Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification

Francesca Arcelli Fontana[†], Jens Dietrich[*], Bartosz Walter[‡], Aiko Yamashita[§], and Marco Zanoni[†]
[†]Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milano, Italy
{marco.zanoni,arcelli}@disco.unimib.it
[*]School of Engineering and Advanced Technology, Massey University, Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz
[‡]Faculty of Computing, Poznań University of Technology, Poznań, Poland
bartosz.walter@cs.put.poznan.pl
[§]Department of Information Technology, Oslo and Akershus University College of Applied Sciences, Oslo, Norway
aiko.fallas@gmail.com

***Abstract*—Anti-patterns and code smells are archetypes used for describing software design shortcomings that can negatively affect software quality, in particular maintainability. Tools, metrics and methodologies have been developed to identify these archetypes, based on the assumption that they can point at problematic code. However, recent empirical studies have shown that some of these archetypes are ubiquitous in real world programs, and many of them are found to not be as detrimental to quality as previously conjectured. We are therefore interested on revisiting common anti-patterns and code smells, and build a catalogue of cases that constitute candidates for "false positives". We propose a preliminary classification of such false positives with the aim of facilitating a better understanding of the effects of anti-patterns and code smells in practice. We hope that the development and further refinement of such a classification can support researchers and tool vendors in their endeavor to develop more pragmatic, context-relevant detection and analysis tools for anti-patterns and code smells.**

***Index Terms*—Anti-patterns, code smells, false positives, detection accuracy, conceptual framework**

I. Introduction

Anti-patterns [1–3] and code smells [4] are archetypes used to describe software design shortcomings that can negatively affect software quality, in particular maintainability. Numerous tools and methods have been developed to identify instances of these anti-patterns and code smells in a program in order to measure its quality. The emphasis on automated detection tools and techniques is based on the assumption that these archetypes can actually point out particular parts of the program that are problematic from the maintenance perspective. By identifying these parts, adequate remedies (i.e., refactoring and/or restructuring) can be applied in order to pay down technical debt [5]. However, recent empirical studies have demonstrated that some of these archetypes are rather ubiquitous in real world programs [6, 7] and not all of them are as detrimental to software quality as previously conjectured [8–10].

This poses a relatively complex challenge of how to use anti-patterns and smells as means to assess software quality. We see two major challenges, one of a practical and another one of a more methodological nature. First, the high number of identified instances leads to time-consuming manual examination, making the usage of code anti-pattern and smell detection for code inspection rather prohibitive. From a methodological perspective, we are facing a construct validity issue (i.e., "are we measuring really what we intend to measure?") as some of the proposed archetypes may constitute forms of program construction that neither reflect the current common understanding of what maintainability is [11] nor constitute the "latent variable" from an empirical perspective.

Thus, we are interested in revisiting common anti-patterns and code smells, and improve their comprehension by describing and classifying instances of cases that constitute "false positives". We conjecture that state-of-the-art anti-pattern and smell detection suffers from low precision due to the lack of attention on these false positives - anti-pattern and smell instances with no net detrimental effect on quality. A recent work addressing the definition of false positives can be found in [12], where some *filters* are proposed to increase the precision of smell detection tools. A *meta-synthesis* [13, 14] was conducted on: a) the last 10 years of empirical studies conducted on code smells and anti-patterns, b) theoretical examples from grey literature, and c) case studies from industry and open source projects. We summarize the results in a catalogue [15] where we provide the available information regarding the definition of each archetype, applicable synonyms, argumentation, sources used to support the argumentation (empirical studies or observations from open source/industrial cases), code examples, and contextual factors that can be used to judge whether an instance is a false positive or not.

In this paper, we propose a preliminary classification derived in a bottom-up fashion based on all the elements from the catalogue. The intention behind this classification is to contribute to a better, richer conceptual framework for code smells and anti-patterns research. We hope that further refinement of this classification can support researchers and tool vendors in their endeavour to develop more precise and therefore more useful tools and methods for anti-patterns and code smell detection.

