Aligning the Normalized Systems Theorems with Existing Heuristic Software Engineering Knowledge

Peter De Bruyn, Geert Dierckx, and Herwig Mannaert
Department of Management Information Systems
Normalized Systems Institute (NSI)
University of Antwerp
Antwerp, Belgium
{*peter.debruyn,herwig.mannaert*}*@ua.ac.be,* {*geert.dierckx*}*@student.ua.ac.be*

***Abstract*—Software applications used by contemporary organizations have to be expendable for incorporating additional functional requirements, as well as adaptable regarding ever changing user requirements. As this evolvability has frequently be noted to be lacking in most information systems, Normalized Systems (NS) theory has recently proposed a framework on evolvable modularity. Based on the concept of systems theoretic stability, NS formulates a number theorems, constituting a set of formally proven necessary conditions in order to obtain such evolvability in modular structures. These theorems were argued to strongly align with heuristic (often tacit) best-practice knowledge of experienced software developers. In order to further validate this claim, the present paper will investigate the degree in which the NS theorems align with best-practice software engineering guidelines based on the set of 22 "bad smells in code" as defined by Fowler and Beck. The analysis shows that the avoidance of the code smells indeed largely aligns with the Normalized Systems theorems. While 14 of the guidelines seem to be reflected by the NS theorems, 4 of them seem to be unrelated to the theorems and another set of 4 code smells seems to be contradicting with NS reasoning.**

***Keywords-Normalized Systems*; *Code Smells*; *Heuristic Knowledge*; *Knowledge Management*.**

I. INTRODUCTION

Software applications used by contemporary organizations have to be expendable for incorporating additional functional requirements, as well as adaptable regarding ever changing user requirements. While many best-practice principles exist in order to improve the evolvability of software programs, the knowledge management concerning these heuristics remains inadequate: most of the principles are often only known tacitly and are not applied consistently. In this regard, Normalized Systems (NS) theory has recently formulated a set of four (formally proven) theorems as necessary conditions to obtain evolvable modular structures in software systems [1]–[3]. While these theorems as such are not to be considered entirely new, their value should be seen in their unambiguous formulation and proof, as well as their unification based on a single postulate. In the present paper, the main focus will be aimed at the best-practice knowledge residing into these Normalized Systems theorems. Indeed, it has already been argued that the Normalized Systems theorems offer in fact a more specific and unambiguous way of representing some already existing (tacit) best-practices in the software engineering community [1]–[3]. Hence, we will try to further support this argument by analyzing how the Normalized Systems theorems seem to be largely supported by the guidelines of Fowler et al. [4] based on the prevention of bad code smells.

After briefly highlighting the essence of Normalized Systems theory, Section III will situate the bad code smells in software engineering literature and the relevance of comparing it with the NS theory. A mapping of both approaches will be proposed in Section IV, after which some conclusions will be presented in Section V.

II. NORMALIZED SYSTEMS

Normalized Systems (NS) is a theory focusing on the evolvability of software architectures, based on the concept of stability from systems theory. For this purpose, it considers software systems as *modular* systems consisting of a set of instantiations of programming constructs (e.g., methods, data structures, etc.). In order to realize proven evolvability in these systems, first, an *unlimited systems evolution* is considered (meaning that in theory the number of instantiated constructs and their mutual dependencies eventually become unlimited in every software system). Next, the *stability* requirement demands that each bounded set of changes to the software system (e.g., adding a new data construct or adding a new version of an action construct) should have a bounded impact as well (i.e., the impact of a change should only be depended on the kind of change performed to the system and not dependent on the size of the system). Changes which do generate an impact dependent on the size of the system are regarded as instable (as their impact becomes unbounded under the unlimited systems evolution assumption) and are called *combinatorial effects*. In order to enable this stability, NS theory proposed the following four (formally proven) *theorems* as necessary conditions to prevent combinatorial effects [1]–[3]:

- *Separation of Concerns*, requiring that every concern (change driver) is separated from other concerns in its

 ISBN: 978-1-61208-230-1

own construct;
- *Data Version Transparency*, requiring that data entities can be updated without impacting the entities using it as an input or producing it as an output;
- *Action Version Transparency*, requiring that an action entity can be updated without impacting its calling components;
- *Separation of States*, requiring that each step in a workflow is separated from the others in time by keeping state after every step.

