Do Developers Care about Code Smells? An Exploratory Survey

Aiko Yamashita
Mesan AS & Simula Research Laboratory
Oslo, Norway
aiko@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
leon.moonen@computer.org

***Abstract*—Code smells are a well-known metaphor to describe symptoms of code decay or other issues with code quality which can lead to a variety of maintenance problems. Even though code smell detection and removal has been well-researched over the last decade, it remains open to debate whether or not code smells should be considered *meaningful* conceptualizations of code quality issues from the developer's perspective. To some extent, this question applies as well to the results provided by current code smell detection tools. Are code smells really important for developers? If they are not, is this due to the *lack of relevance* of the underlying concepts, due to the *lack of awareness* about code smells on the developers' side, or due to the *lack of appropriate tools* for code smell analysis or removal? In order to align and direct research efforts to address actual needs and problems of professional developers, we need to better understand the knowledge about, and interest in code smells, together with their *perceived criticality*. This paper reports on the results obtained from an exploratory survey involving 85 professional software developers.**

***Index Terms*—maintainability; code smells; survey; code smell detection; code analysis tools; usability; refactoring**

I. Introduction

The presence of code smells indicates that there are issues with code quality, such as understandability and changeability, which can lead to a variety of maintenance problems, including the introduction of faults [1]. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of these systems [2]. Because code smells are motivated from situations familiar to developers, design critique that is based on these concepts is likely to be easier interpretable by developers than the traditional numeric OO software metrics. Moreover, since code smells are associated to specific set of refactoring strategies to eliminate them, they allow for integration of *maintainability assessment* and *improvement* in the software evolution process.

Since the first formalization of code smells in an automated code smell detection tool [3], numerous approaches for code smell detection have been described in academic literature [4–13]. Moreover, automated code smell detection has been implemented in a variety of commercial, and free/open source tools that are readily available to potential users.

However, even though code smell detection and removal have been well-researched over the last decade, the evaluation of the extent to which such approaches actually improve software maintainability has been limited. More importantly, it remains open to debate if code smells are useful conceptualizations of code quality issues from the developer's perspective. For example, the authors of a recent study on the lifespan of code smells in seven open source systems found that developers almost never *intentionally* refactor code to remove bad code smells from their software [14]. Similarly, in our empirical study on the relation between code smells and maintainability [15, 16], we found that code smells covered some, but not all of the maintainability aspects that were considered important by professional developers. We also observed that the developers in our study did not refer to the presence of code smells while discussing the maintainability problems they experienced, nor did they take any conscious action to alleviate the bad smells that were present in the code.

So, we can ask ourselves the question if code smells are really important to developers? If they are not, is this due to the lack of *relevance of the underlying concepts* (e.g., as investigated in [15]), is it due to a lack of *awareness about code smells* on the developer's side, or due to the lack of *appropriate tools* for code smell analysis and/or removal? Finally if support for detection and analysis is lacking, which are the features that would best support the needs of developers? To direct research efforts so it can address the needs and problems of professional developers, we need to better understand their level of knowledge of, and interest in, code smells.

To investigate these questions, this paper presents an exploratory, descriptive survey involving 85 software professionals. The respondents were attracted by *outsourcing* the task of completing our survey via an online freelance marketplace for software engineers. This proved to be a successful method for ensuring both sample size and covering diverse aspects of the software profession demography. The paper analyzes and discusses the trends in the responses to assess the level of knowledge about code smells, their perceived criticality and the usefulness of code smell related tooling. Based on our findings, we provide advise on how to improve the impact of the reverse engineering & code smell detection scientific community on the state of the practice.

The remainder of this paper is as follows: Section II briefly discusses the background and related work. In section III, we present our research methodology. In section IV, we present and discuss the results from the study, analyze trends in the responses, and discuss limitations. Finally, we conclude in Section V and discuss directions for future research.

II. Background and Related Work

Code smells are characteristics or patterns that serve as indicators of degraded code quality, which could hinder comprehensibility and modifiability. Code that exhibits code smells can be more difficult to maintain, and this can lead to the introduction of faults. Beck and Fowler [1, Ch. 3] informally describe 22 smells and associate them with different refactoring strategies that can be applied to improve the design. Martin extended the work of Beck and Fowler with an elaboration on a set of design principles and "new smells" that were advocated by the Agile community [17].

Over the last decade, code smells have become an established metaphor for aspects of software design that may cause problems for further development and maintenance of the system [1, 2]. Code smells identify locations in the code that violate OO design principles and heuristics, such as the ones described in the work by Riel [18] and by Coad and Yourdon [19]. As such they are also closely related to the work on design patterns [20, 21] and *anti-patterns* [22], although code smells manifest themselves generally at a more local scale than (anti-)patterns.

Because code smells are linked to challenges in comprehensibility and modifiability, the (automated) analysis of code smells allows us to integrate both *maintainability assessment* and *maintainability improvement* into the software evolution process. Moreover, considering that many of the descriptions of code smells in [1] are motivated by situations familiar to developers, it can be expected that code smells lead to software design *critiques* that are easier to interpret by developers than the traditional numeric OO software metrics.

Van Emden and Moonen [3] provided the first formalization of code smell detection and developed an automated code smell detection tool for Java. Since then, numerous approaches for code smell detection have been described in literature [4–13]. Moreover, automated code smell detection has been implemented in commercial tools such as Together, Analyst4J, Stan4J, InCode, NDepend, and CppDepend, and in free/open source tools like JDeodorant, and OClint.

Mäntylä investigated how developers identify and interpret code smells, and how this compares to results from automatic detection tools [23]. Other studies have empirically investigated the effects of individual code smells on specific aspects related to maintainability, such as *defects* [24–26], *effort* [27–29] and *changes* [11, 30].

Recently, two studies were published that investigated the lifespan of code smells during the evolution of software systems [14, 31]. Both studies found that code smells accumulate in systems over time; smells are usually introduced when the method in which they reside was initially added; smells are almost never removed; and most smell removals were not due to targeted refactoring but as a side effect of other changes. Peters and Zaidman [14] conclude that developers *may be aware*, but are *not concerned* by the presence of code smells.