The paper is organized as follows: Section II presents background and related work. Section III presents the approach followed to develop the catalogue and the classification. Section IV presents and discusses the results from the process. Section V concludes and suggests avenues for future work.

II. Background and Related Work

Anti-pattern is a term defined by Koenig [1] as the counterpart of design patterns: instead of providing a real solution to an identified design issue, it delivers a superficial substitute that does not properly address it. Violations or improper interpretations of widely accepted design principles such as SOLID [16] can also be considered as anti-patterns. 'Code smells' is a term coined by Fowler and Beck [4] for describing symptoms of deeper issues in the code. They informally described and exemplified 22 different code smells, and related them back to well-known violations of different programming and design principles. Different approaches for detecting anti-patterns and code smells have been proposed, and are currently in use. Some examples of the most recent efforts are *detection strategies* [17], which have been implemented in commercial and open source tools (e.g., inFusion[1] and PMD[2]), the *DECOR method* proposed by Moha et al., [18], *JDeodorant*[3], which matches different code attributes with refactoring opportunities, and *machine-learning algorithms* to discover relations between metrics and code smells (i.e., Arcelli et al. [19], Khomh et al. [20]). Recent approaches also consider code evolution via repository mining, e.g., Palomba et al. [21]. Systematic literature reviews reported in [22] and [23] identified empirical studies investigating the impact of code smells on maintenance. However, the results from both reviews are inconclusive with respect to the actual effects code smells have on maintainability. Furthermore, recent work by Sjøberg et al. [8] and Yamashita [9] suggests that the overall capacity of smell analysis to explain or predict maintenance problems is relatively modest. To this we need to consider the fact that recent studies have pointed out that the presence of certain anti-patterns is rather predominant in real world software systems [6, 7].

III. Methodology

Meta-synthesis was the methodology selected to perform this work. According to [13], meta-synthesis "attempts to integrate results from a number of different but inter-related qualitative studies. The technique has an interpretive, rather than aggregating, intent, in contrast to meta-analysis of quantitative studies." In our case, we decided to follow a meta-syntesis instead of a systematic review, because we believe that the later may be inadequate to identify all relevant literature. In particular, we also wanted to cover work that according to Noblit and Hare [24] is "refutational", or that may present oppositional conclusions from the main body of work or understanding/assumptions in a particular research topic. Quantitative meta-analysis (e.g., systematic literature reviews) aims at increasing certainty in cause and effect conclusions, whereas qualitative meta-synthesis (such as in our case) seeks to understand and explain phenomena. In particular, we attempt to look into the "exceptional" cases where code smells do not have negative consequences, as to better understand in which situations they are harmful and when they are not. We stress that we performed a meta-synthesis, not a systematic review. Meta-analysis is not about replicating a given result, but trying to find contradictory results. The definition we used for a false positive for the search was: a false positive for an antipattern AP or a code smell CS is a code structure that satisfies the conditions that define AP/CS, but it is not clear that they have a net detrimental effect on the quality of the software system under analysis, or considered in the case study. Net here refers to the fact that their might be *some* detrimental effect, but this is outweighed by some other benefits associated with this design. Meta-synthesis was conducted on: a) the last 10 years of empirical studies conducted on code smells and anti-patterns, b) practical examples from grey literature, and c) case studies from industry and open source projects. We searched for the terms "code smell", and "anti-pattern" in Google Scholar covering the remaining period 2012-2014. In addition, we looked at the references from Zhang [22] and [23] and searched in GoogleScholar studies citing them. We searched for grey literature on blogs, books and personal experiences from projects within academia and industry. We consider this set a good starting point to investigate false positives from both a theoretical and an empirical perspective. After we identified the studies and revelatory cases reported in detail in [15], we developed a catalogue as to summarize our findings, and via a bottom-up, iterative discussion, we built a situational classification of the false positives.

IV. Results and Discussion

A. Overview of the Catalogue

In Table I, we report an overview of the anti-patterns and smells that we can associate with at least one category of false positives. The table also contains examples in the last column. False positives have been categorized for ten code smells and two anti-patterns. The names of the false positives categories and examples used in the table refer to the definitions reported in Section IV-C.

