Success Factors for Code Smell Detection Tools

Term paper

Sebastian Hunkeler, shunkele(at)hsr.ch

University of Applied Sciences Rapperswil, HSR
Department of Computer Science, MSE

Keywords: Software engineering, code smell detection, success factors, refactoring, usability

Acknowledgement: I would like to thank my advisor Markus Stolze for his attentive reading and many helpful comments. And of course many thanks to Yves Thrier for his valuable feedback.
Abstract
A code smell is a symptom in the source code that potentially indicates a design problem in the corresponding software. Because code smells can be complex and scattered across the code, locating them can be a hard and cumbersome task. Tools called smell detectors have been developed in order to inform the programmers about the presence of smells in their code, and to help them revealing and finally getting rid of them.

In this paper, we propose five success factors for smell detection tools which can help to improve the user experience and therefore the acceptance of such tools. We also describe the results of a survey conducted with 30 software engineers that tests several hypotheses about success factors of smell detection tools, used by the test persons.

1 Introduction

Nowadays object-oriented programming is one of the most popular ways of developing software. Moving away from procedural programming facilitated a new kind of defining software problems [OCBZ09]. Code smells are a way of giving names to these software design problems. A code smell is an awkwardly written code that breaches software design principles. Having code smells does not mean that the code does not fulfill the necessary functionality. But code smells make software difficult to maintain [OCBZ09], source code difficult to understand and errors hard to track down [KPG09]. However, there is a cure for code smells: Refactoring. Refactoring helps to transform existing code into code of better quality [FB99]. For every code smell there is a refactoring and vice versa.

Refactoring and smell detection are both widely supported by various tools for almost every modern programming language [DKWB06]. While several tools are used frequently by software engineers, others are not very popular and therefore used rarely. In section 2 we will propose factors which have influence on the success and the popularity of these tools.

Manually detecting code smells requires a wide experience with programming and a profound knowledge of software design. Furthermore combing through code and searching for code smells can be a tedious and time-consuming task. Code smell detectors can help us mastering the challenge of finding code smells. They not only can simplify smell recognition for programming novices but also experienced programmers can save a lot of time during refactoring [MHPB09]. Assuming we have a duplicated code smell in our code: Of course, locating it is a rather trivial task but it is still a tiresome work. Simplifying this task is the purpose of smell detection tools.

In this paper we want to analyze why it is still not a common task for a software engineer to use such tools. For this purpose, we will analyse the success factors of smell detectors and create hypotheses about the influence on the usage of such tools (section 2). In order to verify these hypotheses we have conducted a survey using a questionnaire which is described in section 3. The results of the
survey are reported in section 3.2 followed by a discussion in section 3.4. We end with a conclusion in section 4.

1.1 Related Work

There has been conducted considerable research regarding the opportunities and limitations of refactoring and the impact of code smells. But little work has been done on analyzing success factors of refactoring and smell detection techniques including the corresponding tools. An interesting paper dedicated to this topic had been published in 2009 [BMH09]. In this paper, Black et al. elaborated a compendium of guidelines for smell detection tools which can improve the user experience and the efficiency of refactoring using smell detectors. However, it is focused on a particular smell detector called “Stench Blossom”. Consequently it covers not all possible aspects of such tools. In this paper we try to get a more general view of success factors of various existing code smell detectors and refactoring tools.

2 Success Factors

As requirements for a software can change over time, changes to the source code have to be made in order to implement the additional functionality. The more maintainable the software is, the less work it is to perform this task. Therefore it is desirable to have smell-free and tidy code. To achieve this a software engineer can either use a smell/bad-practice detector or do it manually. As a matter of fact, it is more efficient to use a tool for this [BMH09]. Among the existing tools, there are some very popular and successful tools while others are used rarely. Based on our experience we have derived a number of factors which have a positive influence on the success and the popularity of such an application. For each factor we proposed a hypothesis which was evaluated using a questionnaire (section 3).

