GENERATING SOFTWARE PRODUCT LINE MODEL BY RESOLVING CODE SMELLS IN THE PRODUCTS' SOURCE CODE

Sami Ouali

University of Technology and Applied Sciences - College of Applied Sciences, Ibri, Oman

ABSTRACT

Software Product Lines (SPLs) refer to some software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production. This concept is recognized as a successful approach to reuse in software development. Its purpose is to reduce production costs by reusing existing features and managing the variability between the different products with respect of particular constraints. Software Product Line engineering is the production process in product lines and the development of a family of systems by reusing core assets. It exploits the commonalities between software products and preserves the ability to vary the functionalities and features between these products. The adopted strategy for building SPL can be a top-down or bottom-up. Depending from the selected strategy, it is possible to face an inappropriate implementation in the SPL Model or the derived products during this process. The code can contain code smells or code anomalies. Code smells are considered as problems in source code which can have an impact on the quality of the derived products of an SPL. The same problem can be present in many derived products from an SPL due to reuse or in the obtained product line when the bottom-up strategy is selected. A possible solution to this problem can be the refactoring which can improve the internal structure of source code without altering external behavior. This paper proposes an approach for building SPL from source code using the bottom-up strategy. Its purpose is to reduce code smells in the obtained SPL using refactoring source code. This approach proposes a possible solution using reverse engineering to obtain the feature model of the SPL.

KEYWORDS

Software Product Line, Code smells, Refactoring, Reverse Engineering, Feature Model.

1. INTRODUCTION

Some organizations and companies build products that have some characteristics in common such as car corporation, phone company, software company that creates multiple software applications... These software applications can share the same architecture, execution platforms, covered business... It is important to manage in the good way these commonalities between the different products. The software product line was designed for this purpose to manage software products commonalities and variabilities.

Software Product Line (SPL) is a family of related software systems with common and variable functions whose first goal is reusability [1]. The SPL approach is tackling the goal of increasing the software productivity and quality by relying on the similarity that exists among software systems, and by managing a family of software systems in a reuse-based way. This concept is inspired from the industrial product lines. SPL aims to minimize effort and cost of development and maintenance, to reduce time-to-market and to ameliorate quality of software [2], [3], [4]. Unsuitable development of a SPL's assets (top-down strategy) or software products (bottom-up strategy) may give rise to bad programming practices, called code anomalies, also referred in the literature as "code smells" [5].

Code smell is often considered as key indicator of something wrong in the system code [5] or undesired code source property. Like all software systems, artifacts of a SPL may contain several code anomalies [6]. Therefore, if these code smells are not systematically removed, the SPL's quality may degrade due to evolution. If the anomaly is present in the SPL model, the problem will propagate in the derived products. Code Smells are very-known in classic and single software systems [7]. However, in the context of SPL, Code Smell is a new topic. [8] proposed a specific SPL's smell, called "Variability Smells". [9] discussed two types of bad smells related on SPL: Architectural Bad Smells and Code Bad Smells. [6] and [10] proposed detection strategies for anomalies in SPL.

The main goal of this work is to propose a solution to reduce code smells in SPL and especially when the selected strategy to build it is the bottom-up. This strategy opts for building the SPL from existing assets and products. Unsuitable development of a SPLs, assets or products may give rise to bad practices such as architectural smells and code smells. Our work tries to reduce development problems through the source code analyze of product variants to detect and correct code smells, identify the variability and build the feature model of SPL. The main purpose in our approach focuses on detecting code anomalies in source code and refactoring it to improve the quality of the obtained SPL. Then, the reverse engineering is the selected technique to design the SPL and to obtain the features/variability model with an input consisting of the different code smell free source code.

This rest of this paper is organized as follow: Section 2 introduces some essential concepts related to Software Product Line, code smells, reverse engineering, and refactoring, Section 3 presents the related work. Section 4 shows the proposed approach and its components. The last section concludes and presents our future works.

2. BACKGROUND

2.1. Software Product Lines

Software reuse is the process of implementing software systems using existing software assets. Software reuse has proven his efficiency as a way to help improving software development quality and productivity. Software development communities recognize SPL as a successful approach for reuse [11], [12]. The reuse in the context of SPL consists on the shared architecture, common assets… This reuse has an impact to produce a better quality software, provide a powerful competitive advantage, and reduce the production costs and time to market. SPL is a software development paradigm that share common feature to satisfy the specific needs of particular market segment [13].