B. Examples from the Catalogue

1) Abstract Singletons Cause Subtype Knowledge: The purpose of the Singleton pattern [25] is to provide classes that can only have one instance by design. Singletons are often used to provide a single access point for global services, such as object creation (in combination with the factory pattern), logging, or access to configuration information. There are good reasons to use Singleton in combination with abstraction. For instance, an abstract class is defined that provides the specification of the services provided by the singleton, and also provides access to the actual singleton instance via a static access method. However, the actual singleton instance is an

[1] http://www.intooitus.com/products/infusion [2] http://pmd.sourceforge.net
[3] http://www.jdeodorant.com/

TABLE I
FALSE POSITIVES CATALOGUE OVERVIEW

Code smell / antipattern	False Positive Categories	False Positive Example
Message Chains	1.1 Design Patterns, 2.3 Analysis Scope	Test Class Method, Builder DP
God Method	1.1 Design Patterns, 1.5 Porting from non-OO, 2.1 Source Code Generators, 2.3 Analysis Scope	GUI Library, GUI Builder, Test Class, Entity Modeling Class, Parser Class, Visitor DP, Persistence Class
God Class	1.1 Design Patterns, 1.3 Frameworks, 1.5 Porting from non-OO, 2.1 Source Code Generators, 2.3 Analysis Scope	GUI Library, GUI Builder, Test Class, Entity Modeling Class, Parser Class, Visitor DP, Persistence Class
Feature Envy	1.1 Design Patterns	Visitor DP
Dispersed Coupling	2.3 Analysis Scope	Test Class Method
Duplicated Code	2.1 Source Code Generators	
Data Clumps	1.4 Optimization, 1.5 Porting from non-OO	
Primitive Obsession	1.4 Optimization, 1.5 Porting from non-OO, 2.1 Source Code Generators	
Data Class	1.1 Design Patterns, 1.2 Programming Language, 1.3 Frameworks, 2.3 Analysis Scope	Exception Handling Class, Serializable Class, Test Class, Logger Class
Shotgun Surgery	1.2 Programming Language, 1.3 Frameworks, 2.2 Representation, 2.3 Analysis Scope	Exception Handling Method, Test Class Method, Getter/Setter Method
Circular Dependencies	1.1 Design Patterns, 2.1 Source Code Generators, 2.2 Representation	Visitor DP, AbstractFactory DP, Parser
Subtype Knowledge	1.1 Design Patterns	Visitor DP, Singleton DP, AbstractFactory DP

```java
public static NetworkAdmin getSingleton(){
  if (singleton==null) singleton=new NetworkAdminImpl();
  return singleton;
}
```

Fig. 1. Singleton instantiation in Vuze (in class NetworkAdmin)

instance of a concrete, instantiable subclass. If the Singleton acts as a factory, this separation results in an Abstract Factory.

Because the abstract singleton base class references its concrete subclass, a subtype knowledge antipattern (STK) [26] occurs. An example can be seen in Azureus[4] and is depicted in Fig. 1. It is often argued that this is indeed an anti-pattern, and there are several widely used approaches to break the reference from the abstract to the concrete type, such as using reflection (often by using additional abstractions such as "service locators" or dependency injection [27]). However, there are good reasons not to do this as any of these methods introduces additional complexity and therefore also has an impact on maintainability. On the other hand, if there is one default implementation that is used in almost all cases, then the approach seen in Fig. 1 seems like the best solution. Also note that while instances of STK are common in real world programs [7], there is little evidence that their presence has a negative impact on maintainability [10].

2) Data Classes May Improve Performance: Data Class is described by Fowler as a "dumb data holder" [4]. It means that the class is used only for storing data and providing read/write access to it, without any other functionality. This is a violation of the Single Responsibility Principle [28], as the infected class has actually no responsibility. However, several reports conclude that this smell has only little negative impact on maintainability of the subject class [8, 29, 30], and could be treated as a secondary issue to be addressed and fixed. Moreover, classes storing several primitive data items are recommended in EJB code as Data Transfer Objects, an enterprise-level design pattern [31]. The combination of attributes in a single entity improves performance of remote data transfers, which is often a critical factor that outweighs the detrimental effects of the code smell.