2.1 Usability

A list of guidelines for smell detectors was proposed in [BMH08] and [BMH09]. These papers experimentally evaluated the relevance of their guidelines. All of the test persons found that these guidelines are important. Since all these guidelines are related to usability, we establish our first hypothesis:

Hypothesis 1 Smell detection tools which follow the guidelines of [BMH09] are more successful.
Restraint | The tool should not overwhelm a programmer with the smells that it detects.
Relationality | When showing details about code smells, the tool should display the relationships between affected program elements.
Partiality | The tool should emphasize smells that are difficult to see without tool support.
Non-distracting | The tool should not distract the programmer.
Estimability | The tool should help to estimate the extent of a smell in the code.
Unobtrusiveness | The tool should not block the programmer from the other work while it analyzes or finds smells.
Context sensitivity | The tool should tell first and foremost about smells related to the code the programmer is working on.
Lucidity | In addition to finding smells, the tool should tell the programmer why particular smells exist.

Tab. 1: “Guidelines for smell detectors” [BMH09]

2.2 Continuity

In [FB99] it is proposed to use the “Two Hats Metaphor” which means to switch between two distinct activities: Adding functionality and refactoring. This kind of refactoring is called “root canal refactoring”. On the contrary, there is floss refactoring which doesn’t distinguish between adding functionality and refactoring, but proposes refactoring as you add code. It has been argued that root canal refactoring lacks two qualities: Fun and efficiency [MHPB09].

When using root canal refactoring you first have to examine the code in order to re-understand the context and thereby being able to refactor. This is time-consuming and cumbersome. In contrast, when using floss refactoring, you are already in the context of the current code. In that case refactoring is less complicated for a programmer. Being less complicated leads to being less error-prone. Furthermore, you will find that most of the time adding functionality to a software is more fun than refactoring an existing, foreign code. This is because it is complicated to get into the context of a foreign, messy code. And since refactoring in that combination is no fun, you probably won’t do it. So you will end up with a “big ball of mud”[FY99].

Thus for refactoring to be both, enjoyable and efficient, it must be a con-
continuous discipline. Visualizing smells and bugs in the current programming context encourages the programmer to do continuous refactoring. Therefore a smell detector should support the programmer continuously with smell detection information and refactoring proposals.

**Hypothesis 2** Smell detection tools which encourage floss refactoring are more successful.

Continuity as a success factor is related to the guideline “availability” proposed in [BMH09]. But since we consider continuity as one of the most important success factors, we describe this as a separate element.

### 2.3 Interpretation

Many existing tools calculate code metrics. But few tools can interpret the metrics and react on these interpretations. A feature that is missing amongst most of the smell detectors is to perform refactorings directly as a solution to a detected smell. We deduce that the smell detection tool should propose one or more possible refactorings which help removing the detected smell. While this may not be possible for all smells, it is possible for simple smells such as “duplicated code smell” [FB99]. In this case the corresponding refactoring proposal may be “Extract Method” or “Extract Class”. The most convenient way for the programmer would be to perform this refactoring with only one mouse click.

**Hypothesis 3** Tools which interpret code metrics and propose refactorings are more successful than pure analytic tools.

### 2.4 Specificity

Beside the conventional smell detectors, there are bad-practice detectors. These detectors are programming language-specific, because every programming language has its own bad practices. Consider the example in figure 1. The Findbugs plug-in for Eclipse has found a Java-related bad coding practice: Using a conventional string concatenation in a for loop which leads to bad performance. Instead it’s recommended to use a StringBuffer. This is Java-related since in many other programming languages the “+=” operator is overloaded to use a StringBuffer or a similar construct automatically in the background.

As we can see, bad practices detectors not only help to improve code quality but also to improve performance. They have a greater potential to improve code, because the detection rules are adapted to a specific programming language. Thus we suppose that bad practices detectors are more popular than conventional smell detectors.

**Hypothesis 4** Tools which detect language-specific bad practices are more successful than traditional smell detectors.
2.5 Integration

The majority of smell detection or metrics tools are delivered either as plug-in or as stand-alone application. Stand-alone applications have the disadvantage, that complying with the interpretation and continuity factors of section 2.3 and 2.2 is hardly possible. In order to make code enhancement suggestions the tool needs a visual integration into the IDE. Continuity cannot be achieved since a stand-alone application never knows which piece of code the programmer is editing. When deployed as a plug-in, a smell detector can display continuously whether any code smells have been found without forcing the programmer to leave his IDE. When a smell has been found the tool can easily display the presence of the smell and a proposal on how to eliminate it. This behavior emphasizes the usability factor, introduced in section 2.1 as well.

Hypothesis 5 Smell detection tools which are integrated into the IDEs are more successful compared to stand-alone applications.