In our earlier longitudinal maintenance case study on six professional developers, we investigated how code smells relate to the maintainability characteristics *considered important* by professional developers [15], and how code smells related to maintenance problems *experienced* by professional developers [16]. We found that although code smells covered some of the maintainability properties that were considered important by developers, a considerable percentage of those maintainability properties were unrelated to code smells. In addition, we observed that the developers in our study did not refer to the presence of code smells in relation to the problems they experienced, they did not look for tools to analyze code smells, nor did they take any conscious action to remove the bad smells that were present in the code.

We conclude that, despite the significant amount of code smell related research that has conducted in our community, and despite the increasing number of code smell detection tools that are available, it remains debatable how useful these notions and tools are *in practice*, from the *professional developer's perspective*, and what would need to change.

We are aware of only one other study that investigated the awareness, concern or perceived criticality of code smells from the developer's perspective (albeit in a specific context): Arcoverde et al. [32] report on the preliminary results from an exploratory survey amongst 33 developers of *reusable assets* in a framework or product line context, investigating why certain code smells are not refactored. They found that developers in this context often skip or postpone refactoring code smells to avoid changing the API or breaking the contract, because this could introduce faults in client code or derived applications. They conclude that better visualizations of the *impact* of refactorings are needed, especially in the context of maintaining frameworks or product lines.

Similarly, we have only been able to locate one study that focused on developer's needs and wishes for code smell related tools: The Stench Blossom work by Murphy-Hill and Black first explores the requirements for code smell detectors to support refactoring [33], and then implements and evaluates an unobtrusive ambient code smell visualization on a mix of students and professional developers [34]. The main conclusion from their study is that code smell detection tools should not get in the way of development activities.

III. Methodology

In order to investigate the developers' level of knowledge and concern on code smells, we chose to use the survey method. According to Fink [35], a survey is a: "*system for collecting information from or about people to describe, compare, or explain their knowledge, attitudes, and behavior*". As such, we consider this the most suitable research method to investigate these questions on a relatively large sample of software professionals. We use Fink's guidelines for setting up and conducting a survey, which involves the following steps and activities: (a) setting objectives for information collection, (b) designing the study, (c) preparing a reliable and valid survey instrument, (d) administering the survey, and (e) managing/analyzing the data. The reminder of this section discusses in more detail what we did for each of these steps.

A. Setting Objectives for Information Collection

As the first step, a brainstorming session was organized to define goals and scope of the survey. The main goal was set to explore the level of *insight* (i.e., awareness, knowledge) professional developers have with respect to code smells, and to determine if, and why they are interested in code smell-related concepts and tools. A secondary goal is to explore how code smells (concept or tools) are currently used within industry, how they could potentially be used within industry, and what would be needed to address a potential gap.

B. Designing the Survey

Considering that our goal is to collect information to better characterize and study the phenomenon of interest, we decided that the best survey format would be an *exploratory, descriptive survey* that consisted of both closed and open questions. Open questions are especially suited for such *qualitative exploratory surveys* when previous experience or literature is insufficient to guide the design of closed questions [35].

We use non-probability sampling, more specifically *convenience sampling* [35], by using freelance marketplaces [36, 37]. The marketplace used to conduct the survey was *Freelancer.com*,[1] via which a total of 85 professional developers where hired to fill out the survey. We chose Freelancer.com because it offers the "Pay for Time" option, which we consider ideal for rewarding tasks such as completing a questionnaire comprising a relatively short period of time.

Although an ideal population sample should be drawn randomly, the costs of hiring many developers and only using the data of a few made this rather prohibitive. Instead, we sampled participants based on their *bidding* on our task (which solely consisted of filling out a survey). In the initial selection, only those participants were selected for which it was clear from their bid that they understood the task. Later, an few additional participants were removed because their response (or the timing of their response) led us to assume that they did not fill out the survey with enough attention. All participants were paid an hourly rate according to their bids, for up to 30US$/hr and up to 2 hrs. The task details and payment details were announced in the Freelancer announcement.

For a detailed discussion of the rationale for, and potential of freelance marketplaces for conducting Software Engineering studies, we refer to our earlier paper [37]. As discussed there, one important aspect to keep in mind is that these marketplaces are populated both by individual professionals (i.e., traditional freelancers) and by software engineering companies acting as a single entity on the marketplace.

C. Preparing the Survey Instrument

We defined a set of background information to collect to characterize developer profiles. This information includes: age, country, gender, predominant roles, programming language expertise, familiarity with programming paradigms, and working experience (in months and in code size).

A 5-point ordered response scale ("Likert-scale") is used for asking the developers to describe: (a) their level of knowledge on code smells, (b) their perception on the degree of criticality of code smells, and (c) their perception on how useful code smell information is for different software engineering tasks.

We also asked the developers to identify one or more from a collection of information sources that helped them to get to know about code smells. We include an option "other sources" to allow the respondents to mention any additional sources of information on code smells.

A set of open-ended questions is used to investigate: (a) the rationale behind the perceived criticality of code smells, (b) which code smells are considered as the most critical, (c) which code-smell related tools have been used previously (and their experience using them), and (d) what are the desired features in a code smell detection tool.

We follow the general code smell questions with a three-option question about the respondent's refactoring habits, to investigate the level of planning involved with refactoring activities (i.e., if they refactor as the project progresses, if they plan ahead to perform refactoring, or if they do a combination of both). This is followed by a few detailed questions to characterize if, how often, and with what kind of support, code smells are removed in practice.

Before conducting the survey we cross-examined and clarified the questions and options for answering with help of a third researcher not involved in our study. Due to time constrains, we did not perform a pilot study on the target audience, although this could have exposed a question on code size that was open to misinterpretation (as we see later).

The list of survey questions is presented in Appendix A.

D. Administering the Survey and Analyzing the Data

The code smell survey was part of a larger survey amongst professional developers conducted at Simula Research Laboratory. The whole survey consisted of two parts: one part on estimation of software tasks and one part on code smells It was registered and administered via Qualtrics Research Suite,[2] an online platform for conducting surveys.

