Software Erosion: A Critical Survey on the Scientific State of the Art

Erosion von Software: Was kann uns die Wissenschaft heute dazu sagen?

Prof. Dr. Rainer Koschke, Dr. Jan Harder, Dr. Saman Bazrafshan
Universität Bremen

Zusammenfassung in Deutsch

Software-Erosion ist der schleichende Verfall der inneren Qualität von Software, sichtbar durch eine Ausbreitung so genannter *Bad Smells*. *Bad Smells* (auch Code-Smells oder technische Schuld, engl. Technical Dept, genannt) führen zu einer zunehmenden Erschwerung beim Verständnis, beim Testen und bei der Änderung von Software. Beispiele für Bad Smells sind duplizierter, unnötig komplexer, langer oder gar toter Code. Auf Architekturebenen sind es zyklische Abhängigkeiten, hohe Kopplung und niedrige Kohäsion von Komponenten sowie Verletzungen von Architekturvorgaben. In diesem Beitrag wird der aktuelle Stand der Wissenschaft zu Software-Erosion – ihre unterschiedlichen Formen, Eigenschaften und Auswirkungen – zusammengefasst und noch offene Forschungsfragen zu diesem Thema genannt.

Summary

Software erosion is the slow decay of the inner quality of software visible in a proliferation of *Bad Smells*. *Bad Smells* (also known as *code smells* or *technical debt*) lead to an increasing difficulty of comprehension, test, and modification of software. Examples of *Bad Smells* include duplicated code, unnecessarily complex, long, or even dead code. At the architectural level, *Bad Smells* show as cyclic dependencies, high coupling and low cohesion of components as well as violations of architecture specifications. This article summarizes the scientific body of knowledge about software erosion—its different forms, properties, and impact—and states open scientific questions.

Forms of Software Erosion

Software erosion manifests itself in an increasing number of *Bad Smells*. *Bad Smells*—also known as *code smells*—are structural deficiencies in source code or design artifacts believed to have a negative impact on the maintainability and evolution of software (Fowler, 1999). The criteria for maintainability are comprehensibility, testability, extensibility, changeability, and reusability and other aspects required to maintain and evolve software. Developers polls on *Bad Smells* (Yamashita and Moonen, 2013; Palomba et al., 2014) and analyses of online discussions in developer forums such as StackOverflow (Tahir et al., 2018) have shown that developers do care about *Bad Smells*.

Fowler and Beck gave several examples for *Bad Smells* as a motivation for refactoring (Fowler, 1999). The *Bad Smell Long Method*, for instance, is a method of excessive size. Likewise, a *Large Class* is a class with many attributes and many long methods. A *Long Parameter List* is a method signature with a high number of parameters. *Divergent Changes* describe the problem that the same code entity (for instance, a class) needs to be changed for apparently unrelated reasons from a developer's point of view, which could indicate low cohesion and a violation of

the principle of separation of concerns. *Shot-Gun Surgery* is a modification that requires a large number of scattered code changes.

Number one on the list by Fowler and Beck (Fowler, 1999) is duplicated code—also known as *software clones*. This *Bad Smell* receives also considerable attention in the research community (Koschke, 2007; Roy et al., 2009). The *Bad Smell Duplicated Code*, or *software clone*, refers to a code fragment that re-occurs identically or similarly somewhere else in the program. There are international scientific workshops, namely *International Workshop on Software Clones*, and Dagstuhl seminars (No. 06301 and 12071) just on this single type of *Bad Smells*. There is an abundance of publications on it in virtually all scientific software engineering conferences, thus, we can consider it a research area in its own right. This research has produced many techniques to detect clones and track clones over time as well as empirical evidence for their harmfulness on maintenance Monden et al. (2002); Chou et al. (2001); Li et al. (2006); Jürgens et al. (2009); Göde and Koschke (2011). Yet, clones are just a small fraction of the much bigger problem of *Bad Smells* in general. For instance, Fowler's list consists of 22 different types of *Bad Smells*. Other types of *Bad Smells* receive considerably less attention in research, as Zhang et al. (2011) pointed out in their research survey on *Bad Smells*. Half of the 39 papers in their survey consider software clones. The focus on clones is not substantiated by empirical evidence, however. As Zhang et al. (2011) conclude in their survey, there is still insufficient knowledge regarding the consequences of *Bad Smells* on maintainability due to the lack of empirical research on *Bad Smells*. We do not know how problematic other *Bad Smells* are in maintenance and evolution. By the same token, we do not know whether other *Bad Smells* are even worse or less bad than clones.

For these reasons, in current research within our research group we address *Bad Smells* in general, leveraging our knowledge, methods, and tools on detecting, tracing, and investigating software clones we have gathered and developed over many years. The focus of most research is source code in procedural and object-oriented languages. For the sake of completeness, we note that *Bad Smells* can be found in other language paradigms and artefacts. For instance, an adaptation and extension of *Bad Smells* for aspect-oriented software was introduced by Srivisut and Muenchaisri (2007). They describe four additional *Bad Smells* specific for aspect-oriented software and the associated refactorings to remove them. Likewise, *Bad Smells* exist in architectural descriptions where Garcia et al. (2009a,b) call them *Architectural Bad Smell* as well as in software development processes (Brown et al., 1998). *Bad Smells* can be subsumed by the term *antipattern* coined by Brown et al. (1998). An antipattern describes any kind of re-occuring situation that has a negative influence on a software project including artefacts and development processes. One of the first authors on general antipatterns is Webster (1995) (he did not use this term though).