In reality, building software applications in conformance with all four theorems seems to require a set of five encapsulated higher-level programming constructs (called *elements*) as building blocks for NS conform applications: data elements, action elements, workflow elements, connector elements and trigger elements [1], [3].

In [2, p. 94], it was already noted that the NS "*design theorems are not new, but relate to well-known heuristic design knowledge of software developers*". For instance, a limited set of manifestations of the above explained theorems were already mentioned in [1]–[3] as summarized in Table I. Therefore, it might be interesting to investigate the extent in which other best-practice heuristics in software engineering (e.g., the bad code smells as formulated by Fowler et al. [4]) conform with the NS theorems.

III. BAD SMELLS IN CODE

In the source code of a software program, *bad code smells* are typically considered as symptoms or indications regarding potentially troublesome or problematic code [4]. As such, the concept should be clearly distinguished from typical bugs: flaws or mistakes in the source code resulting in undesired effects (mostly at runtime), such as erroneous output values or security breaches. Consequently, code smells do not imply an erroneous execution of the software program at the time being. Rather, they point to parts in the code of which experience has shown that they have a high chance of causing real problems in the future, when the source code is adapted (e.g., due to highly complex code or its low evolvable structure). Based on this approach, Fowler et al. [4] presented a set of 22 bad smells in code, all being the expression of their experience-based heuristic programming knowledge build up over the years. In order to avoid the presence of these smells, the same authors propose a set of 72 refactorings further on in their book [4]. Here, *refactoring* is not to be considered as changing the delivered functionalities of a software program. Instead, the purpose is to redesign the structure of (a piece of) software code so that potentially troublesome parts (the "smells") are removed, while the program itself is still exhibiting the same behavior at runtime. A short overview of those 22 code smells from Fowler et al. [4] is presented in Table II by providing the name of each smell, a brief description of the potential problem anticipated to arise, as well as an explanation of the proposed remediation.

In general, the occurrence of code smells has become associated with the number and degree of difficulties programmers might be confronted with when trying to change existing code. The concept has become a generally known, accepted and established way for studying the maintainability of software. Indeed, after the publication of the book of Fowler et al. [4], researchers have been extending the repository of existing code smells or using them as a basis for empirical validation and software evaluation (see e.g., [5]–[8]).

Additionally, some limitations to the formulation of the code smells by Fowler et al. [4], can be derived from earlier related work as well, for instance based on Mäntylä et al. [5], [9]. In [9], the authors first argue that the 22 code smells as defined by Fowler et al. [4] are only presented in a single flat list without providing any classification. Hence, they might surpass the maximum number of guidelines which can be grasped and applied by a human being concurrently. Therefore, Mäntylä et al. [9] proposed a taxonomy of 6 bad smell categorizations, claiming to make the set of bad smells more understandable and clarifying the relationships between them. For example, the smells *Long Method*, *Large Class*, *Primitive Obsession*, *Long Parameter List* and *Data Clumps* were all characterized as "bloaters", representing situations in which a piece of code has grown so much that it becomes difficult to be handled effectively. Another taxonomy of the code smells can for instance be found in [10]. In some way, such taxonomies might already be seen as an early attempt to unify several of the code smells of Fowler et al. [4] while looking for some common causing grounds (i.e., what are the reasons for the smells to show up?). In this sense, the NS theorems (being formulated in a very widely applicable way) might show some analogy with these attempts as their aim was to identify and eliminate causes for barriers regarding evolvability (in terms of combinatorial effects) as well. Further, a second limitation suggested by Mäntylä et al. [5], [9] is the fact that the code smells are still somewhat ambiguous, implying that their most significant benefits are to be situated in the subjective evaluation of software evolvability (i.e., performed by individuals). This would limit their potential for a direct translation into software metrics allowing the full automatic detection of infringements by software tools. Indeed, while Fowler et al. [4, p. 63] claimed that their aim was to offer more specific refactoring clues than merely "*some vague ideas of programming aesthetics*", they specifically stated that they did not want to give very "*precise criteria for when a refactoring is overdue*" based on the argument that human intuition is intrinsically superior to a set of pure metrics. Also, both the studies of Mäntylä et al. [5] and Shneiderman [11] show that some disagreement between a set of such subjective software evaluations can arise (i.e., different peo-

 ISBN: 978-1-61208-230-1

Table I
SOME EXEMPLIFYING MANIFESTATIONS OF NORMALIZED SYSTEMS THEORY THEOREMS IN PRACTICE [1]–[3].

