Understanding the Longevity of Code Smells
Preliminary Results of an Explanatory Survey

Roberta Arcoverde
Opus Group, PUC-Rio - Brazil
rarcoverde@inf.puc-rio.br

Alessandro Garcia
Opus Group, PUC-Rio - Brazil
afgarcia@inf.puc-rio.br

Eduardo Figueiredo
UFMG - Brazil
figueiredo@dcc.ufmg.br

ABSTRACT
There is growing empirical evidence that some (patterns of) code smells seem to be, either deliberately or not, ignored. More importantly, there is little knowledge about the factors that are likely to influence the longevity of smell occurrences in software projects. Some of them might be related to limitations of tool support, while others might be not. This paper presents the preliminary results of an explanatory survey aimed at better understanding the longevity of code smells in software projects. A questionnaire was elaborated and distributed to developers, and 33 answers were collected up to now. Our preliminary observations reveal, for instance, that smell removal with refactoring tools is often avoided when maintaining frameworks or product lines.

Categories and Subject Descriptors
D.2.3 [Software Engineering]: Coding Tools and Techniques – object-oriented programming, program editors, standards.

General Terms
Measurement, Experimentation, Human Factors.

Keywords
Refactoring, code smells, empirical study.

1. INTRODUCTION
Code smells are symptoms in the source code that potentially indicate a deeper maintainability problem [2]. Smell occurrences represent structural anomalies that often make the program less flexible, harder to read and to change. Code smells entail evidence of bad quality code in any kind of software. However, both detecting and removing these anomalies are even more important when reusable code assets are considered, such as libraries, software product lines (SPLs) and frameworks [12]. When it comes to SPLs, for instance, smells found on the core modules will be replicated in all generated applications, propagating the code anomalies to several derived artifacts. In order to avoid these problems, developers should eliminate code smells before they have been propagated to other applications. Refactoring [2] is the most common approach for removing anomalies from code.

Even though recent studies have shown refactoring has become a common practice, with well-known benefits [11], some categories or patterns [7] of code smells seem to be, either deliberately or not, ignored. Understanding which and why these refactoring candidates are neglected can help us to identify improvements for refactoring tools and IDEs. Previous studies [5, 6] were dedicated to understanding common refactoring practices, as well as identifying how and when they are routinely applied. Murphy-Hill has recently presented an extensive study on how programmers refactor the code, identifying several common refactoring habits [5]. Another study [6] shows that usability issues with refactoring tools are one of the main reasons why they are underused, presenting a set of recommendations to improve their speed, accuracy and usability.

Our study intends to complement such previous investigations by revealing recurring factors which lead developers to not worry about certain code smells (Section 2). We designed a questionnaire in order to try to understand why they, either deliberately or not, let anomalies persist in code (Section 3). The questionnaire was made available as an online survey in October 2010 to 33 volunteer developers with diverse programming skills. The questions were trying to identify (i) which refactorings were considered as more difficult to apply, (ii) which of them are seen as the most important, and (iii) why refactorings of certain smells are usually neglected. Based on our survey’s initial results (Section 4), tool proponents can identify improvements and weaknesses of current refactoring tools and processes – and define more effective refactoring strategies for long-life reusable systems, such as libraries and product lines, that are very critical to organizations. It is also our intention to share our preliminary results with others so that improvements to the survey design can be identified, and the next steps of our study (e.g., structured interviews with developers) can be better shaped (Section 5).

2. GOALS AND HYPOTHESES
Our goal is to identify possible reasons why certain smells remain in the source code based on refactoring habits. We defined the refactoring habits of software developers as a sum of the following characteristics: (i) how often code is refactored, (ii) which refactorings are prioritized, (iii) which refactorings are considered to be harder to apply, and (iv) how often and when refactoring tools are used.