Research on *Bad Smells* can be divided into various themes including characterization and classification of *Bad Smells*, their origin, detection, removal, their impact on quality, and their use for software quality indexes. In the following, we delve into these topics in more detail.

Classification of *Bad Smells*

Mäntylä et al. (2003) introduce a classification of the *Bad Smells* described by Fowler (1999) into seven categories:

- **Bloater:** *Bad Smells* increasing the size of code, such as *Long Method*, *Long Parameter List* and *Large Class*
- **Encapsulator:** *Bad Smells* on transferring and encapsulating data such as *Message Chain* or *Middle Man*
- **Dispensables:** these are useless code elements such as *Lazy Class*, *Data Class*, *Dead Code*, or *Duplicated Code*

- **Object-Orientation Abusers:** code in object-oriented languages that does not truly follow the object-oriented paradigm; examples include *Temporary Field*, *Refused Bequest*, *Switch Statements*, and *Parallel Inheritance Hierarchies*
- **Coupler:** these *Bad Smells* are about coupling between two or more code entities such as *Feature Envy* or *Inappropriate Intimacy*
- **Others:** all remaining such as *Comments* and *Incomplete Library Class*

This taxonomy is extended by Marticorena et al. (2006) introducing four additional attributes:

- **Granularity:** describes the level of abstraction of a code entity affected by a *Bad Smells* (class, package, method)
- **Intern versus extern:** a *Bad Smell* can be determined either entirely internal the affected entity or only between multiple entities
- **Inheritance:** the *Bad Smell* relates to aspects of inheritance
- **Violation of Encapsulation:** the *Bad Smell* occurs because one entity accesses the internal parts of another entity

Bad Smells can manifest themselves also as lexical smells (Guerrouj et al., 2017) capturing recurring poor practices in the naming, documentation, and choice of identifiers during the implementation of an entity (e.g., misleading or incomprehensible terms).

Although researchers have developed several techniques to detect *Bad Smells*, which in fact found many instances of *Bad Smells* in real programs, their studies focused only on a limited number of types of *Bad Smells* (Moha et al., 2010; Hassaine et al., 2010; Khomh et al., 2009; Oliveto et al., 2010; Zhang et al., 2011). Consequently, a more comprehensive overview is needed on how often instances of all proposed categories occur in real programs.

Open Research Question 1. *What is the frequency distribution of bad-smell categories in real programs?*

If we knew this answer, we could better assess the relevance of specific *Bad Smells* and focus our research on detecting and handling the relevant *Bad Smells*. It would also enable a better root-cause analysis.

Evolution and effects of *Bad Smells*

The evolution of *Bad Smells*, that is, the question when they start and cease to exist as well as what happens to them during their lifetime, was only rarely and exemplarily investigated. For instance, Vaucher et al. (2009) studied *God Classes* and found out that many times it is difficult to avoid them, and often they are hardly changed in their lifetime. In this study, only *God Classes* were investigated, which is typical for many studies that focus on only very few ”classical” *Bad Smells* that can be easily detected. For instance, Olbrich et al. (2009) examined the following four hypotheses by looking at two *Bad Smells*, namely, *God Class* and *Shotgun Surgery*, in two open-source systems:

H1: The total number of *Bad Smells* increases continuously.

H2: The relative number of classes affected by a *Bad Smell* increases over time.

H3: The change frequency of classes affected by *Bad Smells* is higher than for classes without *Bad Smells*.

H4: The change volume of classes affected by *Bad Smells* is higher than for classes without *Bad Smells*.

The first two hypotheses were falsified in their study. Yet, there was a statistically significant correlation between the size of a system and the number of observed *Bad Smells*, where larger

systems tend to have disproportionately more *Bad Smells*. Hypotheses H3 and H4 were confirmed. This study may suggest a negative impact of these two *Bad Smells* on maintainability. Yet, only two of the 22 *Bad Smells* by Fowler (1999) and only two open-source systems were investigated. This study should be replicated for other systems and extended to other types of *Bad Smells*.

Lozano et al. (2007) proposed to study the evolution of *Bad Smells* to assess any negative impact because their influence may manifest only after some time. As an example, they mention software clones for which the study of their evolution yielded many new findings. In fact, recent work on software clones is showing that considering history is important and leads to better results. Consequently, researchers should use historic data for all the other types of *Bad Smells*, too. Among the earliest examples of using historic data for investigating clones is the study by Kim et al. (2005), who analyzed the genealogy of clones. A clone genealogy shows how clones derive from each other over multiple versions of a program. Their study of two smaller systems found that many clones lived only for a short time. About two third were removed soon after they were created. Yet, they looked at two rather small programs in their early development phase. A similar study by us for larger and longer-lived systems showed the opposite (Göde, 2009). In a follow-up study (Bazrafshan, 2012), we investigated the evolution of clones with modifications and came to the conclusion that clones with modifications require more attention in clone management compared to identical clones. Studies by Krinke (2008) and us (Göde and Harder, 2011) showed that clones are more stable (i.e., are less frequently modified) than non-cloned code, which may either indicate that they do not need to evolve so they are no maintenance problem or may indicate that developers do not dare to change them. Also, a study of the evolution of clones in long-lived systems showed that 90 % of the clones found never change or change only once in an investigated period of three years Göde and Koschke (2011). Thus, using historical information about the changes may help to identify those instances of *Bad Smells* that are more frequently changed and require more attention.