The responses on questions that used an ordered response scale based questions were analyzed and summarized via percentage graphs. The response to the open question about justifying the criticality of code smells, was analyzed via *open and axial coding* [38]. Codes where extracted from the statements given by the respondents. In relation to which smells were considered most critical, the responses were interpreted and grouped into several discernible code smells and anti-patterns. For analyzing the features desirable in a code smell tool, we also interpreted the text and extracted concrete features from each response and grouped them according to our perceived level of similarity in each of the responses.

Although no formal inter-rater agreement tests were conducted, each of the authors conducted the qualitative data analysis individually, and the 3 classification differences were discussed and solved by consensus in a subsequent stage.

[1] Formerly active under the names RentACoder and VWorker.

[2] http://www.qualtrics.com

TABLE I
COUNTRIES OF THE RESPONDENTS

Country	No.	Country	No.	Country	No.
Australia	1	Bangladesh	3	Brazil	1
Chile	1	China	2	Croatia	1
Egypt	1	El Salvador	1	Finland	1
France	1	Germany	1	Hungary	1
India	12	Israel	1	Italy	2
Latvia	2	Lithuania	1	New Zealand	1
Nigeria	1	Pakistan	8	Poland	1
Portugal	1	Romania	8	Russia	1
Serbia	1	Thailand	1	UK	4
USA	10	Vietnam	3		

IV. RESULTS AND DISCUSSION

A. Background and Skills of Respondents

In total, 73 out of 85 developers fully completed the survey, yielding a response rate of 86%. The respondents originate from 29 countries, indicating good international coverage (see Table I). A few countries stood out in terms of number of participants, such as India, USA, Pakistan and Romania.

The age of the respondents ranged from 19 to 53 (average 30.9, median 30), and their industrial experience ranged from 1 to 30 years (average 8.7, median 7, Figure 1 shows the distribution). Most of the participants were male, with a total of 69 (95%) male developers and 4 (5%) female developers. In terms of roles in their daily projects, the majority of respondents indicated that they worked as a developer, followed by the role of team lead. Table II shows an overview of all roles, ordered by frequency.

Figure 2 presents an overview of the self-assessed skills in terms of familiarity with different programming languages and programming paradigms. With respect to their proficiency in programming paradigms, the majority of the respondents reported to be confident with OO-programming paradigm (58% chose "Extremely familiar"). For imperative programming, the groups were evenly distributed, and, perhaps somewhat surprising, a larger group reported to feel quite confident in functional programming than in imperative programming (29% for "Moderately" and 22% for "Extremely" familiar).

When analyzing the self-assessment of programming language skills, we saw that the majority of the respondents were not confident in Python (70% responded "Not at all familiar") and Visual Basic (44% in the same group). Javascript constitutes the programming language for which the majority (34% of the respondents) responded "Somewhat familiar". The C, C++, C# and Java programming languages showed a relatively even distribution (with Java having a quite dominant group of nearly 50% of the developers responding that they had no to slight Java skills). Based on these results, we can assume that the respondents have a fair (but not a strong) understanding of OO programming principles. This consideration is of relevance because many code smell definitions are intertwined with OO design principles.

TABLE II
PREDOMINANT ROLE OF THE RESPONDENTS

Category	No. (%)	Category	No. (%)
Developer	48 (66%)	Self-employed	3 (4%)
Team Lead	13 (18%)	Tester	0 (0%)
Architect	5 (7%)	QA Manager	0 (0%)
Project Manager	4 (5%)		

Other programming languages reported by the respondents were Perl, PHP and Ruby. Although we did not ask for the primary working domain, the survey data suggests that the respondents are rather well-acquainted with web-applications, based on their skill-assessment of Javascript and the fact that many of them mentioned PHP (20) and Ruby (6). Initially, we had intended to triangulate the skill self-assessment with the size of the code (in kLOC) that respondents had produced in these languages or paradigms. However, we discarded the size data because many participants reported unrealistically high values, seemingly confusing kLOC and LOC values.

B. Level of Insight in Code Smells

From the total set of respondents, up to 23 (32%) replied that they have never heard of code smells nor anti-patterns (Figure 3). From the remaining 50, the great majority (37, 50%) belonged to either group 2 (i.e., "I have heard about them in blogs or discussions but I am not so sure what they are.") or group 3 (i.e., "I have a general understanding, but do not use these concepts."). Thus only 18% of the respondents

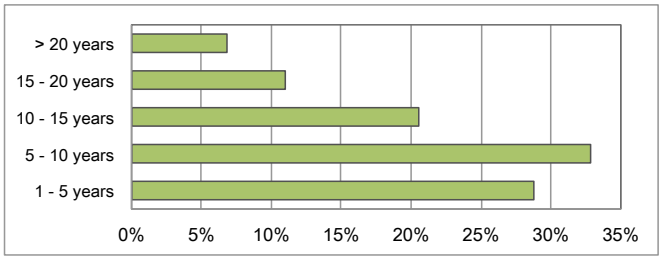

Fig. 1. Experience of the respondents

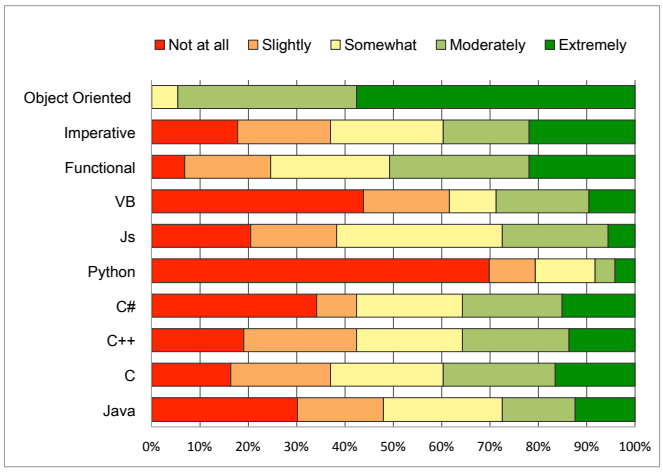

Fig. 2. Familiarity with Programming Languages & Paradigms

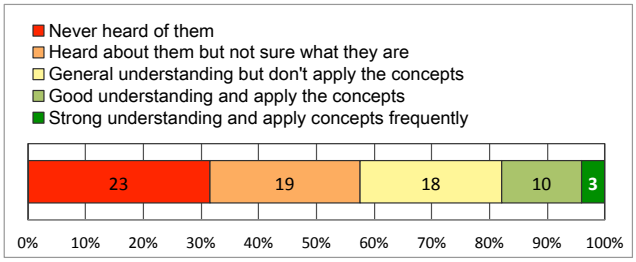

Fig. 3. Knowledge about code smells and anti-patterns

indicated that they had a good or strong understanding about code smells and applied these concepts in their daily activities.

