Journal of Computing and Information Technology - CIT 18, 2010, 4, 333–340
doi:10.2498/cit.1001919

An Empirical Investigation of Code Smell 'Deception' and Research Contextualisation through Paul's Criteria

Steve Counsell, Hamza Hamza and Rob M. Hierons

School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, United Kingdom

Code smells represent code decay and as such should be eradicated from a system to prevent future maintenance problems. A range of twenty smells described by Fowler and Beck each require varying numbers and combinations of refactorings in order to be eradicated — but exactly how many are needed when we consider related, nested refactorings is unclear. In this paper, we enumerate these refactorings when categorised according to Mantyla's smell taxonomy. We then show how, ironically, the 'smelliest' of smells (and hence most difficult to eradicate) are actually those best understood by developers. So, code smells are not only unpleasant to have around, but are deceptive in their nature and make-up. The study is thus a warning against attempting to eradicate what are seemingly easily eradicated smells — these are often the smells the developer needs to be most wary of. Finally, we incorporate the answers to six questions suggested by Paul for 'How to write a paper properly' to position the paper in a reflective way.

Keywords: refactoring, OO, code smell

1. Introduction

Code smells represent a form of code decay and unless eradicated, can become a source of maintenance problems from both an increased fault and effort investment perspective. Fowler (and Beck) (Fowler 1999) list twenty code smells, covering all aspects of OO 'smelly' code and prescribe how each of these smells can be eradicated through the application of one or more refactorings (Counsell et al. 2006; Demeyer et al. 2000; Fowler 1999; Kerievsky 2005). What is not clear from this listing is a) how many related refactorings each requires for its eradication, b) which are the 'smelliest' smells (in terms of refactorings required for their eradication) and c) the views of which smells are understood "most" and "least" by developers. In this paper, we use a smell taxonomy of Mantyla (Mantyla et al. 2003) to determine the range of required refactorings by each smell when placed in categories defined by that taxonomy; we then establish whether there is any correlation between 'smelliness' and an industrial survey of smells by Mantyla using developers and according to the smells they understand the 'least' and 'best' (Mantyla 2003).

The remainder of the paper is organized as follows. In the next section, we describe the motivation for the study and related work. In Section 3, we describe the smell taxonomy of Mantyla and the decomposition of those smells to highlight an important feature of the categories in that taxonomy (Section 4). We then demonstrate how the developer survey of Mantyla shows that the 'smelliest' smells are those that the developer finds easiest to understand (Section 5). In Section 6, we state and then answer the six questions posed by Paul (Paul 2009) to frame the paper in a reflective sense. Finally, we draw conclusions and point to future work (Section 7). Note — a new reader wishing to appreciate the six answers to Paul's questions before reading the paper content and thus to gain the benefit of the reflection therein is directed to Section 6 first.

2. Motivation/Related Work

The motivation for the research described stems from two sources. First, as a software engineering community, we still know very little about developer perception, intent and habits. In particular, we know very little about the views on code smells and their 'smelliness' (i.e., an indication of required effort to eradicate those smells). In this sense, the study we present attempts to form a link between the theoretical and the practical, industry-views. Second, the three studies of Mantyla et al. (Mantyla 2003; Mantyla et al. 2004; Mantyla et al. 2006a; Mantyla et al. 2006b) represented landmarks in the analysis of code smells, but as yet have not been studied in anger with any other analyses. This represents a gap in the area. Third, a previous paper by Counsell et al. (Counsell et al. 2010) has suggested that no strategy currently exists for the assessment of code smells, even though there is significant industrial resonance. The research in this paper is a small step in this direction. Finally, much research has been done in the field of refactoring recently and in seminal work traceable to 1992 (Opdyke 2002; Counsell et al. 2008), with which code smells have a strong relationship. The presented research places some of that previous research in context.