Lozano et al. investigated their hypothesis so far only for software clones and so did the other studies about clones. Whether these observations on clones also hold for other types of *Bad Smells* is not yet investigated. In addition to that, Lozano et al. conjecture that not a single *Bad Smell* alone causes maintenance problems but their combination with other *Bad Smells*. This conjecture was confirmed by a controlled experiment by Abbes et al. (2011) for a single version of a program (see below). For this reason, researchers need to investigate whether interactions of *Bad Smells* influence the evolution of a program, too.

Ratzinger et al. (2005) identify so-called *Change Smells* in addition to *Bad Smells*, that is, anomalies in the evolution. While classical *Bad Smells* are described in terms of structural deficiencies, *Change Smells* hint at problems specifically in the evolution. Ratzinger et al. investigated so called *Co-Changes*. These are code entities that are changed by the same commits. Their study showed that co-changed classes are strongly coupled and difficult to change. In the observation of a one-year period after a major refactoring of such classes to reduce the coupling, they found that the change effort was reduced. This study examined a new kind of *Bad Smell* that goes beyond the classical list of *Bad Smells* by Fowler (1999).

A general belief about *Bad Smells* is that they come into existence as a side effect of later modifications during the evolution of software. Contrary to this belief, Tufano et al. (2015) have found in their analysis of more than 200 open-source projects that slightly more than half of the *Bad Smells* in source-code files existed when files were checked into the version control system for the very first time. That is, most *Bad Smells* are a product of initial development rather than later modifications.

Researchers have studied the evolution and effects of *Bad Smells* and shown there are issues, but their studies are sometimes small, focus on only Java as a programming language, and look only very rarely into closed-source software. Moreover, they do not go as deep given the importance

of the problem, that is, they observe the evolution of *Bad Smells* mostly quantitatively by summarizing statistics. They rarely look into the reasons that cause these observations. Studying the evolution of *Bad Smells* deeply helps to better investigate the root causes and effects of *Bad Smells*. Studying the evolution of *Bad Smells* is a means to answer important research questions.

Open Research Question 2. *What are the effects and evolution of (other types of)* Bad Smells *and combinations thereof? In particular, do the current results also hold for large and closed-source software written in languages other than Java? How long do different types of* Bad Smells *live and how often are they changed?*

Origin of *Bad Smells*

Bad Smells may occur due to diverse reasons such as time pressure, ignorance, or insufficient training—maybe even as a conscious act of job securing. They may also occur because of awkward interactions such as mixing different design principles or an inappropriate application of a design pattern. At present, little is known about the real reasons behind *Bad Smells*.

Knowing the root causes would offer us the opportunity to establish sustainable countermeasures. Knowing the relative frequency of the root causes would help us to prioritize these countermeasures. Even programming language design may take advantage of this knowledge.

Open Research Question 3. *What are the reasons for* Bad Smells*? In particular, how often do these reasons promote the genesis of* Bad Smells*?*

Detecting *Bad Smells*

There are different approaches to detect *Bad Smells*. Some of them use search patterns for known *Bad Smells* (Moha et al., 2010), while others use software metrics indicating potential anomalies (Munro, 2005; Lanza and Marinescu, 2006). Pattern recognition can be further distinguished according to the program representation upon which the search is based. Simple analyses are based on text. Alternatively, a lexical analysis may be used to transform source text into a token stream. Based on the token stream, syntax analysis may construct a syntax tree. A syntax tree can be further enriched by semantic analysis as well as control and data flow analyses yielding a program dependency graph. For all these program representations, there exist different clone detection analyses, for instance (Koschke, 2007, 2008; Roy et al., 2009). Other techniques use a structural global dependency graph, which may be directly derived from Java byte code, for instance, showing the classes and their structural dependencies (Abbes et al., 2011).

The idea of a metric-based search is to measure code aspects indicating *Bad Smells* (Marinescu, 2002; Lanza and Marinescu, 2006). An example of this type of detection is the method by Ratiu et al. (2004) to detect *Data Classes*. A *Data Class* is a class that has only little functionality and is used mostly to store data for other classes. To search for these, one can measure the size and complexity of methods and count the number of public attributes. Another approach by Simon et al. (2001) measures cohesion of different entities based on the similarity of their common attributes. Munro (2005) describes different characteristics and design heuristics indicating *Temporary Field* and *Lazy Class*.

Walter and Pietrzak (2005) state that the use of metric-based approaches is limited because the aggregated numbers give only a static view on the code. For this reason, they propose to use multiple criteria from six different sources to detect *Bad Smells*. In addition to metrics, they use information from the following sources: syntax tree, change history, and dynamic information from runtime analysis. They also try to integrate the developers' experience and intuition in

the assessment. By way of the *Utilité-Additive* method (Jacquet-Lagréze and Siskos, 1982), they group the found *Bad Smells* according different criteria. The intuition of the developer is modeled as a subjective rating of the importance of a *Bad Smells*. This rating may come from real user data or use a predefined ranking criterion.

Bryton et al. (2010) also introduced an approach that uses complexity metrics combined with human experience. They investigated the *Bad Smell Long Method* using a *Binary Regression* model calibrated by the knowledge provided by an expert – that is, a person with sufficient experience in detecting, classifying, and removing *Bad Smells*– to enable the automation for detecting instances of the *Long Method* smell. In their study, the authors filled the role of the expert themselves. Based on a training set, the model is used to overcome subjectivity and to automate the process of code smell detection. In a case study, the model was evaluated and it was found that the subjectivity in the detection of the *Bad Smell Long Method* could be reduced. However, they used only a relatively small number of complexity metrics and a rather small training set in the calibration phase.