C. Classification

Category 1: Imposed Anti-patterns and smells – Imposed anti-patterns are side effects of *conscious* design decisions. An engineer has made the decision to design the software this way despite these anti-patterns or smells as s/he believes that the overall net effect on the quality of the system is positive.

Sub-category 1.1: Anti-patterns and smells imposed by Design Patterns – In some cases, design patterns can directly cause certain anti-patterns or smells. Examples in this category are:

- Logger Class: classes wrapping logging functionalities can be very simple Adapters, that mainly store configuration and redirect or expose existing logging functions. This kind of adapter is sometimes detected as Data Class.
- Visitors: visitors generate circular dependencies between visitor and visited classes, and potentially also the respective packages. Concrete Visitors are sometimes detected as God Classes when they implement a large number of `visit()` operations that contain complex algorithms, also `visit()` methods can also be detected as Feature Envy, since they inspect the content of unrelated classes.
- Singleton and Abstract Factory: some uses of the Singleton and Abstract Factory can induce Subtype Knowledge (see example in Section IV-B).
- Builder: instances of the Builder pattern, using a Fluent Interface style[5], can be detected as Message Chain; this is often wrong, since that style allows chaining calls to the same object, while the Message Chain smell addresses chain of calls to different objects.

Sub-category 1.2: Anti-patterns and smells imposed by the Programming Language used – Sometimes, certain designs

[4] http://sourceforge.net/projects/azureus/

[5] http://martinfowler.com/bliki/FluentInterface.html

are used to overcome limitations of the programming language used. The previously mentioned Visitor example illustrates this point as well: visitors are used to overcome the lack of double dispatch in modern object-oriented languages like Java and C# [32]. Other examples are:

- Exception Handling Class: custom exceptions are often just empty subtypes of existing exceptions or only add some descriptive attribute. Their main goal is to communicate why something happened, and this makes them appear as Data Classes, although most foundation classes of any language are prone to this pattern.
- Serializable Class: the protocol imposed by serialization in Java often results in creating Data Classes with the specific purpose of being serialized and de-serialized, and serve as input/configuration for other classes.

Sub-category 1.3: Anti-pattern and smells imposed by frameworks – Frameworks like J2EE sometimes recommend using certain antipatterns, trading them off with other qualitative attributes that are more valued in a given context. In that case the benefit of using a proven existing framework (or component) outweighs the negative effects of design decisions these frameworks may impose on the software using it. An example is the use of data transfer objects and entity beans in J2EE[6]: while they represent Data Classes, they serve the purpose of providing object-oriented access to the data model. Other examples identified in this category are:

- GUI widget toolkits: certain graphical toolkits provide interfaces that can be extended to integrate a new component to the user interface. These interfaces are often very large, and even just implementing them for simple functionalities risks to create a God Class.
- Getter/Setter Method: Java Beans prescribe the use of getters and setters. Since they are typically very simple methods, they risk to be used in many different parts of a system. Often, this property makes them to be detected as Shotgun Surgery instances. As an established pattern in Java beans, getters and setters can be safely excluded from being Shotgun Surgery, especially when they are in the simplest canonical form. Deciding if a class property can be exposed to a large part of the system is probably more tied to the analysis of the domain than to the applied technical solution.

Sub-category 1.4: Anti-patterns and smells imposed by optimisations – Sometimes, engineers make a conscious decision based on the advantages a given design decision poses over other alternatives that comply better with object-oriented design principles. This somehow resembles how de-normalisation is used to tune databases. As an effect, this can cause anti-patterns such as Primitive Obsession and Data Clumps. An example is the use of integer bit flags to encode state. This has been widely used in Java, for instance in `javax.swing.SwingConstants` or in `java.lang.reflect.Modifier`. Note that recent Java API additions like `EnumSet` and `EnumMap` have added features to Java that offer a more object-oriented replacement without a significant performance penalty [33].