In terms of related work, a well-known 'taxonomy' for allocating code smells (and which we use in this paper) was proposed by Mantyla — an empirical study of industrial developers and their opinion of smells was also reported; a *précis* of the same results was presented in (Mantyla et al. 2004). Both studies gave insights into which smells developers most 'understood', least understood and hence those that they would be most or least likely to want to eradicate. In subsequent work, Mantyla and Lassenius (Mantyla et al. 2006b) also described mechanisms for making refactoring decisions based on smell identification and also for evolving systems (Mantyla et al. 2006a). Counsell et al. reported a link between code smells and refactoring in terms of in — and out-degrees on a dependency diagram (Counsell et al. 2006) and this was supported with empirical open-source data. Research by Khomh et al. has recently explored code smells using the Bayesian approach (Khomh et al. 2009b) — the same authors have looked at changes made to a system as an indication of smells (Khomh et al. 2009a); on a slightly different tack, the interrelationships between smells was explored by Pietrzak et al. (Pietrzak et al. 2006). Finally, in a similar vein to the study presented, Hamza et al. (Hamza et al. 2008) decomposed both Fowler's and Kerievsky's code smells (Kerievsky 2005) to determine smell overlap and commonality. Refactoring was used as a basis for that decomposition (Mens et al. 2003; Mens et al. 2004; Najjar et al. 2003). Finally, Munro (Munro 2005) describes a set of product metrics that can be used to guide the developer to bad smells in code and a tool was developed to aid this process.

3. Smell Taxonomy

For our analysis, we use the smell taxonomy described by Mantyla (Mantyla 2003), the purpose of which is to try and understand, appreciate and tackle the composition, relationships and commonalities between the set of code smells. Table 1 shows the taxonomy of smells according to the five categories.

The *Bloater* smells reflect *something* that has become so large that it can no longer be managed efficiently (e.g. 'Large Class').

Group	Smells in group
Bloaters	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps.
Object-Orientation abusers	Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces.
Change Preventers	Divergent Change, Shotgun Surgery, Parallel Inheritance Hierarchies.
Dispensables	Lazy class, Data class, Duplicate Code, Dead Code, Speculative Generality.
Couplers	Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man.

Table 1. Mantyla's Smell Taxonomy (Mantyla 2003).

The *Object-Orientation abusers* category contains situations where obvious and intuitive features of OO are not fully exploited (e.g., 'Switch Statements' — where polymorphism should be used instead).

The *Dispensable* smells represent 'something' that needs to be removed from the source code (e.g., code that is duplicated).

The *Couplers* reflect harmful and excessive coupling in code (e.g., Feature Envy — where one class uses the features of another class excessively).

Where appropriate, we describe the meaning of a smell.

4. Smell Decomposition

To inform our analysis, a bespoke software tool was developed to generate the number of refactorings for each of the twenty-two code smells listed by Fowler. The algorithm for generating the refactorings was based on a recursive tree-based search of all dependencies between the seventy-two refactorings of Fowler in their respective texts. The tool is described in more detail in Hamza et al. (Hamza et al. 2008), to which the reader is directed.

Table 2 lists each of the code smells and the number of refactorings that each requires to be eradicated (in parentheses). For example, the 'Alternative Classes with Different Interfaces' smell (bolded in Table 2) requires the application of two refactorings in order to be eradicated. This smell occurs when two classes have a similar internal content, but different external composition (i.e., in the parameter list) — they should thus be amalgamated to present a common interface. From Fowler (Fowler 1999), the two refactorings it requires are 'Rename Method' and 'Move Method'. The smells that require the fewest refactorings are the 'Divergent Change' and 'Message Chains' smells (both bolded in Table 2). Divergent Change is a smell which occurs when a class has to be changed frequently in response to a range of change types. The Message Chains smell occurs when a series of objects need to be used to facilitate a relatively simple call (the series should be eliminated).

The smell that requires the highest number of refactorings is the 'Primitive Obsession' smell (requiring 7 refactorings), which arises when there is an over-reliance on primitive data types in a class. Interestingly, the Primitive Obsession refactoring requires the use of many 'replacement' type functions as part of its eradication: e.g., Replace Data Value with Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses and Replace Type Code with State/Strategy refactorings; all five are used by this smell. Table 2 also shows that, on average, the Bloaters category is the set of smells that requires the most effort to eradicate based on number of refactorings (averaging 4.20). The Change Preventers have the lowest average number of refactorings required to be eradicated (averaging 2.0).