Another approach that uses dynamic analysis was proposed by Kataoka et al. (2001). They identify *Bad Smell* candidates for four refactorings by searching for particular invariants in the program. For instance, the refactoring *Remove Parameter* attempts to remove redundant parameters whose values can be computed from the other given parameters. The dynamic analysis by Kataoka et al. (2001) identifies candidates for this refactoring by checking at runtime whether the states of the parameters observed at runtime indicate that one parameter may be derived from the others.

Bad Smells are often described informally in the literature. As a consequence, they are interpreted differently depending upon the perspective of the person searching for them. In a study by Mäntylä and Lassenius (2006) on the manual detection of *Bad Smells* by developers, they found that developers suffer losses in their ability to detect *Bad Smells* objectively with increasing familiarity. In other words, known *Bad Smells* became normal for them. This study was extended by Schumacher et al. (2010) confirming this observation, where developers of two software systems of one company were asked to assess instances of the *Bad Smell God Class*.

Fontana et al. (2012) presented a comparison of four different *Bad Smell* detection tools. In a case study, six versions of a medium-size software project were analyzed to investigate whether automatically detected *Bad Smells* are useful to assess which parts of the software code need to be improved or at least kept track of. Although the authors conclude that the results produced by the detectors are relevant to minimize manual effort to uncover *Bad Smells*, they also found that different detectors for the same *Bad Smell* do often not meaningfully agree in their results.

The lack of formal criteria for *Bad Smells* makes it difficult to automate the detection of *Bad Smells* completely. Because of that, most approaches are only semi-automated, yielding a set of candidates that must be assessed by developers.

In a web-based survey Mäntylä (2003) investigated whether software developer have a uniform opinion on *Bad Smells* in source code. 12 developer of the same company participated in the survey and answered questions regarding *Bad Smells* in the source code of a software system that is internally developed in their company. It was found that the developers' opinions were not uniform. One reason for the disagreement was probably the different experience that the developers had with the software system. Based on the results of the survey, the author concludes that tools for the automatic detection and management of *Bad Smells* are useful to provide information on which parts of the source code are affected by *Bad Smells* as well as on the general quality of a software system.

There are also *Bad Smells* for which no automated detection exists because these *Bad Smells* were not yet described in the literature or because their detection represents an undecidable problem, such as *Insufficient Documentation* or *Badly Chosen Identifiers*. Consequently, automated analysis can hardly replace manual inspection by developers who are generally in a better position

to assess a piece of code based on their experiences and holistic comprehension of programs and the application domain.

The approach by Walter and Pietrzak (2005) is the only approach we know of that uses historical code change information to detect *Bad Smells*. We believe this type of information has great potential. Some *Bad Smells* such as *Divergent Changes* can only be found using evolutionary data. Furthermore, the version history may help to assess the relevance of a *Bad Smell* (Ratiu et al., 2004). For instance, if the version history of a *God Class* shows the tendency to shrink, there is less need to handle it compared to if it were to show a growing trend. In addition to that, version history helps to determine the point in time where the *Bad Smell* was introduced. If it was created a long time ago but has never changed, it may be less problematic – or may be even more problematic because no developer dares to touch it. Thus the evolution of *Bad Smells* can help to find them more reliably and to better prioritize them. In practice, automated *Bad Smell* detectors may find a very large set of *Bad Smells* in a large software system. This set must be ranked and ordered according to relevance and severity.

There are different directions along specific challenges in which bad-smell detection may be improved. First, there are technical issues in order to detect and trace *Bad Smells* more effectively and efficiently. Second, there are definition issues as a prerequisite for bad-smell detection, and third, there is a relevance or value issue, that is, we need a suitable task-dependent ranking of *Bad Smells*. We believe that all these three issues may benefit from considering evolutionary data on *Bad Smells*.

Open Research Question 4. *How can we take advantage of evolutionary data to better define, detect, trace, and rank* Bad Smells*?*

Removal of *Bad Smells* through refactoring

The book by Fowler (1999) describes various refactorings to remove *Bad Smells*. Refactorings are semantic preserving structural changes to improve maintainability.

Many integrated development environments (IDE) such as *Eclipse* offer automated refactorings. According to Xing and Stroulia (2006) they are often used (although not all equally often). Whether they are used to remove *Bad Smells* or for other purposes, however, was not investigated. According to Mens and Tourwé (2004), *Bad Smells* are good indicators to identify code worth to be refactored. Which *Bad Smells* are removed in practice and which effort is needed, was investigated empirically in an initial study by Counsell et al. (2010) in five open-source systems. They found only two *Bad Smells* that are difficult to remove with existing refactorings, namely, *Alternative Classes With Different Interfaces* and *Parallel Inheritance Hierarchies*. While other *Bad Smells* were remedied in all systems, these two were only partially removed by any of Fowler's refactorings. This study is still preliminary because it looked at a limited number of versions and systems, all of the latter being open source.

Open Research Question 5. *Which* Bad Smells *are removed in practice (both in open-source and industrial systems) and how is that done?*

Which (research) tools are used in practice for detecting and presenting *Bad Smells* and what deficiencies they have, is not well researched. There are studies, such as the one by Parnin et al. (2008) evaluating detection and visualization techniques for *Bad Smells* for industrial programs; yet, usually these tools are not used by the industrial developers themselves in the long term but only by the investigating researchers during the period of evaluation.