By identifying which refactorings are more commonly prioritized – and, therefore, which are neglected – it is possible to further analyze the causes of such neglects. For example, our study shows that one of the main causes why refactoring tools are not used is their inability to properly communicate the effects of a given refactoring throughout the code.
We are also particularly interested in assessing whether the nature of the software reuse demands (i.e., wider software usage scope) influences the refactoring habits. If developers have different habits when working on reusable assets, the existence of specific resources, such as full-fledged support for recurring refactorings of product-line variability, for example, can be justified. We also considered the differences between developers who use test-driven development (TDD) [1] and developers who test-last or do not test at all. Since refactoring is part of the development cycle of TDD, we were interested in analyzing whether the refactoring habits were influenced by its use. We also considered the correlation between using test-first approaches and reuse [3,4], which directly relates to the groups of developers and applications we considered in our study. Based on the literature review and on the open questions above, our hypotheses are enumerated below:

H1: There is no difference in the refactoring habits among developers of reusable assets and standalone applications;

H2: The identified factors that contribute for high smell longevity don’t change between developers of reusable assets and developers of standalone applications.

H3: The use of refactoring intensive development methods, such as TDD, does not influence the factors that contribute to the persistence of smells in the code.

3. SURVEY SETTING

In order to collect information that helps identifying which factors influence the persistence of code smells, we applied a survey with different groups of developers. We sent the survey to development teams of different companies. A preliminary version of such survey was defined and applied within a small group of programmers. The survey was then refined based on this group’s observations, regarding the clarity and objectivity of the questions. A glossary was included in the beginning of the survey explaining terms and acronyms used, for disambiguation. The survey’s final version was further divided in three sections:

1. Identification of the developer profile: in order to categorize the subjects in different profiles, the first section was composed of questions regarding their level of experience, development techniques they are familiar with, and how often they work in producing reusable code. This section made it possible for us to analyze the differences between answers from experienced developers and from beginners.

2. Refactoring habits: the second section contained general questions that were focused on aspects related to the developers refactoring habits, as described in Section 2. The developers were also asked which factors are determinant when they do not use a refactoring tool for certain smells.

3. Classifying refactorings: in the third section, the developers were encouraged to rank 5 common refactorings according to 3 different qualities: frequency, difficulty and importance.

Our investigation was also inspired by previous findings of recent observational studies [5,6]. The idea is to strength evidence and better understand previously-revealed smell removal trends. The results found on a previous study [6], for instance, explore some of the reasons why refactoring tools are underused. We used these reasons as options when asking what factors lead the developers to not use automation tools. We also left empty spaces after each question to allow developers leaving their comments.

4. RESULTS

The online questionnaire has been conducted since October 2010 and, so far, 33 volunteer developers answered it. The first questionnaire section groups the subjects into two categories: those who work more often implementing reusable assets, and those who are mostly involved with programming standalone applications. As illustrated in Figure 1, 20 developers (61%) work on standalone applications more frequently. In addition, 9% of programmers declared to work regularly on both software categories. It is important to say that these results cannot be generalized, since we have not used any special procedure for sampling the population.

Figure 1 – Kind of software subjects work on regularly

Reusing and TDD flock together. We also asked how often developers use TDD. In order to avoid misunderstandings, we detailed the TDD technique in the survey’s glossary. Therefore, developers that eventually used it unconsciously could answer the TDD-related questions correctly. For this question, developers could choose a number from 1 to 5, where 1 means “Never” and 5 means “Always”. In the analysis, we considered three different terms for better communicating the results: “rarely” represents answers 1 and 2; “sometimes” represents the subjects that answered 3; and “often” represents answers 4 and 5.

Figure 2 - How often developers use TDD

Regarding the use of TDD, Figure 2 shows an interesting, albeit expected, difference between the two groups of developers. Those working more often on projects with high level of reuse apply TDD more frequently than those involved mainly with standalone applications. Intuitively, we could explain this difference considering that reusable code must be tested properly and be loosely coupled – two known effects from the use of TDD [1, 10].