Group	Smells in group
Bloaters	1. Long Method (4), 2. Large Class (4), 3. Primitive Obsession (7), 4. Long Parameter List (3), 5. Data Clumps (3).
Object-Orientation abusers	6. Switch Statements (5), 7. Temporary Field (2), 8. Refused Bequest (1), **9. Alternative Classes with Different Interfaces (2)**.
Change Preventers	**10. Divergent Change (1)**, 11. Shotgun Surgery (3), 12. Parallel Inheritance Hierarchies (2).
Dispensables	13. Lazy class (2), 14. Data class (3), 15. Duplicate Code (4), 16. Speculative Generality (4).
Couplers	17. Feature Envy (3), 18. Inappropriate Intimacy (5), **19. Message Chains (1)**, 20. Middle Man (3).

Table 2. Smells and number of refactorings.

4.1. Nested related decomposition

The smells in Table 2, while listing the number of refactorings required to eradicate each smell, give only a superficial impression of the actual effort required to eradicate a smell. This claim is made on the basis that each of those refactorings may, in turn, require *other* further refactorings to be completed. For example, smells 9, 11, 12, 14, 17, 18 from Table 2 all require the application of the 'Move Method (MM)' refactoring. The MM refactoring moves a method to a class where it is more conveniently situated

to reduce coupling. The MM refactoring itself requires a set of further refactorings, each of which themselves may require further refactorings (Fowler 1999). Figure 1 therefore shows the 'actual' number of refactorings that each of the twenty smells from Table 2 generate when we analyzed their related refactorings. Smell 1 — 'Long Method' requires 20 'actual' refactorings and smell 2, 'Large Class' requires the application of 40 actual refactorings. In fact, there is a clear trend for smells in the Bloaters category (i.e., smells 1-5) to require relatively more actual refactorings than any other category.

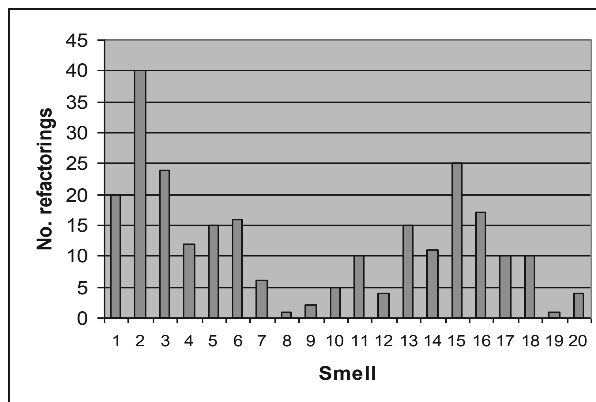

Figure 1. Smells and 'actual' refactorings.

On the other hand, smells in the Object-Orientation Abuser category (smells 6-9) have an average of just 6.25 refactorings. The only category that has a similar number of actual refactorings to the Bloaters is the Dispensable category (smells 13-16) with an average of 17.0 refactorings. Clearly, attempting to eradicate one the Bloater smells is likely to require more effort than any other category (in theory and on average). Large Class generates higher numbers of 'actual' refactorings than any other smell. Smells 8 and 19 (Refused Bequest and Message Chains) are the smells that generate least refactorings. Interestingly, the first is an inheritance-based smell and the second coupling-based; inheritance and coupling are difficult features to untangle because they may require the developer to appreciate knowledge of the scope of many classes.

Table 3 shows the refactorings (Refs.) that the Large Class smell requires in order to be eradicated (Fowler 1999). On average, each of the

Smell	Original Refs.	Actual Refs.
Large Class	1. Extract Class, 2. Extract Subclass, 3. Extract Interface, 4. Replace Data Value with Object	40

Table 3. Large Class refactorings.

original refactorings from this smell requires 10 other refactorings to be undertaken. The reason for this high value is relatively straightforward. The Extract Class, Extract Subclass and Extract Interface refactorings 1, 2 and 3 from Table 3 require a wide range of other refactorings due to the number of code dependencies that need to be resolved. For example, the Extract Subclass (refactoring 1 in Table 3) requires the use of no less than 6 other refactorings, each of which may have their own refactorings.