3 Survey

To verify the hypotheses of section 2, we created a questionnaire consisting of 22 questions about smell detection and code enhancement tools and their characteristics. The test persons were asked about their experience with these tools. The questionnaire was filled out by 30 individual software engineers. They were requested to answer questions about the particular smell detection or code enhancement tool they use most. Furthermore they were provided with three general statements about refactoring and smell detection. The test persons had then to decide whether they think that the statements are either true or false. In case they disagreed with the statement, we asked them to give a reason for their disagreement. In the last part of the questionnaire, the test persons had to rate their preferred smell detection or code enhancement tool according to the guidelines presented in section 2.1.

All questions are related to the success factors presented in section 2 and aim to verify the hypotheses made for each success factor. The questionnaire and the corresponding results can be found under the following URL: http://wiki.ifs.hsr.ch/SemProgAnTr/files/.

Fig. 1: Bad practice found by Findbugs plug-in for Eclipse

```java
if(fields.size()>0){
    s="();
    for(String field : fields) {
        s+=field, ";
    }
    s=s.substring(0,s.length()-2)+")";
}
return s;
```
3.1 Methodology

The questionnaire has been sent to the test persons as an online survey. Half of the subjects are former Bachelor of Computer Science students which graduated this year. Eight of them are doing the Master of Science degree while seven test persons are employed at various companies. The other half of the test persons are Bachelor of Computer Science students in the third year. We selected these test persons because all of them have carried out at least one extensive software development project during the bachelor studies and therefore are qualified for participating in the survey.

The online survey consisted of three parts. The purpose of the first part was to build a usage statistic of smell detection or code enhancement tools and IDEs. The intention of the second and the third part was to confirm the proposed hypotheses.

Task 1: Ranking smell detector categories

Subjects were asked to rank the importance of the three categories of smell detection tools:

- Smell detector
- Language-specific bad practices detector
- Code metric calculator

Since this explicit category ranking made by the test persons is considered rather subjective, we additionally did another calculation which is more meaningful. We calculated the implicit category ranking based on the associated category of the mostly used smell detection tools of the test persons. If the category “Language-specific bad practices detector” takes first place, then Hypothesis 4 is confirmed.

Task 2: Evaluating refactoring types

Next, subjects were provided with a short explanation of floss refactoring and root canal refactoring. They were then asked to indicate in which way they use their preferred smell detection tool. The possible answers were:

- Floss refactoring
- Root canal refactoring
- Doesn’t support refactoring (pure analytic tool)

Furthermore they were asked to state, whether they think that the following statement is either true or false:

“Floss refactoring is more efficient, because I’m already in the code’s context when refactoring.”
If the number of tools that are used in a floss refactoring kind exceeds the number of tools used in one of the other ways, and if the number of subjects that affirmed the above statement exceeds the number of those that denied it, then Hypothesis 2 is confirmed.

**Task 3: Reviewing usefulness of tool’s proposals**

Then, the test persons were asked to report how frequently they follow the recommendations or actions provided by their preferred smell detection tool, in order to eliminate a found issue. The possible answers were:

- Always
- Often
- Rarely
- Doesn’t provide any recommendations or actions

In addition subjects were asked to state, whether they think that the following statement is either true or false:

> “Recommendations for refactoring steps by the smell detection/code enhancement tool are very important, because smells can be complex and difficult to understand. And as a result of this, the necessary refactoring steps may not be obvious.”

In order for Hypothesis 3 to be confirmed, the following two conditions must hold:

- The number of subjects that follow often or always the recommendations must exceed both, the number of subjects which rarely do and the number of subjects whose tool doesn’t provide any recommendations.
- The number of subjects that affirmed the above statement must exceed the number of those that denied it.

**Task 4: Rating IDE integration**

After that, subjects were asked to estimate how well their preferred smell detection tool is integrated into their IDE. The following three options were given:

- Feels like native IDE feature
- Could be better integrated
- Doesn’t provide any integration

Moreover they were asked to state, whether they think that the following statement is either true or false:
“It’s important to have the smell detection/code enhancement tool integrated into the IDE, because it’s more efficient not always having to switch between different windows and applications.”

In order for **Hypothesis 5** to be confirmed, the following two conditions must hold:

- The number of subjects that use a tool which feels like a native IDE feature must exceed the number of subjects which use a suboptimally integrated or non-integrated tool.

- The number of subjects that affirmed the above statement must exceed the number of those that denied it.