Sub-category 1.5: Anti-patterns and smells imposed by porting code from a non-object-oriented programming language – Engineers sometimes port proven implementations written in imperative languages to object-oriented languages. This is commonly used in code that provides established, low-level algorithms in areas such as encryption, compression and random number generation. It is a safer strategy to faithfully port this code, often using a single static method. While this leads to various smells including Primitive Obsession, Large Classes and Large Methods, it is a reasonable decision that minimises project risks.

Sub-category 1.6: Anti-patterns and smells inherited from legacy code – Several programming languages miss features that could express the intent of the programmer more explicitly or effectively. Evolution of the language could extend its capabilities to better address such issues. However, legacy code often remains unchanged. As an example, consider marker interfaces in Java, frequently exploited in the pre-JDK5.0 code to add meta data at a class level. Starting from JDK5.0, they have been effectively replaced with annotations.

Category 2: Inadvertent anti-patterns and smells – Inadvertent antipatterns are created by tools that create (generate) or consume (analyze) code.

Sub-category 2.1: Inadvertent anti-patterns and smells caused by source code generators – Studies have shown that code that is generated automatically is often riddled with anti-patterns and smells (mainly duplicates and size/complexity-related smells) [18, 34]. A simple way of avoiding this kind of false positives (when applicable) is to exclude folders containing only generated code from the analysis. Examples identified in this category are:

- GUI builders: often GUIs are created using graphical editors that generate code. When this code is generated as a single class, it is often detected as God Class.
- Entity Modelling Class: Generated code to support entity management can generate large and complicated classes (e.g., Eclipse EMF[7]) with complex and duplicated code.
- Parsers: parser generators (e.g., ANTLR) tend to create a single monolithic parser class. This code is often identified as God Class, Duplicated Code, and is prone to Primitive Obsession and Circular dependencies due to generated code for AST visitors.

Sub-category 2.2: Inadvertent anti-patterns and smells caused by program representation – Anti-pattern and smell analysis is typically static analysis that investigates a representation of the program without executing it. However, there are different program representations with subtle differences that may have an impact on the precision of anti-pattern detection. A good example is Java with its source code and byte code representations. The compiler infers new artefacts that can create new anti-patterns or smell instances. An example is constant inlining (that could create Shotgun Surgery instances) and

[6] http://www.oracle.com/technetwork/java/transferobject-139757.html

[7] https://eclipse.org/modeling/emf/

the creation of new methods for accessing private members in outer classes from inner classes. The use of (non-static) nested classes leads to Circular Dependencies between the respective classes caused by the compiler-generated reference to an instance of the enclosing class(es), and in some cases to subtype knowledge.

Sub-category 2.3: Inadvertent anti-patterns and smells caused by analysis scope – Software projects are composed of different artifacts and modules. Certain parts of the project may not reach production, and should not be considered during quality assessment, or considered separately. An example is:

- Test Classes and Methods: test code usually follows its own design rules imposed by respective frameworks, quite different from project ones. It is usually sufficient to carefully select the source folders/files to analyze to exclude tests, but sometimes they are mixed with the main code.

V. Conclusion and Future Work

Anti-pattern and smell detection relies on constructing classifiers that assign a given entity to one of two groups: infected and (anti-pattern/smell)-free. Usually these classifiers utilize static analysis: metrics values, structural properties, history of changes or relations with other anti-patterns or code smells. However, empirical studies show that such detectors produce numerous false-positives: entities that are incorrectly classified as infected, whereas manual inspection reveals that they have no net detrimental effect on software quality.

The detection of the proposed false positive categories could be implemented as a set of filters. This is a possible step forward in an effort to define domain- (or context-) specific smells and anti-patterns that are relevant only in a specific context, but meaningless or misleading in other contexts.

The main contribution of this paper is a classification of false positive code smell and antipattern instances. This classification is detailed further in a catalogue [15], which has the aim of collecting descriptions, examples, and references to previous empirical evidence that can be used to define proper filters. Future work includes the implementation of these false positive definitions via filter definitions in code smell detection tools and we plan to apply the filters into InFusion[8] and evaluate the effects of the filters on inspection effort (where a proxy for effort constitutes the number of classes needed to be inspected).