Open Research Question 6. *Which tools are used to detect, visualize, and remove* Bad Smells *in practice? How well suited are they? What information needs do developers have to detect and treat* Bad Smells*?*

If we had answers to these questions, we would know under which circumstances and how developers remove *Bad Smells*. This knowledge may help to prioritize refactoring activities and to improve existing refactoring tools.

Effects of *Bad Smells*

Relatively few studies have investigated the effects of *Bad Smells* on maintainability empirically, that is, through controlled experiments or case studies.

A controlled experiment by Deligiannis et al. (2003, 2004) aimed at investigating the impact of *God Class* on maintainability. The results of this study indicate that *God Class* affects the evolution of design structures negatively and considerably affects the way participants apply the inheritance mechanism.

An experiment by Bois et al. (2006) aimed at verifying that *God Class* decomposition can facilitate comprehension. As one of the indicators of comprehensibility, the authors used the ability to map attributes in the class hierarchy to domain concepts. This ability was significantly affected by the decomposition and its interaction with the institution from which the participant was enrolled.

Abbes et al. (2011) were the first to investigate the interaction of *Bad Smells*. In a controlled experiment examining the two *Bad Smells God Class* and *Spaghetti Code*, the participants' performance (time, effort, and correctness in a maintenance task) was not affected by the presence of a single *Bad Smell*. Only when both *Bad Smells* were present, the decrease of performance was statistically significant.

We assessed the effect of software clones on corrective maintenance tasks by way of a controlled experiment, too (Harder and Tiarks, 2012). Although we could not found a statistically significant correlation between the presence of clones and the time needed to fix an error therein, we did observe cases in which developers overlooked that they should have corrected a defect once more in a copy of the erroneous code fragment.

Most case studies studying the effect of *Bad Smells* focus on *Duplicated Code*. One surprising result was found by Monden et al. (2002), one of the first studies on the effect of software clones. Their study indicates that the relation of *Bad Smells* and maintainability is a non-obvious one. They found that modules in an old and large COBOL system with a maximal clone of at least 200 lines had the highest defect density, but modules with maximal clones in between 100 and 200 lines had a lower defect density than modules whose maximal clones were shorter than 100 lines.

There have been other case studies on software clones indicating that software clones may lead to defects (Chou et al., 2001; Jürgens et al., 2009; Li et al., 2006; Göde and Koschke, 2011), in particular if the changes made to them are inconsistent. On the other hand, a study by Lozano et al. (2007) did not reveal any strong influence of software clones on change frequency and impact. A study by Krinke (2008) indicated that clones are actually more stable than non-cloned code. His study was later replicated, extended, and confirmed by studies of us (Göde and Harder, 2011; Harder and Göde, 2013).

Additional studies by our research group (Göde, 2010; Bazrafshan and Koschke, 2013) showed that software clones are rarely removed actively despite their potential bad influence. If they are removed, then mostly as a side effect of other major restructurings.

Only few case studies consider other types of *Bad Smells*. One such exception is a study by Li and Shatnawi (2007) on the relation of defects and the *Bad Smells God Class*, *Data Class*,

God Method, *Refused Bequest*, and *Shotgun Surgery*. Their results for three released versions of *Eclipse* indicate that some of these *Bad Smells* do have an influence on defects. Their study also showed that these defects could be avoided if one detects *Bad Smells* during development, but not after the software was released.

There is a strong emphasis in research on the effects of *God Class* and *Duplicated Code* on maintainability. Other types of *Bad Smells* are less investigated.

Open Research Question 7. *How and when do* Bad Smells *in general affect maintainability?*

If *Bad Smells* do not affect maintainability, there is no need for further research in detecting and removing them. We expect that different types of *Bad Smells* influence maintainability in different ways. Thus, empirical research along these lines may help to prioritize types of *Bad Smells* for both research and practice.

Assessing software quality based on *Bad Smells*

Even though the relation of *Bad Smells* and internal and external software quality attributes is not sufficiently understood, *Bad Smells* are used in software quality assessments.

The *Software Quality Index* by Simon et al. (2006), for instance, measures the occurrence of several *Bad Smells* and compares this frequency distribution to a benchmark of industrial and open-source systems. Using this approach, one can make a relative comparison of a given system to this benchmark.

This idea was later revived by the German TÜVit[1], which offers a *Trusted Product Maintainability* certificate based on the ISO/IEC 9126 standard for software quality in cooperation with the company *Software Improvement Group*. They measure various product metrics (volume, duplication, unit size, unit complexity, unit interfacing, and coupling) hinting at *Bad Smells*. Analogous to the *Software Quality Index*, a given system is rated relatively to other systems with respect to these metrics.

Relative comparisons are problematic because they depend upon the systems contained in the benchmark. The rated quality of a software may be 'improved' simply by adding worse systems to the benchmark. Furthermore, the significance and completeness of the considered aspects are not sufficiently validated; see our above summary of existing studies on *Effects* of *Bad Smells*. A better understanding of the effects of *Bad Smells*, that is, an answer to research question 7, would provide a better empirical foundation of such approaches to assess maintainability.