**Task 5: Judging guideline conformity**

Finally the test persons were asked to evaluate the conformity of their preferred smell detection tool to each of the guidelines of section 2.1. For this purpose we used a four-level Likert scale. Additionally subjects could choose “I don’t know” in case they could not make an accurate judgement.

For the evaluation of this task, we selected the four most popular smell detection tools among all tools used on a regular basis by the test persons. Then we calculated the weighted sum of the ratings for these tools. If the rank of each tool regarding its ratings for the guideline conformity is equal to the rank related to the share in smell detectors, then **Hypothesis 1** is confirmed.

We chose the weights for the ratings as follows:

- Strongly agree: 2
- Agree: 1.5
- Disagree: 0.25
- Strongly disagree: 0.25
- I don’t know: 0

It is possible that a particular smell detection tool entirely lacks a feature or property which should be judged. In this case the ratings would all be negative. But we wanted that lacking a feature or property has little impact on the result. Therefore we chose a rather small weight for both “disagree” and “strongly disagree”. Moreover we didn’t want that “I don’t know” ratings have any influence on the result. Thus we set their weight to zero.

**3.2 Results**

According to the results of the survey, there are only few smell detection tools and IDEs which are popular, while the other applications are mostly unknown among the majority of the test persons. The actual share in IDEs and smell
detection tools among the test persons is shown in Figure 2 and Figure 3. The share in the programming languages is divided mainly between Java with 47%, C# with 40% and C++ with 7%.

**Fig. 2: Share in IDEs**

**Fig. 3: Share in smell detection tools**

**Results to task 1: Ranking smell detector categories**

The survey results confirmed **Hypothesis 4** that tools which detect language-specific bad practices are more successful than traditional smell detectors. Among the tools which are regularly used by the test persons, every tool except the code metric calculators “Structure101” and “Metrics” is in the category “Language-specific bad practices detector” (see figure 3). The score\(^1\) of the weighted calculation\(^2\) for the rank of the smell detector category was 71 for both categories “Smell detector” and “Language-specific bad practices detector”. The score for “Code metric calculator” was only 35.

**Results to task 2: Evaluating refactoring types**

The survey results confirmed **Hypothesis 2** that smell detection tools which encourage floss refactoring are more successful. 48% of the test persons use their

---

\(^1\) Sum of all weighted rank counts

\(^2\) Items ranked first are valued higher than the following ranks
preferred smell detection tool for floss refactoring, while 21% use it for root canal refactoring. 31% of these tools do not support any refactoring. Moreover 72% of the subjects agreed with the statement that floss refactoring is more efficient.

Results to task 3: Reviewing usefulness of tool’s proposals

The survey results confirmed Hypothesis 3 that tools which interpret code metrics and propose refactorings are more successful than pure analytic tools. 79% of the subjects follow often or always the recommendations of their smell detection tool. Only 21% of the users don’t follow these recommendations or use a tool which doesn’t support such a functionality. 81% of the subjects agreed with the statement that recommendations for refactoring steps by the smell detection/code enhancement tool are very important.

Results to task 4: Rating IDE integration

The survey results confirmed Hypothesis 5 that smell detection tools which are integrated into the IDEs are more successful compared to stand-alone applications. 69% of the smell detection tools used by the subjects felt like being natively integrated into the IDE. 21% thought that their tool could be better integrated. Only 10% of the tools are not integrated at all. 97% of the test persons agreed with the statement that it is important to have the smell detection/code enhancement tool integrated into the IDE.

Results to task 5: Judging guideline conformity

The survey results confirmed Hypothesis 1 that smell detection tools which follow the guidelines of [BMH09] are more successful. Ranking 1 describes the popularity (see figure 3) of smell detection tools which is as follows:

1. ReSharper (40%)
2. FindBugs (17%)
3. Structure101 (13%)
4. Metrics (7%)

Ranking 2 characterizes the conformity of smell detection tools to the guidelines (see section 2.1) which is as follows:

1. ReSharper (79.5 points)
2. FindBugs (33.25 points)
3. Structure101 (8.25 points)
4. Metrics (6 points)
The particular ratings can be seen in figure 4. Since Ranking 1 and Ranking 2 are equal, Hypothesis 1 is confirmed. The result additionally confirms Hypothesis 3 because the language-specific bad practices detectors are ranked significantly higher than code metric calculators in both rankings.