Figure 3 - Programmer's experience

Our survey also considered the programmers’ experience in software development in order to interpret how this experience influences refactoring habits. As Figure 3 shows, most of the developers have between 2 and 10 years experience with software.
development. Even though there is no significant relation between using TDD and the level of the developers’ experience, we could notice different results when comparing their refactoring habits.

**Recurring code smells and refactorings.** The aim of the second part of the survey was to analyze how developers apply refactoring routinely in order to distinguish the refactoring habits for each profile. The first question assessed the developer’s familiarity with different code smells. The developers were asked to indicate, from a predefined list of anomalies, the ones they find more frequently in the code they work with. According to their answers, the most recurring smells are duplicated code (26), long methods (23), inadequate naming for classes, methods and variables (23), long classes with too many responsibilities (14) and classes that deeply depend on details of other classes (10). Surprisingly, the results among developers working with reusable assets and those working with standalone applications are similar. This might be an indication that the nature of the software being developed does not influence the kind of code smells commonly found. We also identified that all experienced programmers (having 5 or more years of experience) indicated duplicated code as a common smell. Our results can help tool proponents to prioritize refactorings that assist to either identify code clones or deal more effectively with these recurring anomalies.

Regarding the frequency of use of refactoring tools, the results show some differences between the two groups of developers, as indicated in Figure 4.

![Figure 4 - How often refactoring tools are used](image)

An interesting result is that around 50% developers working on standalone applications frequently use refactoring tools. The opposite seems to occur among developers working on applications with high level of reuse. About half of these developers said they seldom use refactoring tools. These were the main reasons they pointed out for not using a refactoring tool are (i) the tool does not support the desired refactoring (41%), (ii) they do not know any refactoring tool (37%) and (iii) tools do not allow them to easily follow the effects of refactoring (28%).

Usability problems of the refactoring tools were recently pointed out [6] as one of the main reasons why they are underused. Specifically, the inability to properly visualize the effects of a triggered refactoring is mentioned as a major problem, which was confirmed in our study. The difficulty in verifying whether the original behavior of the program was preserved was also pointed out as a problem – indicating another issue of refactoring tools regarding their visualization mechanisms. Our survey also revealed that most of the developers that claimed not knowing any refactoring tool had at most 5 years experience in software development. Later interviews with participants revealed that inexperienced professionals were unaware of modern IDE capabilities regarding refactoring automation. For instance, most of them did not consider renaming to be a refactoring, which explains why they claimed not knowing any refactoring tool.

On the third part of the survey, developers were asked to rank five refactorings in terms of three qualities: frequency, difficulty and importance. The refactorings - rename, extract method, inline, extract local variable and move member - were chosen because they are the top 5 most commonly performed refactorings based on the Murphy-Hill study [5]. We also considered the different natures of these refactorings regarding the scope of their changes. For example, rename can have a wider change scope when applied to a public member, extract local variable, on the other hand, has a narrower change scope, and usually does not affect other modules, methods or classes.

We divided the following questions between developers of standalone applications and those working on reusable assets – so that we could compare the differences found. The first question referred to how difficult the application of the listed refactorings is. The programmer could choose for each refactoring a value from 1 to 5, in order of difficulty. We considered in Figure 5 only the refactorings ranked as 4 and 5.

![Figure 5 - Most difficult refactorings](image)

Rename was indicated as the most difficult to apply by developers of reusable assets. Some comments they left on the survey, combined with preliminary interviews, show that this is associated with the required reverse compatibility of this kind of software – for example, renaming a method can break client code of a library or framework. The difficulty in renaming methods and interfaces relies on the impact that this refactoring can have on external code. Local scope refactorings were not pointed out by any developer as being difficult to apply – thus, Inline and Extract Local Variable do not appear in Figure 5.

Move Member was considered the most difficult refactoring to apply for both groups. Lack of tool support was pointed out as one of the reasons why this is considered a difficult refactoring, combined with the difficulty in identifying possible breaks when it is applied to a public member. This can be an indication that current refactoring tools are not yet sufficient for handling refactorings on libraries, frameworks or product lines – where this sort of change can have an impact on client and derived applications. A simple visualization mechanism warning the users of possibly breaking changes could be an important asset for tool proponents targeting reusable code.