Broy et al. (2006) developed a maintainability model with industrial partners distinguishing in activities for maintaining internal system properties. The more abstract activities of the IEEE standard 1219 *analysis, implementation, test*, and *deployment* are further put in more concrete terms (location, change impact analysis, coding, integration test, etc.). These activities are related to structural software properties such as clones, formatting, identifier naming, etc. Some of these can be measured automatically, some require manual reviews. Broy et al. (2006) expect a negative influence of *Bad Smells* on particular maintenance activities, although they have not yet empirically investigated this conjecture. Another related research project along these lines—but much broader—is *Quamoco*[2] aiming at improving the practicality of general quality standards such as ISO/IEC 9126 und ISO 25000 (Wagner et al., 2010). Although, maintainability is also one quality aspect considered in this project, the focus is on more general external aspects of software quality such as performance, dependability, etc.

[1] `http://www.tuvit.de/`
[2] `https://quamoco.in.tum.de/`

Open Research Question 8. *How can we provide scientific empirical evidence to quality models based on* Bad Smells*?*

We believe that maintainability can be better assessed if not only a single version of a system is analyzed but the version history. For instance, if a *God Class* exists for a very long time and was never changed, it is likely less problematic than a younger and growing *God Class*. This leads us to the following open research question.

Open Research Question 9. *How can quality models based on* Bad Smells *be improved by considering the software evolution?*

Summary and Advice

By and large, software developers do care about *Bad Smells*. This brief summary of the existing body of knowledge in the scientific literature shows that their concerns are backed up by research results. There is a large number of empirical studies indicating that *Bad Smells* indeed may lead to defects and have a negative influence on the effort to understand, test, and modify a program. More studies are nevertheless required to strengthen this evidence, in particular for languages other than Java and especially for industrial closed-source software. The latter can only happen when practitioners and researchers work more closely together. Not all types of *Bad Smells* are equally well investigated and, hence, more research is needed for the less understood types of *Bad Smells*. The relative weight of *Bad Smells* in terms of impact on maintainability must be explored further in order to equip developers with a relevance ranking of the detected *Bad Smells* because their number can be huge. This relevance ranking should also take into account the specifics of the primary task developers are trying to achieve. Generally, their primary task is not to clean up the code. Rather their primary task is to fix a bug or to add a feature, and *Bad Smells* simply stand in their ways. The relevance ranking should also take into account the planned changes in the future. If code is deemed to be removed soon, putting additional efforts into removing its *Bad Smells* can hardy be justified economically. Speaking about economics, ideally we should have a better understanding of the maintenance costs of *Bad Smells*—the interests paid due to technical debt—and also the costs and risks to remove them so that we can make more informed decisions. Another interesting questions is when to present *Bad Smells* to developers. Should it be the time when a developer introduces a *Bad Smells* while typing in the IDE (which may distract a developer from her or his actual task), when the code is checked into the version control system, during code review, or in a development phase specifically dedicated to clean up the code before releasing it?

The enumeration of open research questions has shown that there are still many open questions to be answered until we have a clear understanding of these issues. What can practitioners do until we have more certainty? While researchers ultimatively strive for generalizability, practitioners are quite happy with facts that just hold in their own organization. To gather more knowledge actionable in their daily job, practitioners should first assess their code objectively to get a clearer picture of which kinds and to what extent *Bad Smells* actually occur in their code. These existing *Bad Smells* can be used as a kind of baseline. This baseline helps to form the goal to get gradually better or at least not to get worse. Integrated into continuous quality processes, developers should re-assess their code on a regular basis. These cycles should be short (e.g., weekly) rather than long (e.g., every major release). A small team of leading developers should be formed that takes on the task to look at the trends of *Bad Smells* periodically and to make decisions on how to act on them. It goes without saying that the search for *Bad Smells* must be automated and integrated into the continuous integration process as much as testing and this automated process must determine which *Bad Smells* have existed before and which were added, removed, or other-

wise modified by the latest commits. This delta analysis helps to make a distinction between old and new *Bad Smells*. The history or evolution of *Bad Smells* is a valuable source of information in assessing them and in making informed decisions. More recent *Bad Smells* can be more easily removed generally. The harmfulness of bad smells can be better assessed if one has knowledge about their age and the frequency of changes of parts of the program affected by *Bad Smells*. If there is little intial knowledge on the relevance of *Bad Smells* based on empirical data, developers should focus on the *Bad Smells* that already have caused problems in the organization during development or are known to cause problems according to the scientific literature. If eventually the detection and tracking of *Bad Smells* integrated into the continuous integration process has gathered sufficient data, developers can analyze these data in the context of other maintainability aspects in their organization such as number of defects, test effort, change frequency and effort, and the time it takes to understand a piece of software. Associating *Bad Smells* with these maintainability indicators and looking for recurring patterns through data mining will help them to better understand cause-effect chains and to plan and steer appropriate counter-measures. This is a continuous learning process driven by empirical data rather than just gut feeling and general (mis-)beliefs. The reward for this effort is software that can be economically maintained for the time needed and that allows to add features quickly. Highly maintainable software saves development costs. Yet, arguably even more important in highly dynamic markets, high changeability may be the major decisive factor to compete through inventions in short cycles.

About the Author

Dr. Rainer Koschke is a full professor for software engineering at the University of Bremen in Germany and is heading the software engineering group. He received a doctoral degree in computer science at the University of Stuttgart, Germany. His research interests are primarily in the fields of software engineering and program analyses. His current research includes architecture recovery, feature location, program analyses, clone detection, bad smells, reverse engineering, and security. He is one of the founders of the Bauhaus research project (founded in 1997) and its spin-off Axivion GmbH[3] (founded in 2006) to develop methods and tools to support software maintainers in their daily job through reconstructed architectural and source code views. He is teaching reengineering and software engineering.