Comparing the ratios between the scores of the language-specific bad practices detectors across both rankings shows another potential relation:

- The ratio of the share in smell detectors between ReSharper and FindBugs is \(2.625\)
- The ratio of the positive ratings regarding the guideline conformity between ReSharper and FindBugs is \(2.4\)
- The deviation of these two numbers is \(0.225\)

Since the deviation is rather small, it seems that the conformity of smell detection tools to the proposed guidelines is numerically related to their popularity. In order to verify this assumption, further research and a separate survey would be necessary, which is out of scope of this paper.

### 3.3 Limitations

The survey has been conducted exclusively with students and former students which graduated this year as explained in section 3.1. Therefore these results may deviate from results which could be obtained by asking software engineers with several years of work experience.

### 3.4 Discussion

Although the share in smell detection tools of the category “Language-specific smell detector” confirmed Hypothesis 4, the general opinion of the test per-
sons is that language-specific smell detectors and conventional smell detectors are equally important. The fact that most of the modern language-specific smell detectors have also detection rules for conventional smells explains why language-specific smell detector are nevertheless more successful.

Even though all the hypotheses have been confirmed, there are some interesting remarks of test persons which disagreed with some of the statements in the survey. One subject believes that doing one thing at a time is more efficient and thus root canal refactoring may be the preferred type of refactoring. Focussing on one particular task at a time certainly is a good practice. But root canal refactoring requires you to re-understand the context of a code and therefore is time-consuming and exhausting. This may lead to deferring the refactoring task to a later point. And our experience has shown that later never comes.

Another test person stated that refactoring as a separate activity becomes necessary as a project grows in its size and complexity and suggested to use both refactoring types complementary. We must admit that in real-world projects this will definitely be the case. But if floss refactoring would be consequently performed by all participating software engineers, refactoring as a separate discipline would hardly be necessary.

One test persons thinks that floss refactoring is not a good practice, because it sometimes breaks his code when performing refactorings and adding functionality at the same time. But refactoring performed correctly never breaks any code.

An interesting objection against floss refactoring has been raised by another test person. The statement is as follows: When you are doing floss refactoring and you are in the context of the code which needs to be refactored, you will presumably miss some of the present smells. This is because you probably have seen your own code so many times that it just prevents you from seeing the wood for the trees and thus from actually finding the code smell. It would have to be evaluated in a separate experiment whether this is a veritable statement, and therefore is left open.

4 Conclusion

Modern smell detection tools are very sophisticated and thus widely used. They are an important part of today's software engineering. However there are still some features and qualities missing. The smell detection tools have difficulties to show the relationships between affected program elements of a smell because they may be widespread. Furthermore they still lack to display the found issues grouped by their importance. A threshold based visualization is introduced in [BMH09] as a possible solution to this problem. Another drawback is that most of the smell detection tools do not provide a continuous code analysis and need to be run manually. One hindrance may be that the code is changing continuously during programming, and therefore the analysis of the code would need to be updated all the time.

A desired feature that is mostly missing among the smell detection tools
is to detect design level smells which are spread across different classes. Both analyzing and visualizing such smells is a difficult task. Moreover it is hardly possible to have automated refactoring recommendations for every code smell, because with some code smells such as feature envy there are just too many options for possible refactorings.

Another problem of code smells is that they may exist although good coding practices have been applied. For example the visitor pattern [GHJV94] inevitably breaks feature envy [SMA]. In this case a trade-off between using approved design patterns and having minor code smells must be found.

The biggest problem however is that it will never be possible to definitively detect a complex code smell because of Rice’s theorem [Ric53]. Although the development of smell detection tools progresses, it will not be possible to replace sanity and reason and expert knowledge.

4.1 Future Work

Our empirical analysis suggests that future work on smell detectors could on the one hand include a tighter integration into the IDE, on the other hand a better customization and visualization. Customization may be achieved either by letting a user define custom smell detection rules, or by contributing to a data set which is maintained and released regularly by a community. Although the community approach is rather suitable for open source projects, it could also be adapted by proprietary tools.

References


[GHJV94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994

[Ker05] Kerievsky, Joshua: Refactoring to Patterns. 2005


[TCC08] Tsantalis, Nikolaos; Chaikalis, Theodoros; Chatzigeorgiou, Alexander: JDeodorant: Identification and Removal of Type-Checking Bad Smells. (2008)