It would appear that a wide variation exists in the effort required to eradicate each set of smells, as well as between individual smells.

5. Mantyla's Developer Survey

In theory, we might reasonably expect developers to choose the smells to eradicate that have a small set of required refactorings. We would also expect the smells that developers 'understand best' to correlate (negatively) with the effort required to eradicate that smell. In other words, a tentative hypothesis is that easily understood smells are easier to eradicate than less easy to understand smells and developers will therefore attempt to eradicate the former. Mantyla (Mantyla et al. 2004) surveyed 12 developers from a Finnish software development company developing in Delphi. The developers were asked for their opinions about the set of different code smells.

Table 4 shows the three smells (a, b, and c) that were most often left as either 'Don't Understand (DU)' or 'Don't Know (DK)' by the developers in the same survey. These were the smells that developers knew about, but did not understand easily (the former case) and simply did not know anything about (in the latter case). In the case of smells b and c (inheritance-based refactorings) Mantyla suggests that there

Smell	DU	DK
a. Data Clumps	3	10
b. Alternative Classes with Different Interfaces	0	7
c. Refused Bequest	0	6

Table 4. 3 least understood smells (from (Mantyla 2003)).

are good reasons why developers 'didn't know' about these smells — manipulating and recalling inheritance structures is fraught with complexity. Mantyla also suggests that in the case of 'Data Clumps' (c), the lack of understanding was due to weakness in the survey style. The data from Table 2 (and Figure 1) shows that Data Clumps has a relatively high number of both related refactorings and 'actual' refactorings, and this may be a factor. The Data Clumps smell uses the Extract Class, Introduce Parameter Object and Preserve Whole Object refactorings; again, we see that extraction of class features can add significantly to the effort of smell eradication.

Table 5 shows the standard percentage responses found in the same study by Mantyla. Only 64.3% of developers responded that they understood the Data Clumps smell. Equally, 100% of developers understood the meanings of the Long Parameter List and Duplicate Code smells. Most interesting from Table 5 is the response rate of smells in the second row of the table — the high response of 97.3% implies that these smells were relatively well understood (cf. Figure 1). The bolded smells in Table 5 are drawn from the Bloaters category of bad smells and many of those bolded smells thus have a relatively high number of actual refactorings from Figure 1. Only the Data Clumps smell is outside the top nine smells from Mantyla's survey. *In other words, the smells that developers tended to understand most were the classes that, from our analysis, tended to require the largest number of refactorings in order to be remedied.* On the other hand, the italicized smells in Table 5 are the five smells with the fewest number of required refactorings from Figure 1 (requiring a low number of actual refactorings). *Counter-intuitively, three of the five were those least understood by developers.*

Smell	%
Long Parameter List, Duplicate Code.	100
Long Method, **Large Class**, *Message Chains*, *Middle Man*, Lazy Class, **Primitive Obsession**, Temporary Field, Shotgun Surgery.	97.3
Speculative Generality, Feature Envy, Switch Statements	94.6
Comments, Incomplete Library Class.	91.9
Parallel Inheritance Hierarchies, Divergent Change, Data Class, Inappropriate Intimacy.	89.2
Refused Bequest.	83.8
Alternative Classes with Different Interfaces.	81.1
Data Clumps.	64.3

Table 5. Code smell responses (from (Mantyla 2003)).

It appears from the preceding analysis that developers understood best the smells that were likely to require *most* effort to eradicate and, conversely, they least understood the smells that appear to require *less* effort to eradicate. Smells therefore provide a facade in terms of what they are, how they are decomposed and how they are resolved — a deception and mask for the real effort required in their eradication.

5.1. Discussion

We note that any study based on empirical data will suffer from a number of threats to its validity.