Contact

Internet: `http://www.informatik.uni-bremen.de/~koschke/`
E-Mail: `koschke@uni-bremen.de`

[3] `http://www.axivion.com`

References

Abbes, M., F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *European Conference on Software Maintenance and Reengineering*, pp. 181–190. IEEE Computer Society Press.

Bazrafshan, S. (2012). Evolution of near-miss clones. In *Workshop Source Code Analysis and Manipulation*, pp. 74–83. IEEE Computer Society Press.

Bazrafshan, S. and R. Koschke (2013). An empirical study of clone removals. In *International Conference on Software Maintenance*, pp. 50–59. IEEE Computer Society Press.

Bois, B. D., S. Demeyer, J. Verelst, T. Mens, and M. Temmerman (2006). Does god class decomposition affect comprehensibility? In *IASTED International Conference on Software Engineering*, pp. 346–355. IASTED/ACTA Press.

Brown, W. J., R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray (1998, 4). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1 ed.). Wiley.

Broy, M., F. Deissenboeck, and M. Pizka (2006). Demystifying maintainability. In *WoSQ '06: Proceedings of the 2006 International Workshop on Software Quality*, New York, NY, USA, pp. 21–26. ACM.

Bryton, S., F. Brito e Abreu, and M. Monteiro (2010). Reducing subjectivity in code smells detection: Experimenting with the long method. In *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, pp. 337–342. IEEE Computer Society.

Chou, A., J. Yang, B. Chelf, S. Hallem, and D. R. Engler (2001). An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pp. 73–88.

Counsell, S., R. M. Hierons, H. Hamza, S. Black, and M. Durrand (2010). Is a strategy for code smell assessment long overdue? In *WETSoM '10: Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, New York, NY, USA, pp. 32–38. ACM.

Deligiannis, I., M. Shepperd, M. Roumeliotis, and I. Stamelos (2004). An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software 72*(2), 129–143.

Deligiannis, I., I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd (2003, February). Controlled experiment investigation of an object oriented design heuristic for maintainability. *Journal of Systems and Software 65*(2), 127–139.

Fontana, F. A., P. Braione, and M. Zanoni (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology 11*(2), 5:1–38.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Garcia, J., D. Popescu, G. Edwards, and N. Medvidovic (2009a). Identifying architectural bad smells. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, Washington, DC, USA, pp. 255–258. IEEE Computer Society.

Garcia, J., D. Popescu, G. Edwards, and N. Medvidovic (2009b). Toward a catalogue of architectural bad smells. In *QoSA '09: Proceedings of the 5th International Conference on the Quality of Software Architectures*, Berlin, Heidelberg, pp. 146–162. Springer-Verlag.

Göde, N. (2010). Clone removal: Fact or fiction? In *International Workshop on Software Clones*, pp. 33–40. ACM Press.

Göde, N. and J. Harder (2011). Clone stability. In *European Conference on Software Maintenance and Reengineering*, pp. 64–74. IEEE Computer Society Press.

Göde, N. and R. Koschke (2011). Frequency and risks of changes to clones. In *International Conference on Software Engineering*, pp. 311–320. ACM Press.

Guerrouj, L., Z. Kermansaravi, V. Arnaoudova, B. C. Fung, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc (2017, September). Investigating the relation between lexical smells and change- and fault-proneness: An empirical study. *Software Quality Journal 25*(3), 641–670.

Göde, N. (2009). Evolution of type-1 clones. In *Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation*, pp. 77–86. IEEE Computer Society Press.

Harder, J. and N. Göde (2013). Cloned code: stable code. *Journal on Software: Evolution and Process 25*(10), 1063–1088.

Harder, J. and R. Tiarks (2012). A controlled experiment on software clones. In *IEEE International Conference on Program Comprehension*, pp. 219–228.

Hassaine, S., F. Khomh, Y.-G. Guéhéneuc, and S. Hamel (2010). Ids: An immune-inspired approach for the detection of software design smells.

In *Quality of Information and Communications Technology*.

Jacquet-Lagréze, E. and J. Siskos (1982). Assessing a set of additive utility functions for multicriteria decision-making, the UTA method. *European Journal of Operational Research 10*(2), 151 – 164.

Jürgens, E., F. Deissenboeck, B. Hummel, and S. Wagner (2009). Do code clones matter? In *International Conference on Software Engineering*. ACM Press.

Kataoka, Y., M. D. Ernst, W. G. Griswold, and D. Notkin (2001). Automated support for program refactoring using invariants. In *International Conference on Software Maintenance*, pp. 736–743.

Khomh, F., S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui (2009). A bayesian approach for the detection of code and design smells. In *International Conference on Quality Software*. IEEE Computer Society Press.

Kim, M., V. Sazawal, D. Notkin, and G. C. Murphy (2005). An empirical study of code clone genealogies. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE*.

Koschke, R. (2007). Survey of research on software clones. In R. Koschke, E. Merlo, and A. Walenstein (Eds.), *Duplication, Redundancy, and Similarity in Software*, Number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

Koschke, R. (2008). *Software Evolution*, Chapter Identifying and Removing Software Clones, pp. 15–36. Springer Verlag. Herausgeber: Serge Demeyer und Tom Mens.

Krinke, J. (2008). Is cloned code more stable than non-cloned code? In *Workshop Source Code Analysis and Manipulation*, pp. 57–66. IEEE Computer Society Press.

Lanza, M. and R. Marinescu (2006, August). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Berlin: Springer.