The analysis on information sources will include only the respondents that belong to this last group. When asked about the most common source of information for code smells or design-patterns, *Blogs and developer forums* were the most answered amongst the respondents (see Figure 4). The least mentioned sources are *Tool vendor sites* and *scientific papers*, which constituted, each 4% of the options chosen by the respondents. It is interesting to note that amongst *other sources* than the predefined ones that were mentioned by the respondents, *colleagues*, and *seminars* were a frequent source of information (20%).

These findings suggest that, to increase research impact on industry, the code smell analysis communities' findings and tools should be easily accessible via resources such as Internet forums, technical blogs and industry seminars because that is where professional developers collect their information.

C. Perceived Criticality of Code Smells and Anti-Patterns

With respect to the level of concern that respondents expressed about the presence of code smells, the majority of the developers (19 respondents) mentioned that they were moderately concerned about the presence of code smells in their source code. A very small selection of 6% (3 respondents) were not concerned at all, whereas 14% (7) responded that they were extremely concerned (see Figure 5).

In order to better understand the rationale behind the perceived criticality of code smells on software evolution,

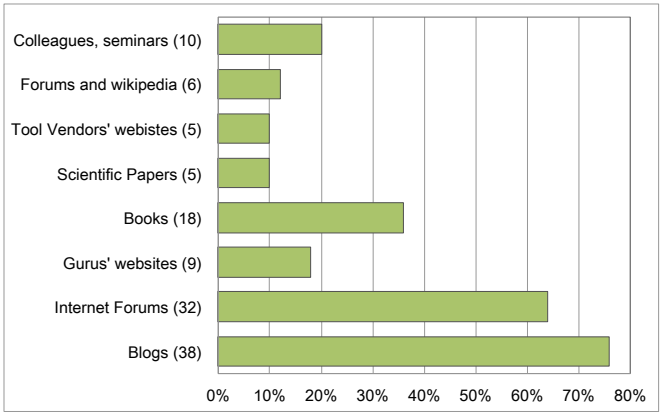

Fig. 4. Sources of information on code smells/anti-patterns

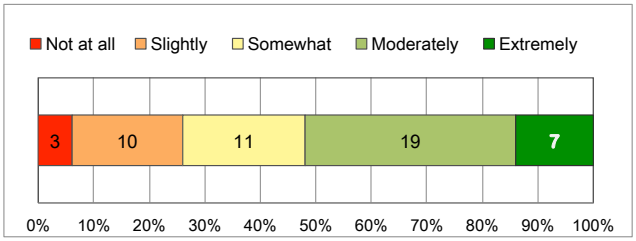

Fig. 5. Concern about the presence of code smells and anti-patterns

we used coding techniques following grounded theory [38] to analyze the answers to the open questions.

Table III shows some examples of the statements, and the codes assigned to each. The complete list of statements with the coding were not included due to space limitations, but are available in the technical report that accompanies this paper [39]. In total, we extracted 32 codes during open coding which were grouped using axial coding into 10 higher abstraction categories. These 10 categories can be thought of as *rationales* and they can be characterized as follows:

1) *Developer productivity:* This category captures rationale in connection to productivity and efficiency in a project, at individual and at group levels (e.g., the amount of desired outcome per unit of time). Codes that belong to this category are: 'Effectiveness', 'Efficiency in teams', 'Efficiency', and 'Productivity'.
2) *Product evolvability:* This category relates to potential risks, and issues that occur during the product evolution. The codes that belong to this category are: 'Impact on software evolution', 'Risk', 'Ripple effect', 'Time-consuming debugging', and 'Testability'.
3) *Quality of end-product:* This category is mainly concerned with product quality from the end-user's or customer's perspective. The codes in this group are: 'Product quality', 'Product reliability', 'Error probability', and 'Product performance'.
4) *Self-improvement:* This category covers the rationale of respondents wishing to increase/improve their skills and knowledge (i.e., respondent: "I am very concerned because existences of code smells shows does not show professionalism in coding. It makes me feel like a novice or amateur"). Codes that belong to this group are: 'Improve own skills', and 'Programmer skills/status'.
5) *(Lack of) Developer's skills:* This category groups codes indicating that the respondents did not know enough about the subject, and therefore were not concerned with the presence of code smells (e.g., respondent: "Because I have no such clear idea about it"). The codes in this category are: 'Intuition', 'Lack of knowledge', and 'Programmers skills'.
6) *(Lack of) Organizational support:* This category encompasses cases where respondents explained that they lacked of support from the organization (i.e., their project managers) or the members of their team. Codes in this category are: 'Lack of support by management', 'Difficult to promote', and 'Lack of time'.

TABLE III
STATEMENTS FROM THE RESPONDENTS WITH THEIR CORRESPONDING CODING

ID	Statement	Codes
2	I want to be reliable as a developer, and adhere to universal conventions on software development, so that the code I produce is reliable, performant and adheres to established standards used for software development.	Standard compliance, Improve own skills, Product Quality, Product reliability
30	Code smells - It seems like a pretty small bug to start off and if left ignored could be hugely destructive. We have seen a similar case in one of our projects and this lead to a delay in project when we could have finished way before the timelines.	Product quality, Ripple effect, Impact on evolution
34	There are some code smells/anti-patterns that I don't like and fight/educate against them. / Then there are others like long identifiers, use of literals, premature optimizations, etc. where I'm concerned but choose to ignore most of the time because it's not worth my time to try to convince the developer.	Case-by-case basis, Difficult to promote
40	I'm concerned about code smells because if there will be many of them the programming process and estimate work-hours of my projects can be multiplied because needs of more refactoring.	Efficiency, Productivity
44	I am very concerned because existences of code smells shows does not show professionalism in coding. It makes me feel like a NOVICE or AMATEUR	Programmer skills/status