NS theorem	Exemplifying manifestations
Separation of Concerns	• Message or integration bus to separate the use of various messaging protocols • The separate use of external workflows by workflow management systems • Multi-tier architectures separating presentation logic, application or business logic, database logic, etcetera
Action version Transparency	• Polymorphism in object-orientation • Wrapper functions (e.g., in C) • Interface definition languages (IDL's) (e.g., used by frameworks such as CORBA and COM)
Data version Transparency	• XML-based technology (e.g., for web services) • Information hiding in object-orientation (e.g., getters and setters in the JavaBean component architecture) • Data structure passing via URL's, property files, tag-value pairs, etcetera
Separation of States	• Asynchronous communication systems • Asynchronous processing in general • Stateful workflow systems

ple might evaluate the same code differently). Therefore, the NS theorems might offer an interesting point of comparison in this situation as well, as their formulation was aimed at providing specific and formally proven guidelines for obtaining software evolvability and offering an unambiguous means of identifying violations against them. Hence, it might be interesting to analyze the extent in which we can map the bad code smells of Fowler et al. [4] on the NS theorems. Or, stated otherwise, to investigate the extent in which the bad code smells can be seen as manifestations or instantiations of (violations regarding) the NS theorems. This will be exactly our goal of the next section.

IV. ALIGNING NS THEOREMS WITH THE AVOIDANCE OF CODE SMELLS

In this section, our aim will be to examine the degree in which we can find conformance between the bad code smells of Fowler et al. [4] and the NS theorems. In order to do so, Table III tries to visualize and map each of the code smells with the NS theorems. The analysis reveals three kind of categories: (1) a set of 14 code smells in full, partial or indirect compliance or support of the NS theorems, (2) a set of 4 code smells not related to the NS theorems and (3) a set of 4 code smells contradicting with the NS theorems. We will now briefly discuss each of them.

A. Code smells in full, partial or indirect compliance or support of the NS theorems

Most of the code smells appear to be in full accordance with the Normalized Systems theorems. We will elaborate two examples here. First, the *Duplicate Code* smell straightforwardly follows from the Separation of Concerns theorem. Indeed, consider a situation in which a certain processing function A includes (amongst others) code chunk X, which is duplicated in another processing function B as well. This reveals that functions A and B contain at least two concerns (i.e., change drivers, tasks). In case X would then be changed (e.g., due to a new version or mandatory upgrade), both processing functions A and B should be adapted. Considering an unlimited systems evolution perspective, the eventual impact might become related to the overall system size and hence result in a combinatorial effect. Second, the *Long Parameter* smell is supported by, amongst others, the Action version Transparency theorem. Suppose that a processing function A has an interface w requiring a set of (primitive) input parameters $S_1, S_2, ..., S_a$ to perform its functionality. Suppose further that a set of L processing functions is calling A. A new version of A in order to incorporate some additional functionality might require additional primitive input parameter(s) and hence, the interface w might have to change (e.g., an additional input parameter S_b is added to the parameter list). In this case, all L processing functions should be adapted in order to keep calling A correctly (resulting in a combinatorial effect under to unlimited systems evolution assumption). The Action version Transparency theorem will prohibit the creation of such *Long Parameter* smell by requiring each processing function to exhibit version transparency. In practice, this is realized by avoiding the use of primitives (such as String, integer, etc.) in the interface of a processing function. Instead, objects as a whole are passed (i.e., encapsulated data structures as suggested by Fowler et al.). This would for example mean that instead of passing parameters amount (integer), beneficiary name (String), etcetera, the object Invoice will be passed to a processing function. Based on this reference to the Invoice object, the processing function can request all information it needs to perform its function. A new version of a processing function A (e.g., requiring information about the currency of the invoice) can now be implemented while keeping the same interface (i.e., the reference to the Invoice object) and not requiring any additional changes in the L calling processing functions.