**Refactoring prioritization.** We also asked the developers to rank the five chosen refactorings in terms of their importance - or priority order. We considered in Figure 6 only the refactorings ranked as 4 and 5. We can notice that developers of reusable assets also consider the importance of a consistent and stable API when choosing which refactorings should be prioritized. This explains the difference between the two groups of developers regarding the Rename refactoring – we can see that 90% of conventional applications developers considered it important, but only 40% of the reusable code developers gave the same answer.

Even though both groups of developers have considered the importance of **extract method** similarly (around 80%), we can...
notice a significant difference with respect to other refactorings. When relating these results to the refactorings difficulty, we see that rename was pointed out as one of the most difficult to perform, but it’s not considered very important for developers of reusable assets. Some comments left on the survey indicated that those developers deliberately postpone wide scope refactorings due to the risk of breaking backwards compatibility, which is why such refactorings are not prioritized. Move member was pointed out as one of the least important refactorings by both groups of developers, yet it is considered difficult to apply. This observation that the more difficult the refactoring application is the less important it is considered may indicate a cause-effect relation.

![Figure 6 - Most important/prioritized refactorings](image)

**Figure 6 - Most important/prioritized refactorings**

At last, answers regarding the most frequent refactorings follow the same pattern of the previously presented ones. For instance, rename is the most applied refactoring by developers of standalone applications but is one of the least applied by developers working with highly reused code. The presence of tool support in most modern IDEs for performing rename might justify why this refactoring was considered the most frequent by standalone applications developers; on the other hand, developers of reusable assets also consider API stability, and seem to neglect rename operations deliberately. In this context, we can also note that nearly 40% of the reusable asset developers consider renaming to be an important refactoring, although only 20% claim to use it frequently. However, it is well known that the program lexicon need to change very frequently as software evolves [8]. This can be an indication that developers do not perform it due to lack of tool support for renaming operations on reusable software.

![Figure 7 - Most frequent refactorings](image)

**Figure 7 - Most frequent refactorings**

There is also an interesting variation with respect to extract local variable. Around 40% of reusable applications developers consider it “high priority”, but 80% claimed to perform it frequently. One possible cause for this variation is that local scope changes are easier to apply than global changes, encouraging developers to perform it more frequently. On the other hand, crosscutting concerns are often the cause of defects and should be unavoidably refactored [9]. We can also notice that extract local variable is closely related to duplicated code [2], one of the most commonly found anomalies, according to our previously detailed results. This is another possible reason why extract local variable was mentioned as the most frequently performed refactoring by developers of reusable systems.

5. CONCLUSIONS

This work presents preliminary results of a study aimed at identifying factors that contribute to the longevity of code smells. Our analysis can help refactoring tool proponents to identify specific improvements in the context of libraries, frameworks and software product lines – where issues like backwards compatibility and API stability are a major concern. Some of the obtained conclusions from the results of our survey are: (i) developers of widely-scoped reusable code consider the possibility of breaking APIs before refactoring; (ii) developers of reusable assets consider contract breaking changes harder to apply than developers of standalone applications; (iii) on average, developers that apply TDD use refactoring tools more frequently; (iv) refactoring tools are often used; and (v) refactoring prioritization is fairly different when we consider reusable assets versus standalone applications.

We discovered developers of reusable assets deliberately postpone refactorings for different reasons – they are specifically concerned with the possibility of breaking client code or derived applications when performing global scope changes. Improving the visualization mechanisms of refactoring tools can address this concern – since we also identified that the difficulty in visualizing changes is one of the main reasons why tools are not used as much as they could be. Further studies could deepen these results, applying the survey to a broader audience with different profiles and backgrounds. Another ongoing work is the execution of interviews with a subset of developers in order to clarify some points and deepen the understanding of the obtained results.

6. REFERENCES