7) *(Lack of) Tool support:* This category represents respondents stating that they are not concerned with smells because of the lack of tools that can help to detect them.
8) *Cost/benefit considerations:* This category is composed of statements relating to trade-offs between costs (e.g., time available in relation to deadlines) and quality (intrinsic and extrinsic) of the product. Codes in this category are: 'Trade-offs', 'Case-by-case basis', and 'Within project constraints'.
9) *Availability of alternative approaches:* This category represents statements where respondents argued that alternative approaches or concepts can be used instead of code smells and anti-patterns.
10) *Understandability:* This category relates to concerns on readability and comprehensibility of the source code, and approaches to improve these aspects. Codes included in this category are: 'Code aesthetics', 'Code understandability', 'Compliance to standards/known practices', and 'Software inspection/code review'.

Figure 6 presents an overview of the number of times (codes for) these rationales were mentioned by respondents, grouping them on perceived criticality of smells, as stated by the respondent. By analyzing all rationales matching one criticality group (i.e., one single color), we get more insight into the reasons *why* those respondents were, or were not, concerned about the presence of code smells or anti-patterns.

For example, the rationales *Quality of end-product*, *Product evolvability*, and *Developer productivity* were the most frequent for developers who responded that they were either moderately or extremely concerned about code smells/anti-patterns. Amongst the respondents belonging to the group that was somewhat concerned with smells, the rationale for their concern was diverse, covering all categories except for *Self-improvement*. The rationale for concern given by respondents that were slightly concerned by smells was also diverse, including concerns for reduced *Understandability*, and only being slightly concerned by smells because alternative quality evaluation approaches exist/were used.

To investigate if the respondent's background (i.e., role, experience, familiarity with programming languages/paradigms, and familiarity with code smells) could explain their perception of criticality, we conducted a *Categorical Regression Analysis* [40, 41] using SPSS.[3] Categorical regressions enable the quantification of categorical data by assigning numerical values to the categories, resulting in an *optimal linear regres-*

[3] http://www.ibm.com/software/analytics/spss

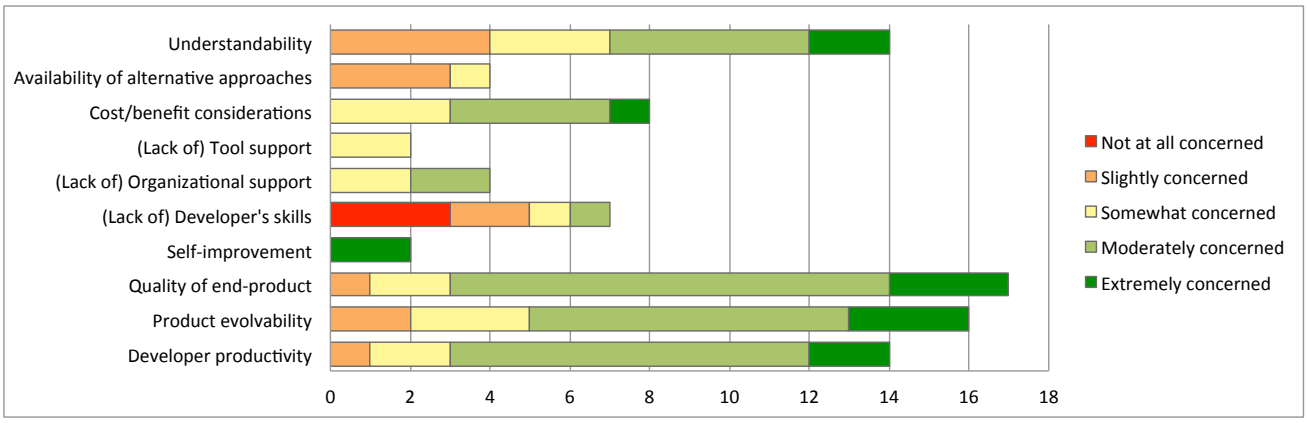

Fig. 6. Frequency of concern categories across respondent groups

sion equation for the transformed variables [40]. This type of analysis supports the development of predictive and explanatory models where both the dependent and independent variables can be of continuous, ordinal, or nominal scale.

The result of the analysis was that none of the background variables except for a respondent's *familiarity with code smells* had a significant contribution to the perceived level of criticality of code smells for that respondent ($B = .388$, $p < .05$). This is in accordance to our observations from Figure 6 and signifies that, in general, the more details people actually know about code smells, the more concerned they are by the presence of smells in their code.

Although we could not find a significant effect of the respondent's role on the level of criticality, we did notice that respondents who had the role of *Project Manager* tended to be less concerned. In contrast, respondents who defined their role as *Self-employed* were much more concerned (Figure 7).

D. Ranking of Code Smells and Anti-Patterns

In total, 34 code smells/anti-patterns were mentioned by the respondents. To summarize this data into a ranked list, we used the *Borda count* [42]. This is a rank-order aggregation technique where, if there are n candidates, the first ranked candidate will get n points, $n-1$ points for a second preference, $n-2$ for a third, etc. By weighing a series of "votes" for each respondent, the Borda count yields a consensus-based ranking instead of a majority-based one. We used a small variation where the first-ranked candidate receive one point, the second-ranked candidate receives half of a point, the third-ranked candidate receives one-third of a point, etc. We chose this variation to keep the aggregated points at a manageable level, due to the large set of candidates. Table IV shows the results from the aggregation, where smells that "scored" 1 point or less where excluded for brevity. This table shows that Duplicated code was by far the most mentioned smell, followed by smells/anti-patterns related to size and complexity: Long Method, Large Class and Accidental Complexity.[4]

E. Use of Analysis Tools in Practice

We asked the participants to indicate which tools they had used for analysis of code smells or anti-patterns and to comment on the usefulness of these tools. Not all participants answered this question, and the ones that did in general interpreted the question wider than just tools aimed at code smells. Instead they reflected on all tools and plug-ins that they used to assure software quality. We received 50 of 73 responses, of which 15 (30%) answered that they used one or more tools and 35 (70%) of them did not use additional tools.