Software product line's approach focus on the sharing of a reference architecture between products and managing the variability between them. This variation is managed with respect of particular characteristics and constraints. The variability in the context of SPL is defined as the ability of a core asset to adapt to usages in the different product contexts that are within the product line scope [14]. Variability must be anticipated by the identification of possible future uses or evolutions and should be continuously maintained to obtain wished results. The software product line engineering (SPLE) is the development process of product lines which tries to maximize the commonalities and reduce the cost of variations [15]. The SPLE process focuses on two levels of engineering [14]: Domain Engineering (DE) and Application Engineering (AE). DE focuses on developing reusable artifacts which are used in AE to construct and derive a specific product. Figure. 1 presents the SPLE process and his two levels.

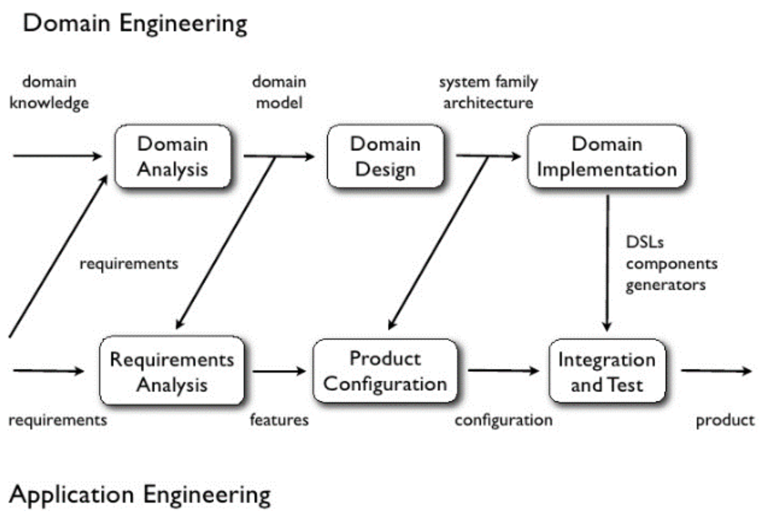

Figure 1. Domain Engineering and Application Engineering [14]

The Domain Engineering is also known as the engineering for Reuse. It corresponds to the study of the area of product line, identifying commonalities and variabilities among products, the establishment of a generic software architecture and the implementation of this architecture [11]. The Application Engineering, known as the engineering by reuse. It is used to find the optimal use for the development of a new product from a product line by reducing costs and development time and improve the quality [11]. One concept appears using these two domains known as the configuration management which consists on the configuration of artefacts, the selection of alternatives...

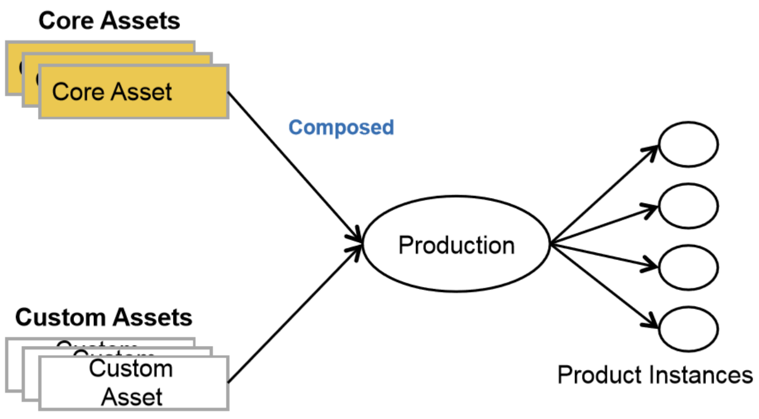

Figure 2. Products derivation [14]

Figure 2. presents the way how the product instances can be derived from the core assets and the custom assets during the application engineering. The core assets can be found in the product instances. The custom assets should be fixed using the configuration management by selecting the alternatives, options…

2.2. Code Smells

A software system evolves over time. Its evolution is one of the critical phases of the process of its development. Moreover, the software system changes, moreover the structure of the program deteriorates. So, complexity increases until it becomes more profitable to rewrite it from the scratch. Which can involve threats on the software quality.

Software system's bad quality is a key indicator of existing bad programming practices, also known in the literature as source code flaw, code smells or code anomalies [5].

Code smells are usually symptoms of low-level problems such as anti-patterns. They are indicators of something wrong that structures in the source code [5], their presence can affect in maintenance and slow down software development. Also, the software evolution can face some issues related the misunderstanding of the existing code.