Also, it is strikingly to note that the definition by Fowler et al. [4] concerning the two code smells which were mapped on all four NS theorems (i.e., *Divergent Change* and *Shotgun Surgery*) reflect the typical operationalization of evolvability in NS. For example, the definition of the *Divergent Change* smell almost fully corresponds with the notion of change drivers in NS as it demands for the identification of objects

Table II
AN OVERVIEW OF THE 22 CODE SMELLS AS DISCUSSED BY FOWLER ET AL. [4]

Code Smell	Summary
Duplicate Code	A code fragment occurring in two or more places in the code base. The code should be refactored in such way that all the duplicate fragments are grouped and become located at one place.
Long Method	A method containing a long sequence of code, often reflecting the incorporation of multiple functionalities. For ease of use and maintainability purposes, the code should be refactored in such way that each functionality becomes separated in its own method.
Large Class	A class containing too many (infrequently used) member variables and methods, often being an indication of duplicate code as well. The code should be refactored in such a way that duplicate code is eliminated and fixed clumbs of variables are separated in a distinct class.
Long Parameter	Methods having a long list of needed parameters for calling them, result in complexity and maintenance issues. The code should be refactored in such a way that instead of variables, objects are passed to the called methods. The needed information from those objects can than be requested by using for instance typical get-methods.
Divergent Change	The phenomenon that a class becomes frequently changed in different ways for different reasons. The code should be refactored in such a way that everything that changes for a particular cause becomes separated in its own class.
Shotgun Surgery	The phenomenon that when a (small) kind of change is aimed for, many (little) adaptations have to be made to a lot of different classes. This causes additional effort to perform changes and creates risks regarding internal consistency. The code should be refactored in such a way that changes remain contained in a single class resulting in a one-to-one link between common changes and classes.
Feature Envy	The case in which the method of an object tends to use more frequently variables (data) from other classes, than its own variables. This can occur as objects in object-orientation are typically a combination of both data (variables) and actions (methods). The code should be refactored in such a way that the method is replaced to the class from which it is intensively using the data.
Data Clumbs	Groups of data (variables, parameters) often occurring together in different objects. This hampers adaptability and increases complexity of method callings. The code should be refactored in such a way that the bunches of recurring data become separated in their own class.
Primitive Obsession	The (excessive) use of pure primitives or record types (i.e., a structure of data into a meaningful group) to pass on data in software. This is often a complex and inefficient way to deal with data. Rather, the code can often be refactored in such a way that small set of primitives is grouped into a (small) object such as a money class with variables for the number, currency, ranges, etcetera.
Switch Statements	Switch statements have the tendency to indicate duplicate code in the source code as often the same switch statement is scattered about a program in different places. In case a new clause is added, removed or changed within the statement, all statements have to be found and changed. As such, it is proposed to refactor the code by use of polymorphism.
Parallel Inheritance Hierarchies	The phenomenon in which a change in a subclass of one class implies a change in the subclass of another class. This can be seen as a special case of Shotgun Surgery smell. The code should be refactored in such a way that the instance of one hierarchy refer to the instances of the other.
Lazy Class	Each class created requires effort to create, maintain and understand. Hence, in case classes are present which are not performing enough functionality to justify these efforts,the code should be refactored in such a way that they are removed.
Speculative Generality	The presence of methods and classes incorporating future functionalities, but which do not always tend to be used in practice. The code should be refactored in such a way that this overhead is reduced in order to improve understandability and maintainability.
Temporary Field	The situation in which a class has an instance variable which is only set in certain circumstances. This works confusing and adds to complexity. As such, the code should be refactored in such a way that the temporary fields are replaced to a new class, in which each instantiation effectively uses the fields.
Message Chains	When a client asks for a certain object, the situation might occur that this object makes a request to another object, making at its turn a request to yet another object, and so on. Such method chain creates coupling and a dependency between the client and the calling stack. The code should be refactored in ways like adding a separate method handling the chain navigation.
Middle Man	The phenomenon in which delegation is taken to an extreme situation in which a class is nearly passing all of its incoming requests to other classes performing the actual functionality. The code should be refactored in such a way that the delegate (“middle man”) is eliminated from the hierarchy structure.
Inappropriate Intimacy	The case in which a class is too “intimately” tied to another class, often reflected in a low degree of cohesion of the considered, as well as a high degree of coupling between them. The code should be refactored in such a way that the coupling between the classes is lowered by for instance moving fields (variables), methods, rearranging directional links between classes, etcetera.
Alternative Class with Different Interface	The occurrence of a number of methods doing the same thing, but having several different interfaces. Frequently, this goes hand in hand with duplicate code. The code should be refactored in such way that the methods are renamed and adapted so they all have the same name and interface, and duplicate code becomes removed.
Incomplete Library class	When reusing external library classes when building your own code, these library classes may turn out to be incomplete for performing all required functionalities. Most often, adapting these library classes is very difficult or simply impossible.
Data Class	The occurrence of classes only having data fields with getter and setter methods. They form “dumb” data classes, often being manipulated in too much detail by other classes. The code should be refactored in such a way the data fields become grouped in the same class as the methods which mostly perform actions upon them.
Refused Bequest	The case in which a subclass does not need (many) of the methods its inherits from its base class. Sometimes, this is an indication of a wrong class hierarchy. In this situation, the code should be refactored in such a way that these inconsistencies become removed.
Comments	If many commentary notes are present in the source code, this often indicates bad quality of the considered code as apparently, many aspects need additional clarification. They are often a symptom of the above mentioned code smells. The code should be refactored in such a way that only a little or no extra comment is required in the source code.