With respect to code smell related tools that were used, only two of the respondents (4%) used specific code smell detection tools and they also used refactoring tools in connection with these to remove the smells. The tools were JetBrains Resharper which combines smell detection and refactoring, and the combination of DevExpress CodeRush and Refactor!Pro.

We learned that the most popular tools were automated software inspection tools that help to adhere to coding standards and detect potentially problematic code patterns as a means to ensure software quality. These were used by 9 respondents and included tools such as CheckStyle, FindBugs, FxCop, and PMD. The next largest group of tools were used by 6 respondents and were dedicated to analyzing performance of web applications. This included tools such as FireBugs, Pagespeed, Yslow and Pingdom. Five of the respondents mentioned that they made use of the exploration and code organization features in their IDEs (either Visual Studio or Eclipse) to do manual code reviews. Three respondents used tools to detect duplicated code (clone detection). Other tools mentioned included a pretty-printer/beautifier to ensure readability of the code and a tool to identify potentially difficult code by means of computing the cyclomatic complexity.[5]

Unfortunately, the respondent's answers on tool usefulness were too divergent to detect consistent patterns. Where one developer found a given tool very useful in their context, another developer would respond that they did not feel the

[4] Accidental complexity signals a mismatch between the degree of complexity in the solution and the complexity of the problem to be solved.

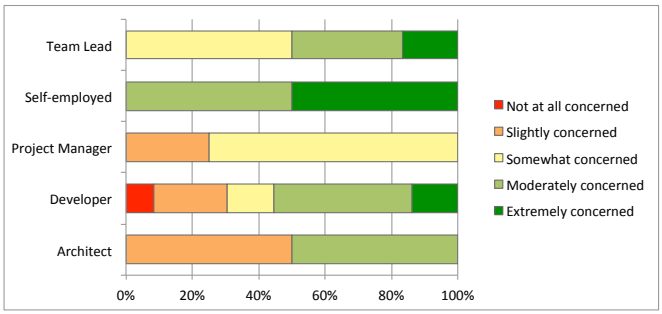

Fig. 7. Role vs. Perceived Criticality

[5] Note the total number of tools used is larger than the number of respondents because some people used more than one tool.

TABLE IV
RANKING OF MOST POPULAR CODE SMELLS/ANTI-PATTERNS

Smell/Anti-Pattern	Points
1. Duplicated code	19.53
2. Long method	9.78
3. Accidental complexity	8.32
4. Large class	7.09
5. Excessive use of literals	3.04
6. Suboptimal information hiding	2.70
7. Lazy class	2.33
8. Feature Envy	2.33
9. Long parameter list	2.31
10. Dead code	2.25
11. Bad (or lack of good) comments	1.50
12. Use deprecated components	1.50
13. Single Responsibility	1.20
14. Complex conditionals	1.12
15. Bad naming	1.12

TABLE V
MOST DESIRED FEATURES FOR CODE SMELL ANALYSIS TOOLS

Feature	Points
1. Detection of duplicated and near-duplicated code	10.00
2. Dynamic analysis (number of calls, etc)	4.08
3. Define and customize detection strategies	3.50
4. Support code inspection	2.67
5. Suggest refactorings	2.50
6. Good usability	2.50
7. Detect potentially problematic areas	2.33
8. Real-time update	2.33
9. Detect memory leaks	2.25
10. Detect dead code	2.03
11. Integrate with versioning & deployment infrastructure	2.00
12. Report generation	1.33
13. Integration with IDE	1.33

tool contributed much. Overall, most developers were positive about the benefits of the automated software inspection tools, there were complaints about the number of false positives reported, and the respondents reflected positively on how some of the tools did *not* intrude on their development work-flow.

F. Desired Features for Code Smell Related Tools

We extracted 29 desired features or characteristics for tools supporting detection or analysis of code smells and anti-patterns. To summarize the results, we again applied the variation on the Borda count aggregation technique that was described above. Table V shows those features or characteristics that scored higher than 1 point. Not surprisingly in the light of the most mentioned smell, the most desired feature by the respondents was the detection of duplicated and near duplicated code. In addition, respondents would like to see support for dynamic analysis, closely followed by desire to define and customize detection strategies (e.g., detection rule templates) based on their project context.

One of the respondents answered that they would like a better version of their current tool, requiring readily usable but customizable rules for detecting problematic code. This corresponds with the fact that developers ranked "support code inspection" and "suggest refactorings" as the most desirable features after the "customizable detection strategies".

G. Usefulness of Code Smells, and Refactoring Habits

Based on the answers of the 50 respondents that had previously used code smell related tools, we analyzed their responses with respect to the usefulness of code smells. Approximately half of the developers expressed that code smells/anti-patterns can be either moderately or extremely useful for conducting different activities in a project (Figure 8). The overall distribution was quite balanced with two activities scoring a bit better: 29 (60%) of the developers responded that code smells and anti-patterns can be moderately or extremely useful for *Code Inspection*, and 33 (66%) responded that they can be moderately or extremely useful for *Error Prediction*.

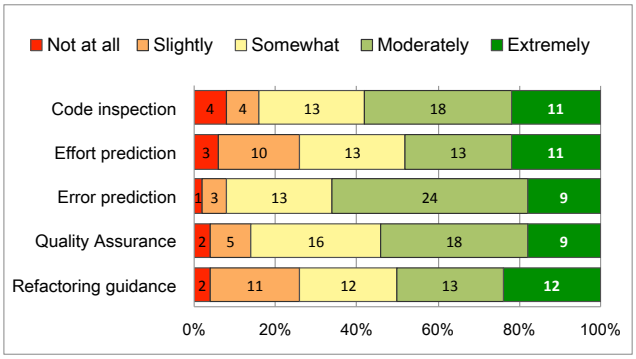

Fig. 8. Usefulness of code smells/anti-patterns for various activities

Finally, with respect to the refactoring habits of the respondents, 17 (34%) answered that they refactor *on the fly*, 8 respondents (16%) answered that they *plan for refactoring* in their projects, and 25 (50%) of the respondents replied that they follow a combination of these strategies.

