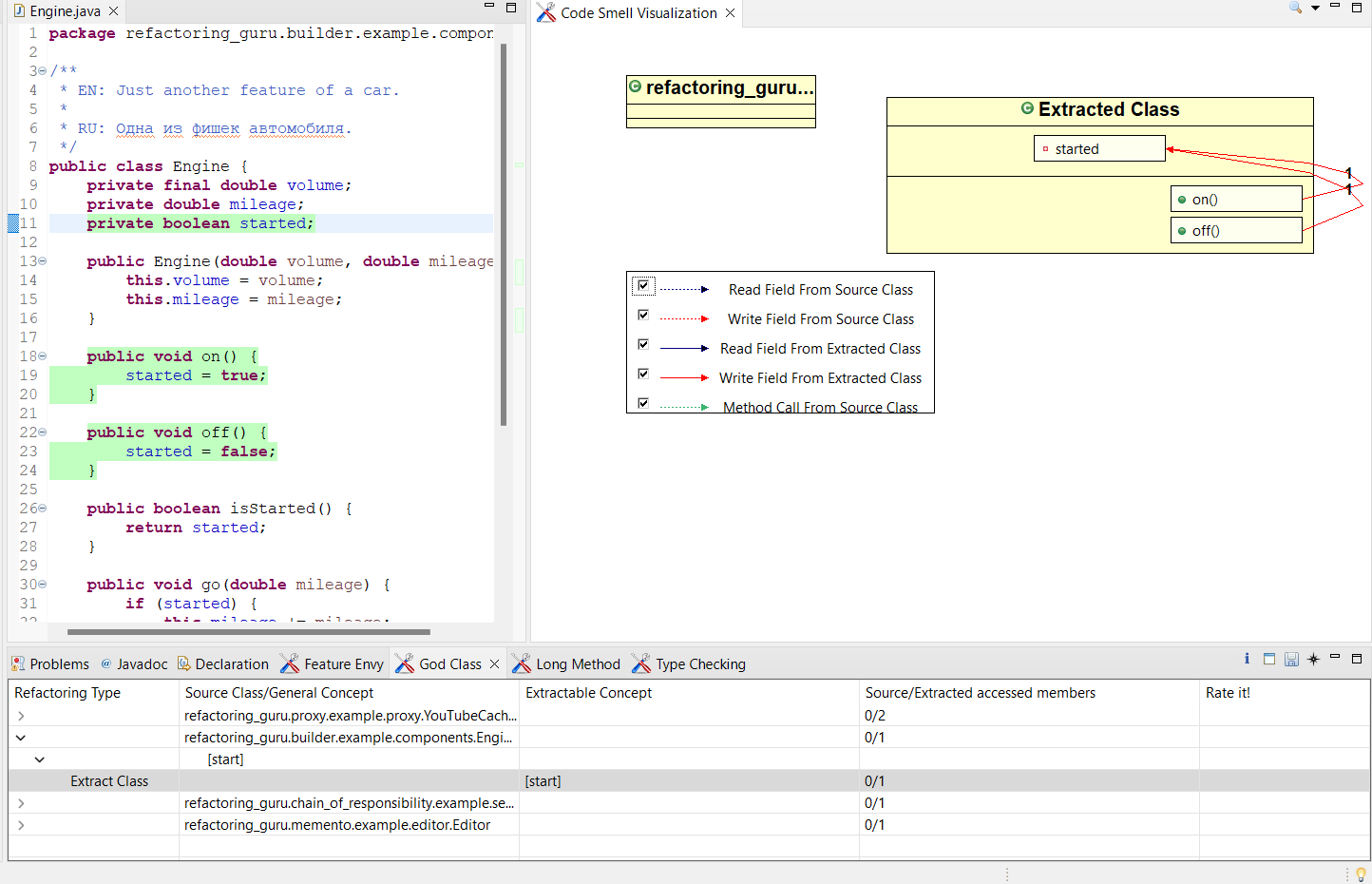
JDeodorant is a plug-in for Eclipse that can identify chances for deriving cohesive classes from "God Classes" and then automatically carry out the developer-selected refactoring.