 ISBN: 978-1-61208-230-1

Table III
MAPPING THE 22 CODE SMELLS OF FOWLER ET AL. [4] WITH THE NORMALIZED SYSTEMS THEORY THEOREMS [1]–[3].

Code Smell	SoC	AvT	DvT	SoS
Duplicate Code	●			
Long Method	○			
Large Class	○			
Long Parameter		●	●	
Divergent Change	●	●	●	●
Shotgun Surgery	●	●	●	●
Feature Envy		contradicting		
Data Clumbs			●	
Primitive Obsession			●	
Switch Statements	○			
Parallel Inheritance Hierarchies		not related		
Lazy Class		not related		
Speculative Generality		contradicting		
Temporary Field		contradicting		
Message Chains				●
Middle Man				●
Inappropriate Intimacy	●			
Alternative Class with Different Interface	●			
Incomplete Library class	○			
Data Class		contradicting		
Refused Bequest		not related		
Comments		not related		

●: Code smell guideline fully complying with a Normalized Systems theorem
○: Code smell guideline partly or indirectly complying with a Normalized Systems theorem

being subject to only one kind of change at a time. An instance of the *Shotgun Surgery* smell highly resembles the definition of a combinatorial effect and the notion of instability, entailing that a small change might have an impact located in multiple (and eventually an unbounded amount) of places. All four NS theorems precisely aim at avoiding these smells to show up. These examples further clearly illustrate that the code smells could be regarded as a kind of *symptoms* of low evolvable software architectures (i.e., one does not want to be confronted with *Shotgun Surgery*), whereas the NS theorems aim to focus on the *root causes* of these symptoms (i.e., how can one avoid the occurrence of *Shotgun Surgery* based on a set of proven theorems).

Finally, some code smells only seem to be partially or indirectly supported by the NS theorems. For instance, *Long Methods*, *Large Classes* or the use of *Switch Statements* are in themselves no strict violations of any of the NS theorems. However, as Fowler et al. [4] argue that they often tend to give rise to duplicate code and the combination of multiple change drivers (i.e., a violation of Separation of Concerns), they can be thought of as indirectly supporting the NS theorems.

B. Code smells contradicting with the NS theorems

A limited set of four code smells seems to contradict with the Normalized Systems theorems. For instance, the code smells *Feature Envy* and *Data Class* require and recommend programmers to incorporate both data and the actions that are most commonly performed on this data, in one single construct (class). However, in [3] it was argued to analyze the dynamic nature of programming constructs in a multi-dimensional way (i.e., considering different versions for a data structure, different versions for the interface of a processing function and different versions for each of the tasks a function consists of). As the dimensions of variability increase even further when both data and actions are combined into one construct (e.g., a class in typical object-oriented methodologies), the NS theorems imply the use of separate data elements (encapsulated with its get- and set-methods and supporting tasks for cross-cutting concerns such as remote access and persistence) and action elements (containing a single functional tasks and encapsulated with supporting tasks for cross-cutting concerns such as logging and access control). The arguments and parameters needed by an action element are thus to be encapsulated separately into their own data element. This reasoning contradicts with the guidelines based on the code smells from Fowler et al [4]. To the extent that the *Temporary Field* code smell is advocating the same combination of methods together with a set of variables (which all have to be used by those methods), this code smell seems contradictory to the Normalized Systems theorems as well. Indeed, in NS, the identification of separate data elements is prescribed even when not every action element will necessarily use all the fields in every instance.