In literature, different Code Smells have been defined. For instance, in Fowler's book [5], Beck define a list of 22 code smells. Some of these code smells are related to the incomplete or incorrect application of object-oriented programming principles, methods and classes with a huge size that they are hard to work with, excessive coupling between classes, unneeded code, expansive change in the code... For example, "Long Method" is a method that is too long and has too many responsibilities, so it makes code hard to maintain, understand, change, extend, debug and reuse. "Large Class" is a class that contains many fields, methods or lines of code. This means that the class is trying to do too much. "Long Parameter List" is a code smell related to the used list of parameters. This smell has an impact on the understanding of such list of parameters. "Duplicated Code" is considered as an unneeded code and has negative impacts on software development and maintenance. For example, they increase bug occurrences: if an instance of duplicate code is changed in one part of the code for fixing bugs or adding new features, code may require various changes in other parts all over the source code simultaneously; if the correspondents are not changed inadvertently, bugs are newly introduced to them [16].

2.3. Reverse Engineering

Reverse Engineering also called backwards engineering or back engineering is the process of analyzing a system. The purpose is to identify system structure, its components and the relationships between them [17]. This concept is not only related to computer sciences, but for any product, it can be applied to identify the product design from the different parts and components.

Reverse Engineering can create representations of the system through transformations between or within different abstraction levels. It can also extract design information from source code [17] and may be used to re-implement the system.

The reverse engineering process can be done through automated analysis or manual annotations. The next steps concern the identification of program structure and the establishment of traceability matrix. In our approach, we intend by using reverse engineering to obtain the design of an SPL from the identified concepts and features present inside the variant source code. These features represent the common core assets and custom assets. The purpose of the traceability matrix is to link the design components with the corresponding assets present in the original source code variants.

2.4. Refactoring

Refactoring's purpose is to improve the quality and maintainability of an existing code [5] and to make the software easier to understand. This process tries through the software system changing to improve its internal structure without having an impact on the external behavior of the code. It is a process for restructuring the source code by applying a series of basic micro-refactoring, each of which is a tiny change in the program's source code.

There are many code refactoring techniques such as removing code duplication, moving features between objects, simplifying conditional expressions, simplifying method calls, breaking code, encapsulating fields, generalizing types…

Refactoring can be a solution for code smells. This process takes as input a source code with problems and outputs good ones. The resulting code can be reused. The refactoring allows the code smells identification. Also, it offers the possibility to change the original code containing these code smells by code restructuration to get an output code without code smells.

3. RELATED WORK

Common industrial practices lead to the development of similar software products, then they are usually managed to each other using simple techniques, e.g., copy-paste-modify. This is bad practice leading a low software quality, as we mentioned above the "Duplicated Code" code smell. During the past few years, several studies have investigated two things: how to detect code smells [18], [19], [20], [21], [22], [23] and how to correct [5], [18], [24] them in a single software. To the best of our knowledge we found few studies [6], [8], [9], [10], [25], [26] that can be considered related to our research.

[9] performed a Systematic Literature Review (SLR) to find and classify published work about bad smells in the context of SPL and their respective refactoring methods. They classified 70 different bad smells divided in three groups: (i) Code Smells; that are symptoms of something wrong in the source code, (ii) Architectural Smells; that are an indication of problem in higher levels of abstraction and (iii) hybrid Smells; that are a combination between architectural smell and code smells. [26] proposed a method to derive metric thresholds for software product lines.

The goal is to define thresholds values that each metric can take in order to identify potential problems in the implementation of features. They use 4 software metrics: Lines of Code (LOC) counts the number of uncommented lines of code per class. The value of this metric indicates the size of a class. Coupling between Objects (CBO) counts the number of classes called by a given class. CBO measures the degree of coupling among classes. Weight Method per Class (WMC) counts the number of methods in a class. This metric can be used to estimate the complexity of a class. Number of Constant Refinements (NCR) counts the number of refinements that a constant has. Its value indicates how complex the relationship between a constant and its features is. Their study is based on 33 SPLs which are divided into three benchmarks according to their size in terms of Lines of Code (LOC).

Benchmark 1 includes all 33 SPLs. Benchmark 2 includes 22 SPLs with more than 300 LOC. Finally, Benchmark 3 is composed of 14 SPLs with more than 1,000 LOC. The goal of creating three different benchmarks is to analyze the results with varying levels of thresholds. In term of that they illustrate a detection strategy to detect two types of code smells, God Class and Lazy Class. Figure 3 presents the way to identify God Class and Lazy Class.

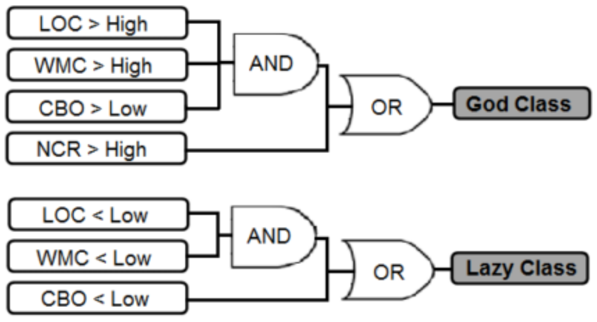

Figure 3. Code Smells identification.