Li, W. and R. Shatnawi (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software 80*(7), 1120 – 1128. Dynamic Resource Management in Distributed Real-Time Systems.

Li, Z., S. Lu, S. Myagmar, and Y. Zhou (2006, March). Copy-paste and related bugs in large-scale software code. *IEEE Computer Society Transactions on Software Engineering 32*(3), 176–192.

Lozano, A., M. Wermelinger, and B. Nuseibeh (2007). Assessing the impact of bad smells using historical information. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, New York, NY, USA, pp. 31–34. ACM.

Mäntylä, M. (2003). Bad smells in software – a taxonomy and an empirical study. Masters thesis, Helsinki University of Technology, Finland.

Mäntylä, M., J. Vanhanen, and C. Lassenius (2003). A taxonomy and an initial empirical study of bad smells in code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, pp. 381. IEEE Computer Society.

Mäntylä, M. V. and C. Lassenius (2006). Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering 11*(3), 395–431.

Marinescu, R. (2002). *Measurement and Quality in Object-Oriented Design*. PhD dissertation, Politehnica University of Timisoara.

Marticorena, R., C. Lopez, and Y. Crespo (2006). Extending a taxonomy of bad code smells with metrics. In *WOOR'06:Workshop on Object-Oriented Reengineering*.

Mens, T. and T. Tourwé (2004). A survey of software refactoring. *IEEE Computer Society Transactions on Software Engineering 30*(2), 126–139.

Moha, N., Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur (2010, Jan.-Feb.). Decor: A method for the specification and detection of code and design smells. *IEEE Computer Society Transactions on Software Engineering 36*(1), 20–36.

Monden, A., D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto (2002). Software quality analysis by code clones in industrial legacy software. In *Symposium on Software Metrics*, pp. 87–94.

Munro, M. (2005, Sept.). Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Symposium on Software Metrics*, pp. 15–15.

Olbrich, S., D. S. Cruzes, V. Basili, and N. Zazworka (2009). The evolution and impact of code smells: A case study of two open source systems. In *International Symposium on Empirical Software Engineering and Measurement*, pp. 390–400. IEEE Computer Society Press.

Oliveto, R., F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc (2010). Numerical signatures of antipatterns: An approach based on b-splines. In *European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press.

Palomba, F., G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia (2014, Sept). Do they really smell bad? a study on developers' perception of bad

code smells. In *International Conference on Software Maintenance and Evolution*, pp. 101–110.

Parnin, C., C. Görg, and O. Nnadi (2008). A catalogue of lightweight visualizations to support code smell inspection. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, New York, NY, USA, pp. 77–86. ACM.

Ratiu, D., S. Ducasse, T. Gîrba, and R. Marinescu (2004). Using history information to improve design flaws detection. In *European Conference on Software Maintenance and Reengineering*, pp. 223–232. IEEE Computer Society Press.

Ratzinger, J., M. Fischer, and H. Gall (2005). Improving evolvability through refactoring. In *Working Conference on Mining Software Repositories*, pp. 1–5.

Roy, C. K., J. R. Cordy, and R. Koschke (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Journal Science of Computer Programming 74*(7), 470–495.

Schumacher, J., N. Zazworka, F. Shull, C. Seaman, and M. Shaw (2010). Building empirical support for automated code smell detection. In *International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, pp. 1–10. ACM.

Simon, F., O. Seng, and T. Mohnhaupt (2006). *Code-Quality-Management – Technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht*. dpunkt.verlag.

Simon, F., F. Steinbrückner, and C. Lewerentz (2001). Metrics based refactoring. In *European Conference on Software Maintenance and Reengineering*, Washington, DC, USA, pp. 30. IEEE Computer Society.

Srivisut, K. and P. Muenchaisri (2007). Defining and detecting bad smells of aspect-oriented software. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, Washington, DC, USA, pp. 65–70. IEEE Computer Society.

Tahir, A., A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell (2018). Can you tell me if it smells?: A study on how developers discuss code smells and anti-patterns in stack overflow. In *International Conference on Evaluation and Assessment in Software Engineering*, pp. 68–78. ACM.

Tufano, M., F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk (2015). When and why your code starts to smell bad. In *International Conference on Software Engineering*, pp. 403–414. ACM Press.

Vaucher, S., F. Khomh, N. Moha, and Y.-G. Guéhéneuc (2009). Tracking design smells: Lessons from a study of god classes. In *Working Conference on Reverse Engineering*, pp. 145–154. IEEE Computer Society Press.

Wagner, S., K. Lochmann, S. Winter, A. Goeb, M. Klaes, and S. Nunnenmacher (2010). Software quality models in practice - survey results. `https://quamoco.in.tum.de/wordpress/wp-content/uploads/2010/01/Software_Quality_Models_in_Practice.pdf`.

Walter, B. and B. Pietrzak (2005). Multi-criteria detection of bad smells in code with UTA method. In *XP*, pp. 154–161.

Webster, B. F. (1995). *Pitfalls of Object Oriented Development*. M & T Books.

Xing, Z. and E. Stroulia (2006). Refactoring practice: How it is and how it should be supported – an eclipse case study. In *International Conference on Software Maintenance*, pp. 458–468. IEEE Computer Society Press.

Yamashita, A. and L. Moonen (2013). Do developers care about code smells? an exploratory survey. In *Working Conference on Reverse Engineering*, pp. 242–251. IEEE Computer Society Press.

Zhang, M., T. Hall, and N. Baddoo (2011). Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice 23*(3), 179–202.