References

[1] A. Koenig, "Patterns and antipatterns," *J. of Object-Oriented Programming*, vol. 8, no. 1, pp. 383–389, 1995.

[2] W. Brown, R. Malveau, T. Mowbray, and J. Wiley, *AntiPatterns: refactoring roftware, architectures, and projects in crisis.* Wiley, 1998, vol. 3, no. 4.

[3] B. F. Webster, *Pitfalls of object-oriented development.* M&T Books, 1995.

[4] M. Fowler, *Refactoring: Improving the design of existing code.* Addison-Wesley, 1999.

[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, no. 0, pp. 193–220, 2015.

[6] H. Melton and E. Tempero, "An empirical study of cycles among classes in Java," *Empirical Soft. Eng.*, vol. 12, no. 4, pp. 389–415, 2007.

[7] J. Dietrich, C. McCartin, E. Tempero, and S. Shah, "Barriers to modularity – An empirical study to assess the potential for modularisation of Java Programs," in *Proceedings QoSA2010*, 2010.

[8] D. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. on Soft. Eng*, vol. PP, no. 99, p. 1, 2013.

[9] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Soft. Eng.*, vol. 19, no. 4, pp. 1111–1143, aug 2014.

[10] T. D. Oyetoyan, J.-R. Falleri, J. Dietrich, and K. Jezek, "Circular dependencies and change-proneness: An empirical study," in *Proceedings SANER'15.* IEEE, 2015.

[11] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" 2012, pp. 306–315.

[12] F. Arcelli Fontana, V. Ferme, and M. Zanoni, "Filtering code smells detection results," in *Proceedings ICSE'15 (poster track).* IEEE, 2015.

[13] D. Walsh and S. Downe, "Meta-synthesis method for qualitative research: A literature review," *J. of Adv. Nursing*, vol. 50, no. 2, 2005.

[14] University of Toledo, "Types of literature reviews," 2014.

[15] F. Arcelli Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Preliminary catalogue of anti-pattern and code smell false positives," Poznań University of Technology, Tech. Rep. RA-5/15, 2015. [Online]. Available: http://www2.cs.put.poznan.pl/wp-content/uploads/2015/11/RA-5-2015.pdf

[16] R. C. Martin, *Agile Software Development, Principles, Patterns and Practice.* Prentice Hall, 2002.

[17] M. Lanza and R. Marinescu, *Object-oriented metrics in practice.* Springer, 2006.

[18] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. on Soft. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[19] F. Arcelli Fontana, M. Zanoni, A. Marino, and M. Mäntylä, "Code smell detection: towards a machine learning-based approach," in *Proceedings ICSM'13.* IEEE, 2013.

[20] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," *J. of Systems and Software*, vol. 84, no. 4, pp. 559–572, apr 2011.

[21] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings ASE'13.* IEEE, 2013.

[22] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, apr 2011.

[23] A. Yamashita, "Assessing the capability of code smells to support software maintainability assessments: Empirical inquiry and methodological approach," Doctoral Thesis, Univ. of Oslo, 2012.

[24] G. Noblit and R. Hare, *Meta-Ethnography: Synthesising qualitative studies.* Newbury Park: Sage, 1988.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson, 1994.

[26] A. J. Riel, *Object-oriented design heuristics*, 1st ed. Boston, MA, USA: Addison-Wesley, 1996.

[27] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004, http://www.martinfowler.com/articles/injection.html.

[28] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[29] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, jul 2007.

[30] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceeding WCRE'09.* IEEE, 2009.

[31] M. Fowler, *Patterns of enterprise application architecture.* Addison-Wesley Professional, 2003.

[32] R. Muschevici, A. Potanin, E. Tempero, and J. Noble, "Multiple dispatch in practice," in *Proceedings OOPSLA'08.* New York, NY, USA: ACM, 2008.

[33] J. Bloch, *Effective Java (2nd ed.)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[34] C. Parnin and C. Görg, "Lightweight visualizations for inspecting code smells," in *Proceedings SoftVis'06.* ACM, 2006.

[8] https://www.intooitus.com/products/infusion