First, we accept that an assumption of the analysis presented is that each refactoring takes a similar amount of effort to undertake and that this is unlikely in practical reality. However, without the data on timed effort by actual developers, or a set of correspondingly subjective assumptions, we feel that this represents a starting point for an analysis of code smells.

Second, our analysis makes the assumption that a developer has no sense of the effort required to eradicate a smell. In other words, that a developer is oblivious to the presence of smells or the effort required for their eradication. On reflection, this stance could be seen as misguided on our part. A developer might be able to detect

a smell; the same developer might also be fully aware of the implications of leaving that smell to become a "stench" (i.e., an advanced form of smell) or to eradicating that smell.

Third, developers also have to make difficult choices as to how they allocate their time. We have no evidence that developers actively avoid smells, yet there is no evidence, as we presented, to suggest that developers do eradicate smells.

Finally, there may be many other types and variations of code smell that a developer would consider eradicating before those we have considered. We cannot necessarily assume that the 22 smells listed in Fowler (Fowler 1999) are the definitive set of smells that we should adopt. In a commercial setting, a likely recognizable set of smells is that of in-house smells, as sinister as any of those listed in Fowler. Future work will consider an experiment involving actual developers and measurement of times taken for refactoring activities. The dependency graph used as the basis of the present work is described in detail in Counsell et al. (Counsell et al. 2006) and size precludes its inclusion in this paper. The data for this graph can be made available to interested researchers for the replication of studies.

6. Paul's Criteria

One issue that arises with any research paper is that, at some point after publication, even the authors of the paper are unable to remember the message of the paper when they re-visit it, the story it tells and the value of the research to a reader. This has implications for how widespread the research message can become. The purpose of publishing research should be motivated by the desire to extend the boundaries of knowledge in a specific area. To that end, Paul (Paul 2009) has suggested six questions that need to be answered for 'How to write a paper properly'. In this section, we explore those five questions with one objective in mind: to frame and abstract the material in the previous six sections from the perspective of those six questions. The six questions posed by Paul (Paul 2009) and treated individually in turn are as follows:

Question 1: *What 'story' is the paper trying to tell the reader?* Paul provides a number of guidelines as to why this is an important question to answer.

Paper focus: there should be one story, not many. There may be two or three major points to the story, but much more than that is likely to confuse the reader.

Longevity: a story written for the reader should be able to be understood in ten years time by the author if they need to revisit the paper. A story written for the authors only is likely to leave the authors as perplexed in ten years time as readers are now. The authors' response that encapsulates our story is as follows:

Question 1 Response: Code smells are an ongoing problem for developers and project managers alike since they represent a code that badly needs refactoring. Smells vary in the likely effort they require to be eradicated. We would expect 'really' bad smells to require relatively more effort to eradicate. Paradoxically however, the 'smelliest' of smells (and hence the most difficult to eradicate) are actually those that are best understood by developers. Code smells are therefore unpleasant to have around, but more sinisterly are deceptive; they might seem easy to eradicate, but in reality are not. The study is a warning for developers against attempting to eradicate what are seemingly easily eradicated smells — these are often the smells the developer needs to be most cautious of.

Question 2: What will the reader know after reading your story that they did not know before reading the story? Paul provides a guideline as to why this is an important question to answer.

Purpose: What is the point of the paper?

Question 2 Response: The paper provides an insight into the problems that might arise when trying to eradicate code smells. The data presented ties into the first motivation for the broader research question stated in Section 2 — '*the software engineering community still know very little about developer perception, intent and habits... we know very little about the views on code smells and their 'smelliness' (i.e., an indication of required effort to eradicate those smells)*'. Second, the tie-up between the theoretical and practical is often overlooked in research studies. A further point of the paper is informed by the statement in Section 2, which, paraphrased, can be summarized as an attempt

to form a link between the theoretical and the practical, industry-view of code smells. Finally, code smells have been an under-researched area of software engineering, as reported by Counsell et al. (Counsell et al. 2010). A further point of the paper is to explore the area of code smells following the publication of early leading studies. Section 2 states that '*the two studies of Mantyla reported in* (Mantyla 2003) *represented landmarks in the analysis of code smells, but as yet have not been studied in anger with any other analyses*'.