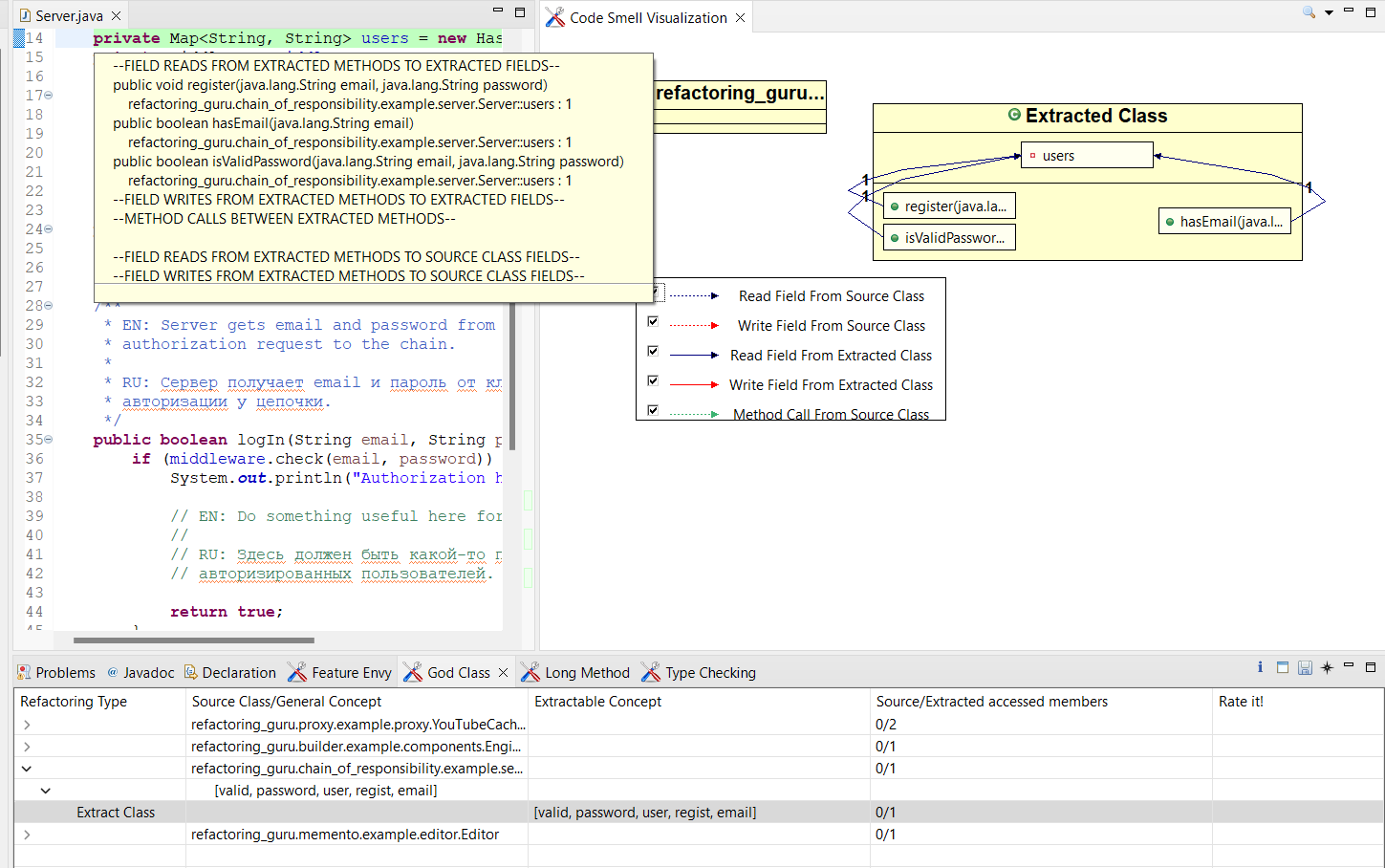
In the background of eclipse, inCode is active. If a programmer creates a faulty structure while writing code, Eclipse will indicate this by displaying "eclipse show error" and warnings in the form of red color blocks next to the code. Free to download, InCode is offered by the business Intooitus, of which Marinescu is a co-founder. We spent 700 euros on the Together license from Borland. As a result, it should be simple to replicate our work using the same tools[5]

In the Java source code, PMD tracks duplicated or dead code, empty catch or switch statements, variables that are not used, and dead code. Bad odors like long parameter lists and duplicate codes can be detected by this toll.

Java application bugs can be found with FindBugs. It can spot common programming errors, including thread synchronization problems and the wrongful use of API methods.

JDeodrant tool





**COMPARISON TOOLS**

There are a number of code-smell detection programs on the market, each of which has a unique feature set. We review a few of them in this part.

Clock Sharp

The Clock Sharp code organizer is included as a part of both Visual Studio 2008 and 2010, and it is a code organizer for the C# programming language. A command-line tool that can be used as a search engine and code verification tool, it checks code using more than 100 programming rules.

Locate bugs

A lightweight open source project called Find Flaws performs an analysis of Java byte code for bugs while identifying the four most likely causes of alarming and unsettling errors by using ongoing research to speculate what the most likely culprits might be.

PMD

With the help of a source code analyzer called PMD (PROGRAMMING MISTAKE DETECTOR), it is possible to locate five kinds of errors in source code: pests, such as insertions or copies of code; dead code; redundant code; empty try, catch, finally, and switch statements; private methods and parameters; string and string buffer usage; and overly complex terminology that is ineffective and overly complicated. The majority of the code is inefficient, has local void variables, has for-and-while statements, and has unnecessary and dead code[4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Comparison Criteria | Developed Software | Clock Sharp | Find Bugs | Programming Mistake Detector |
| Tool Description | Standalone | Plug-in Tool | Stand-alone | Plug-in Tool |
| Threshold | Fixed Threshold value | No threshold value | No threshold value | No threshold Value |
| Smell Filtration | Can view all error module wise | View all the errors at the output | View all the errors at the output | View all the errors at the output |
| User Interface | User friendly | Not user friendly | User friendly | User friendly |
| Results | Represented in grphics | Is too long to read | Can be filter by classes, packages | Not true error |
| Memory Released | Yes | - - - | - - - | - - - |