H. Limitations of the Work

The main threat to the external validity of survey results is attracting a large enough and representative sample of the population, which is generally difficult to obtain for studies on professional software engineers. We used outsourcing on freelance marketplaces to overcome this challenge. We refer to our companion paper for a detailed discussion of the challenges and opportunities of this approach [37], but repeat here that the challenges lie in uncertainty about background and skills of participants, and it is difficult to objectively assess aspects such as competence, education and experience.[6]

The main limitations of the particular survey questions come from their exploratory nature. One example is that we did not collect further information on the expertise and "business domain" of the respondents. In addition, we had to dismiss responses to one of the questions because it was open to interpretation. Additional information might have explained better why many of them were not better acquainted with code smells, and what were the reasons for the (dis)interest.

Another limitation is that is hard to control for completeness and clarity in open responses within a survey. This means answers may be incomplete, or may require interpretation by the researcher. As a result, the background of the researchers that interpreted the answers can become a source of bias.

Nevertheless, considering there was little prior knowledge on the topics of this survey, we believe it was a good decision to use open questions as a first exploratory step. The analysis of responses presented in this paper can now act as a basis for developing further concrete research questions.

V. CONCLUSION AND FUTURE WORK

This paper reports on the findings from a survey conducted on 85 software professionals, in order to better understand

[6] Note that we have left out education levels in our survey as these are next to impossible to compare internationally.

the level of knowledge about code smells, their perceived criticality and the usefulness of code smell related tooling.

We found that a considerably large proportion (32%) of respondents stated that they did not know about code smells. Respondents indicated that they use technical blogs, programmer forums, colleagues and industry seminars as their main sources of information. Our advise is that the research community should target these channels to make findings and tools easier to access, and increase the impact on practice.

With respect to the perceived criticality of code smells and anti-patterns, the responses where divided, but the majority of respondents were *moderately concerned*. Deeper analysis of the responses to open questions showed that respondents who did not care at all about code smells also indicated that they did not know much about smells and anti-patterns. Respondents who were *extremely* or *moderately* concerned gave as rationale reasons like product evolvability, end-product quality, and developer productivity. Respondents who were somewhat concerned about code smells indicated that it is often difficult to obtain organizational support, that they lacked adequate tools, and that they often need to make trade-offs between code quality and delivering a product on time.

Looking at individual smells and anti-patterns, *Duplicated Code* was mentioned most by the respondents, followed by smells and anti-patterns related to code size and complexity, such as *Large Class*, *Long Method*, and the anti-pattern *Accidental Complexity*. Identification of the latter could be an interesting problem for the research community to work on, but will be far from trivial to assess automatically.

Finally, with respect to tool support, the majority of respondents expressed the need for better tools to detect duplicated code/duplicated logic (another sign that the research community's result are not easily accessible for practitioners), and for customizable detection strategies that would enable context-sensitive (or domain specific) detection of code smells.

In general, we found that software professionals who are interested on code smells and anti-patterns expressed a need for better support during the software evolution cycle. More specifically they expressed the need for a user-friendly, real-time tool support for conducting code inspections, which could ultimately help them to identify problematic areas (e.g., using error prediction). Refactoring tools should provide better support for understanding which choices developers have for refactoring/restructuring their code to improve the quality.

As future work, we intend to contact some of the respondents of the survey and conduct a semi-structured interview in order to investigate in detail the motivation and challenges for using code smells during software evolution, and to investigate specific features that should be supported in a tool. We also plan to conduct a more extensive, structured survey, based on the answers obtained from this exploratory study, involving a larger sample of software professionals.

Acknowledgments: The authors thank Magne Jørgensen for helping us collect the data for this study. This work was partly funded by Simula Research Laboratory and the Research Council of Norway through the project EvolveIT (#221751).

REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[2] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2005.
[3] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Working Conf. Reverse Eng.*, 2001, pp. 97–106.
[4] R. Marinescu and D. Ratiu, "Quantifying the quality of object-oriented design: the factor-strategy model," in *Working Conf. Reverse Eng.* IEEE, 2004, pp. 192–201.
[5] R. Marinescu, "Measurement and quality in object-oriented design," in *IEEE Int'l Conf. Softw. Maintenance*, 2005, pp. 701–704.
[6] N. Moha, Y.-g. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," in *21st IEEE/ACM Int'l Conf. Automated Softw. Eng.* IEEE, 2006, pp. 297–300.
[7] N. Moha, "Detection and correction of design defects in object-oriented designs," in *ACM SIGPLAN Conf. Object-oriented programming, systems, languages, and applications*, 2007, pp. 949–950.
[8] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien, "A domain analysis to specify design defects and generate detection algorithms," in *Fundamental Approaches to Softw. Eng.*, 2008, pp. 276–291.
[9] A. A. Rao and K. N. Reddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," in *Int'l MultiConf. Engineers and Computer Scientists*, 2008, pp. 1001–1007.
[10] E. H. Alikacem and H. A. Sahraoui, "A Metric Extraction Framework Based on a High-Level Description Language," in *IEEE Int'l Conf. Source Code Analysis and Manipulation (SCAM)*, 2009, pp. 159–167.
[11] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *Working Conf. Reverse Eng.* IEEE, 2009, pp. 75–84.
[12] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.
[13] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Formal Aspects of Computing*, vol. 22, no. 3, pp. 345–361, 2010.
[14] R. Peters and A. Zaidman, "Evaluating the Lifespan of Code Smells using Software Repository Mining," in *European Conf. Softw. Maintenance and ReEng.* IEEE, 2012, pp. 411–416.
[15] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *IEEE Int'l Conf. Softw. Maintenance*, 2012, pp. 306–315.
[16] A. Yamashita and L. Moonen, "Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study," in *Int'l Conf. Softw. Eng.*, 2013, pp. 682–691.
[17] R. C. Martin, *Agile Software Development, Principles, Patterns and Practice*. Prentice Hall, 2002.
[18] A. J. Riel, *Object-Oriented Design Heuristics*, 1st ed. Boston, MA, USA: Addison-Wesley, 1996.
[19] P. Coad and E. Yourdon, *Object-Oriented Design*. Prentice Hall, 1991.
[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1994.
[21] C. Larman, *Applying UML and Patterns*, 3rd ed. Prentice Hall, 2004.
[22] W. Brown, R. Malveau, S. McCormick, and Tom Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.
[23] M. V. Mäntylä, "Software Evolvability - Empirically Discovered Evolvability Issues and Human Evaluations," PhD Thesis, Helsinki University of Technology, 2009.
[24] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, 2007.
[25] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the Impact of Design Flaws on Software Defects," in *Int'l Conf. Quality Softw.*, 2010, pp. 23–31.
[26] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Working Conf. Mining Softw. Repositories*, 2010, pp. 72–81.
[27] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