Question 3: Why should anyone believe you? If the paper is not believable, then this will be the downfall of the paper.

Question 3 Response: The paper presents empirical data produced from a tool to support the conclusions it makes. It also presents evidence from a study by Mantyla (Mantyla 2003) to compare empirical data with conclusions from Mantyla's study. We therefore rely on data and conclusions drawn from the data. Although there are various threats to validity (described in Section 5.1), we believe that the conclusions are credible. Of course, a single empirical study will not be definitive on its own and more studies need to be undertaken in this area before a body of knowledge can be formed; this is one of the reasons why we offer to provide the data from the study at the end of Section 5.

Question 4: Why should anyone care about the story being told? Paul provides some guidance on this particular question. The reader needs to attach value to the points being made; without value, the significance of any research contribution can be questioned.

Question 4 Response: Developers and project managers have to make decisions on how to allocate their limited time. Smells are a problem that every developer faces; the choice of what to do about those smells is another question altogether. In other words, the value of the paper is in highlighting the relative effort that smells may induce. If the reader is a developer or project manager, then we have confidence that the value of the research will be high. If the reader is a researcher, then we hope that the research will provide the impetus for further replication studies.

Question 5: In one sentence what is the essence of your paper?

Question 5 Response: We would envisage such a sentence to be: 'Code smells are a development fact of life and should be eradicated, but serious thought should be given to which smells are most practical to eradicate, based on the relative effort that eradication may induce.'

Paul (Paul 2009) also asks (as a final question 6) the authors to consider what motivated their research and the paper written? We refer the reader to Section 2 of the paper and to the elements of our response to question 2, which contains further background to the motivation for the paper.

7. Conclusions/Further Work

In this paper, we have described an analysis of bad code smells. The basis of our analysis is that code smells hide the true effort that they require to be eradicated when nested refactorings are explored. We used the smell taxonomy of Mantyla (Mantyla 2003) to analyze the categories of smells and the refactorings they required. The 'Bloater' category was found to be the most expensive in this sense. Counter-intuitively, many of the smells from this category were best understood by developers as part of a study of developers; on the other hand, the cheapest smells (in terms of perceived effort) were actually the least understood by the same developers. The real stink about smells is therefore their capacity to deceive. As well as providing empirical evidence to support this stance, we provide a critique of Paul's six questions as to what makes a good research paper. Future work will focus on a developer-based experiment to replicate Mantyla's developer survey and on an exploration of the testing implications of smell eradication. The results presented here are the first of many smell studies and we welcome further exploration in this area.

8. Acknowledgments

The research in this paper was supported by a grant from the UK Engineering and Physical Sciences Research Council (EPSRC) Grant number: EP/G031126/1.

References

[1] S. COUNSELL, Y. HASSOUN, G. LOIZOU, R. NAJJAR, Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS. *Proceedings of IEEE/ACM Symposium on Empirical Soft. Engineering*, (September 2006) Rio de Janeiro, Brazil, pp. 288–296.

[2] S. COUNSELL, R. M. HIERONS, Is a Strategy for Code Smell Assessment Long Overdue? *Proceedings of the Workshop on Emerging Trends in Software Metrics, ICSE 2010*, (May 2010) Cape Town.

[3] S. COUNSELL, S. SWIFT, Refactoring Steps, Java Refactorings and Empirical Evidence. *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, COMPSAC 2008, (28 July – 1 August 2008), Turku, Finland. IEEE Computer Society, pp. 176–179.

[4] S. DEMEYER, S. DUCASSE, O. NIERSTRASZ, Finding refactorings via change metrics, *ACM Conf. on Object Oriented Prog. Systems Languages and Applications (OOPSLA)*, (2000) Minneapolis, USA, pp. 166–177.

[5] M. FOWLER, *Refactoring (Improving The Design of Existing Code)*. Addison Wesley, 1999.