The *Speculative Generality* smell stresses that the incorporation of future functionalities should only be implemented when there is a reasonable chance that the functionality will eventually be used: the implementation of less likely future functionalities would only add unnecessary complexity. To

 ISBN: 978-1-61208-230-1

a certain extent, this might be considered as contradictory to the NS approach as NS would prescribe to isolate each change driver (i.e., each part of code which is anticipated to evolve independently) as soon as possible in its own construct as a way of anticipating all basic elementary future changes, irrespective of its frequency of occurrence. This might indeed introduce some additional design complexity initially, but avoids the occurrence of combinatorial effects later on. On the other hand, in parallel of the concerning code smell, NS would obviously also not prescribe to identify parts of code which are completely unlikely to change independently, as change drivers.

C. Code smells not related to the NS theorems

While most code smells seem to be supporting the NS theorems, some of them seem to be somewhat unrelated to NS as well. For instance, the *Lazy Class* smell deals with the deletion of code parts no longer used. This issue is not directly discussed in NS theory (which considers the deletion of unused parts to be an automatic process of garbage collection instead of a change to the information system) and therefore seems unrelated to the theorems. Equivalently, inheritance structures (cf. the *Parallel Inheritance Hierarchies* and *Refused Bequest* smells) are not really discussed in NS theory as it focuses on the very basic constructs of information systems in terms of data and actions. However, as inheritance typically suggests to use the typical combination in object-orientation of data and actions in one construct (i.e., a class) these smells do not seem to arise in NS systems (arguing for the use of separate data and action constructs). Finally, NS theory does not directly consider the use of *Comments* in the source code.

V. Conclusion and Future Work

In this paper, we presented Normalized Systems theory as an approach for building evolvable software systems based on a set of formally proven theorems, which seem to relate to existing (but often tacit) best-practice heuristic software engineering knowledge. With the aim of supporting the claim that the NS theorems correlate with this practitioners knowledge, we explored the relevance of the set of 22 bad code smells as formulated by Fowler et al. [4] in this regard. Each of the code smells was mapped onto the 4 NS theorems. The analysis showed that most of the bad code smells are reflected by NS reasoning, with the most prevalent impact apparently coming from the Separation of Concerns and Data version Transparency theorems. However, a set of 4 code smells seemed to be unrelated, while another set of 4 code smells even seemed to be contradicting with NS theorems. Besides relating both approaches to each other, this paper (1) supports the work of Fowler et al. [4] by offering a sound theoretical basis for most of their formulated heuristic design guidelines and (2) might offer practitioners more insights into how violations regarding NS theorems might manifest themselves in practice. To some extent, the code smells could then be regarded as a kind of symptoms of low evolvable software architectures, whereas the NS theorems aim to focusing on the root causes of these symptoms. Future research might then be aimed at relating other knowledge repositories regarding software engineering heuristics towards the NS theorems.

Acknowledgment

P.D.B. is supported by a Research Grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

References

[1] H. Mannaert and J. Verelst, *Normalized systems: re-creating information technology based on laws for software evolvability*. Koppa, 2009.

[2] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, vol. 76, no. 12, pp. 1210 – 1222, 2011.

[3] ——, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, pp. 89–116, 2012.

[4] M. Fowler, K. Beck, J. Brant, O. W., and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[5] M. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, pp. 395–431, 2006.

[6] 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.

[7] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 390–400.

[8] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, jan.-feb. 2010.

[9] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proceedings of the International Conference on Software Maintenance*, 2003.

[10] W. C. Wake, *Refactoring Workbook*. Addison-Wesley Professional, 2003.

[11] B. Shneiderman, *Software psychology: human factors in computer and information systems*. Winthrop Publishers, 1980.

 ISBN: 978-1-61208-230-1