[28] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *IEEE Int'l Conf. Softw. Maintenance*, 2008, pp. 227–236.
[29] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *15th European Conf. Softw. Maintenance and ReEng.* IEEE, 2011, pp. 181–190.
[30] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *IEEE Int'l Conf. Softw. Maintenance*, 2010, pp. 1–10.
[31] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smells in Object-Oriented Code," in *Int'l Conf. Quality of Information and Communications Technology*. IEEE, 2010, pp. 106–115.
[32] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells," in *Ws. Refactoring tools (WRT)*. New York, New York, USA: ACM Press, 2011, pp. 33–36.
[33] E. Murphy-Hill and A. P. Black, "Seven habits of a highly effective smell detector," in *Int'l Ws. Recommendation Systems for Softw. Eng. (RSSE)*. ACM, 2008, pp. 36–40.
[34] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Int'l Symposium on Softw. Visualization (SOFTVIS)*. ACM, 2010, pp. 5–14.
[35] A. Fink, *The Survey Handbook*, 2nd ed. Thousand Oaks, California: SAGE, 2003.
[36] D. F. Bacon, Y. Chen, D. Parkes, and M. Rao, "A market-based approach to software evolution," in *Conf. Object-oriented programming, systems, languages, and applications*. ACM, 2009, p. 973.
[37] A. Yamashita and L. Moonen, "Surveying Developer Knowledge and Interest in Code Smells through Online Freelance Marketplaces," in *User Evaluations for Softw. Eng. Researchers (USER)*, 2013.
[38] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 1998.
[39] A. Yamashita and L. Moonen, "Do Developers Care About Code Smells? An Exploratory Survey," Simula Research Laboratory, Technical Report 2013-01, 2013.
[40] J. J. Meulman, "Optimal scaling methods for multivariate categorical data analysis." SPSS, Inc., Tech. Rep., 1998.
[41] J. P. Van Der Geer, *Multivariate analysis of categorical data: Applications*, advanced q ed. SAGE, 1993.
[42] B. Reilly, "Social Choice in the South Seas: Electoral Innovation and the Borda Count in the Pacific Island Countries," *International Political Science Review*, vol. 23, no. 4, pp. 355–372, 2002.

APPENDIX A
SURVEY QUESTIONS

[NB: the layout below was somewhat adapted to meet space limitations]

Section I: Background

1. What is your predominant role within your organization?

[ ] Developer [ ] Architect [ ] Self-employed
[ ] Team Lead [ ] QA Manager
[ ] Tester [ ] Project mngr

2. What is your level of skill in the following languages? (1=novice, 5=expert, please specify which other languages if relevant):

Language	Level	Language	Level	Language	Level
Java		C#		VisualBasic	
C		Python		other:	
C++		Javascript			

3. What is your level of experience (in kLOC and months) in the following languages?

Language	size	time	Language	size	time
Java			Python		
C			Javascript		
C++			VisualBasic		
C#			other:		

4. Rank the following programming paradigms according to how familiar you are with each? (1=least familiar, 5=most familiar)

Paradigm	Functional	Imperative	Object Oriented
Familiarity			

Section II: Code Smells

5. How familiar are you with code smells or design anti-patterns? (please choose one)
[ ] I have never heard of them
[ ] I have heard about them, but I am not so sure what they are
[ ] I have a general understanding, but do not use these concepts
[ ] I have a good understanding, and use these concepts sometimes
[ ] I have a strong understanding, and use these concepts frequently

6. What are the sources from which you learn on code smells? (multiple choices possible)
[ ] Blogs [ ] Books
[ ] Discussion forums [ ] Research papers
[ ] Guru's websites [ ] Tool vendors' websites

7. How concerned are you with the presence of code smells or anti-patterns in your code? (1=not concerned, 5=very concerned) Please motivate why?

8. Are there specific code smells / anti-patterns that you are concerned about? Please list them in order of their perceived importance.

9. Rank the situations where do you think code smell analysis/tools can be helpful (1=not helpful, 5=essential)

Situation	Level
Refactoring guidance (find out where to refactor)	
Quality assessment (e.g., certification processes)	
Bug prediction (identify code likely to have more defects)	
Effort prediction (identify code that takes time to change)	
Code inspection (prioritize areas of the code to improve)	
Others (mention):	

10. Have you used tools for detecting/analyzing code smells? Which ones?

11. Did you find the tools useful? Why/why not?

12. What features would you like in a tool for supporting detection or analysis of code smells? (list the most important ones first).

No.	Feature

13. Do you remove code smells "on the fly" or you plan and allocate time to "cleanup your code"? (please choose one)
[ ] On the fly [ ] Plan [ ] Combination

Section III: Removal of code-smells

14. How often do you refactor to remove code smells? (choose one)
[ ] Never
[ ] Almost never
[ ] Sometimes, when it is absolutely essential
[ ] On a regular basis
[ ] Refactoring is included as a formal activity within the project

15. Can you characterize how much (seldom/regularly/often) of the refactoring is manual, tool assisted or combined?

Method	Manual	Tool assisted	Combined
Frequency			

16. Can you estimate how much (seldom/regularly/often) of the refactoring done is of low (renaming methods), of medium (relocating classes, extracting methods) or high (modify large segments of the code, replace solutions with the usage of patterns, etc.) complexity?

Refactoring complexity	Low	Medium	High
Frequency			

17. Would you like to know more about code smells / anti-patterns or refactoring? Why?