[6] H. HAMZA, S. COUNSELL, G. LOIZOU, T. HALL, Code Smell Eradication and Associated Refactoring, *Proceedings of the European Computing Conference (ECC)*, (September 2008) Malta.

[7] J. KERIEVSKY, *Refactoring to Patterns*. Addison-Wesley, 2005.

[8] F. KHOMH, M. DI PENTA, Y. GUEHENEUC, An Exploratory Study of the Impact of Code Smells on Software Change-proneness, *Proceedings of the 15th Working Conference on Reverse Engineering*, (2009a) Antwerp, Belgium.

[9] F. KHOMH, S. VAUCHER, Y. GUEHENEUC, H. SAHRAOUI, A Bayesian Approach for the Detection of Code and Design Smells. In Choi Byoung-ju, editor, *Proceedings of the 9th International Conference on Quality Software (QSIC)*, 2009b IEEE Computer Society Press.

[10] M. MANTYLA, J. VANHANEN, C. LASSENIUS, Bad Smells — Humans as Code Critics. *20th IEEE International Conference on Software Maintenance (ICSM'04)*, (2004) Chicago, USA, pp. 399–408.

[11] M. MANTYLA, C. LASSENIUS, Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. *Journal of Empirical Soft. Engineering*, vol. 11, no. 3, 2006a, pp. 395–431.

[12] M. MANTYLA, C. LASSENIUS, Drivers for Software Refactoring Decisions. *Proceedings of the Intl Symposium on Empirical Soft. Engineering*, (2006b) Rio de Janeiro, Brasil, pp. 297–306.

[13] M. MANTYLA, Bad Smells in Software — A Taxonomy and an Empirical Study. Master's Thesis, Helsinki University of Technology, Software Business and Engineering Institute, 2003.

[14] T. MENS, A. VAN DEURSEN, Refactoring: Emerging Trends and Open Problems. *Proceedings of the 1st Intnl. Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)*, (2003) Univ. of Waterloo.

[15] T. MENS, T. TOURWE, A Survey of Software Refactoring, *IEEE Transactions on Software Engineering* 30(2): (2004), pp. 126–139.

[16] M. MUNRO, Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. *IEEE METRICS*, 15: (2005).

[17] R. NAJJAR, S. COUNSELL, G. LOIZOU, K. MANNOCK, The role of constructors in the context of refactoring object-oriented software. *IEEE European Conference on Software Maintenance and Reengineering*, (March 26-28, 2003) Benevento, Italy, pp. 111–120.

[18] W. OPDYKE, Refactoring object-oriented frameworks, PhD. Thesis, Univ. of Illinois, 1992.

[19] R. PAUL, The Overseas Contribution to the International Conference on Information Technology Interfaces (ITI), Editorial for CIT Special Issue, *Journal of Computing and Information Technology*, 18(2), pp. ii-iv., (2010).

[20] B. PIETRZAK, B. WALTER, Leveraging Code Smell Detection with Inter-smell Relations, *Proceedings of XP 2006*, Oulu, Finland, pp. 75–84, Springer.

Received: June, 2010
Accepted: November, 2010

Contact address:
Steve Counsell
School of Information Systems,
Computing and Mathematics
Brunel University
Uxbridge, Middlesex
UB8 3PH
United Kingdom
e-mail: steve.counsell@brunel.ac.uk

STEVE COUNSELL is a senior lecturer in the Department of Information Systems and Computing at Brunel University. He received his PhD from Birkbeck, University of London in 2002 and his research interests relate to empirical software engineering; in particular, refactoring, software metrics and the study of software evolution. Before his PhD, he worked as an industrial developer.

HAMZA HAMZA is currently a PhD student in the Department of Information Systems and Computing at Brunel University. His research interests focus on the empirical aspects of software engineering and, in particular, that of real-time systems. His first degree was obtained from the University of Aleppo in Syria.

ROB M. HIERONS received a BA in mathematics (Trinity College, Cambridge), and a Ph.D. in computer science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full professor in 2003.