**RECOMMENDATIONS TO PREVENT BAD SMELLS**

Although code smell can frequently be avoided, engineering teams frequently need more time and resources to do so. Many times, developers would rather wait for issues to manifest before addressing them than seek future problems that could have an influence on operations in the future. Furthermore, as a result, code smells continue to exist—even those that are simple to address. Your teams can address code smells by using the appropriate tools. We've already discussed how the rework rate indicator might show whether your team is producing bad code that adds to technical debt[3]

Once you have established your starting points, you can use the engineering benchmarks we determined by studying roughly 2,000 development teams to create goals for improvement. Not only does LinearB present these indicators, but we also offer a framework for improvement that is entirely based on measurements. By performing things like warning you about lingering PRs, perfunctory reviews, or high-risk deltas so you can proactively intervene and stop smelly code from getting merged into production, our WorkerB automation bot can help you achieve those goals. Although strong, LinearB isn't tricky to use. It is simple to use, quick to set up, and seamlessly interacts with Git and programs like GitHub, GitLab, and Jira.

Utilize automation; current IDEs like Eclipse and IntelliJ IDEA can carry out a variety of refactoring tasks automatically. They can assist you, for instance, in renaming classes or methods while updating all coding references to those changes. If a test is missing for the source code you are reworking, write one first. Then, make sure the test still passes after your changes to the code. Modest discrete modifications—make small, one-at-a-time changes while continually ensuring that tests don't fail. Don't be hesitant to rewrite; refactoring frequently necessitates rewriting portions of the code. When it's required, give it the time it deserves. Refactoring cannot be done when time is limited, therefore approach the task rested. To help the team enhance the quality of their code, team leaders should be aware of the value of refactoring and allocate calm periods of time. The lack of appropriate tools is the primary source of refactoring issues, not issues with the code itself. Refactoring almost always involves moving things around, updating method signatures, and renaming variables and methods. Making all of these adjustments by hand is a recipe for catastrophe[1]

In our opinion, this problem is essential to every developer so he or she can be prepared to face code refactoring situations due to bad smells in the code in the future.

**CONCLUSION AND FUTURE SCOPE**

Last but not least is the fact that we have to admit that comparing the tools is not only challenging but in some situations, employing them may be just as challenging and urgent. The source code of the banking system is found to have various code smells that are associated with the graphical user interface designed for the banking system. Based on the generated object-oriented metrics, it can be demonstrated that there is a significant relationship between each statistic and specific coding errors that may occur. Our goal in this article was not to evaluate the tools but rather to describe our usage of them and to draw attention to the challenges involved in performing a comparative analysis. The first practical investigation into how to code odors affect software maintenance work in an illegal industrial setting. We conducted a multiple linear regression analysis using the same technique of analysis for each of the odors. Code smells are primarily common negative patterns linked to poor training techniques that cause more severe issues with software maintenance. Code smells can make software products challenging to maintain. In this research project, we suggest a code smell detection tool that makes use of the threat notion. We created an automatic risk-based code smells detection tool as a proof of concept. Using the tool, we were able to identify issues in a C# case study. In the case study, code smells like long methods, long parameter lists, lazy classes, dead code, comment lines, temporary fields, and lazy catch blocks were found with the total amount of memory used and unused (Before and after refactoring). When the maintainability index goes down, the risk factor will eventually follow.

The results available here are just the beginning of many scent studies that we plan to conduct in the future. The future work will focus on a replication of Mantyla's developer research experiments and on an analysis of the implications that smell suppression will have for testing, in addition to the creation of an experiment based on developers. These are just the first of many small studies that we will conduct in the future to determine how to improve the maintenance of software, its systems, and many other fields.

**References:**

[1] Erich Gamma, John Vissides, Ralph Johnson, Richard Helm: Design Patterns Elements of Reusable Object Oriented Software, Addi- son Wesley Professional, 1995.

[2] Joshua Kerievsky: Refactoring to Patterns, Addison Wesley Professional, 2004.

[3] Danphitsanuphan, Phongphan, and Thanitta Suwantada. "Code Smell Detecting Tool and Code Smell-Structure Bug Relationship." Engineering and Technology (S-CET), 2012 Spring Congress on. IEEE, 2012.

[4]Ito, Yu, et al. "A Method for Detecting Bad Smells and ITS Application to Software Engineering Education." Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on. IEEE, 2014.

[5]Mens, Tom, and Tom Tourwé. "A survey of software refactoring." Software Engineering, IEEE Transactions on 30.2 (2004): 126-139.

[6]Fontana, Francesca Arcelli, et al. "Automatic metric thresholds derivation for code smell detection." Proceedings of the SixthInternational Workshop on Emerging Trends in Software Metrics. IEEE Press, 2015.

[7]Abdelmoez, Walid, Essam Kosba, and Ali Falah Iesa. "Risk-Based Code Smells Detection Tool." The International Conference on Computing Technology and Information Management (ICCTIM2014). The Society of Digital Information and Wireless Communication, 2014.