Apel et al. [8] proposed bad smell specific to SPLs called variability smell; that is an indicator of an existing undesired property in all kinds of artifacts in an SPL, such as feature models.

Fernandes and Figueiredo [6] investigated code anomalies in the context of SPLs, they propose new detection strategies for well-known anomalies in SPL such as God Class and God Method, ultimately they propose new anomalies and their detection strategies and they propose supporting tool for the proposed detection.

De Andrade et al. [25] conducted an exploratory study that aims at characterizing architectural smells in the context of software product line.

Abilio et al. [10] proposed means to detect three code smells (God Method, God Class, and Shotgun Surgery) in Feature-Oriented Programming source code, FOP is a specific technique to deal with the modularization of features in SPL. They performed an exploratory study with eight SPLs developed with AHEAD; which is an FOP language, to detect code smells in a SPL by using 16 source code metrics. These metrics corresponds to the detection of three code smells mentioned above. Table 1 presents some of these metrics that can be used to calculate some formulas to detect three code smells when the formulas are corresponding to some defined values.

Table 1. Metrics used to detect code smells [10]

Acronym	Name	Description
NOF	Number of Features	Number of Features which has code artifacts
NCR	Number of Constant Refinements	Number of refinements which a constant has
NMR	Number of Method Refinements	Number of refinements which a method has
TNCt	Total Number of Constants	Number of constants (classes, interfaces - constant)
TNR	Total Number of Refinements	Total of refinements (classes, interfaces - refinement)
TNMR	Total Number of Method Refinements	Total of refinements of a method
TNRC	Total Number of Refined Constants	Total of refined constants
TNRM	Total Number of Refined Methods	Total of refined methods

Considering the discussed related work, we propose an approach aiming to develop an SPL with minimal code smells risks. After the code smells' detection and identification, the source code should be refactored to obtain a clean code i.e. without code anomalies to facilitate the task of features identification inside the refactored source code. These features can be considered as a common core asset or custom asset. If the feature exists in all the variants, it is going to be considered as mandatory/common, otherwise it is a custom asset.

4. Proposed Approach

The main goals in our study are to (i) investigate the state of the art on code smells in the context of SPLs as we show above, (ii) propose a solution to decrease code smells in developing software product lines.

Unsuitable development of a SPLs may give rise to bad practices such as architectural smells and code smells that induce maintenance and development costs problems. Therefore, we propose to build an SPL from the scratch using reverse engineering methods, which can help us to detect and correct code smells from the start. Thus, we can guarantee great quality of SPL.

The main challenge in this task is to analyze the source code of product variants in order to (i) detect and correct code smells, (ii) identify the variability among the products, (iii) associate them with features and (iiii) regroup the features into a variability model. The proposed approach is object-oriented language and only uses as input the source code of product variants.

First of all, we use as input source code of product variants then we apply detection strategies for code anomalies as duplicated code, uncovered code by unit tests and too complex code, after that we correct them using an automated bad smell correction technique based on the generation of refactoring concepts. Refactoring is a change made to the internal structure of software to rewrite the code, to “clean it up”, to make it easier to understand and cheaper to modify without changing its observable behavior [27]. In step 2 and after having a clean code, we are interested in the determination of the semantic relations between the names of the classes, the names of the methods and the attributes of all the source codes of the existing products having different terminologies and not necessary having the same meaning. In term of that we are interested in the harmonization of names, and more particularly in unifying fragments of source codes. During unification, we determine the semantic correspondences between the source code elements based on semantic knowledge base YAGO [28].

YAGO is a semantic knowledge base derived from many data sources like Wikipedia, WordNet, WikiData, GeoNames, and other. Aside YAGO, we will base on Machine Learning methods to get better semantic correspondences between source code elements. In fact, Machine Learning algorithms can be helpful in the classification of the features. Machine Learning proved his efficiency in many complex domains like Predictive Analytics [29], image processing [30], and signal processing… At the end of this step, all names with a semantic relationship would be harmonized and can be further analyzed in the next step of identifying commonalities and variability. Thus, we extract features by identification of common block (CB) and variation blocks (VB). CB groups the elements present in all the products while VB groups the elements present in certain products and not all of them. The role of these blocks is to group subsets to implement features. Once the common block and the variation blocks are completed, the extraction of mandatory elements and variation atomic blocks is supported, we associate them to features. Once the common properties and variability of product variants are identified, the feature model(s) will be constructed. Consequently, we can obtain one or more than one SPL. Our approach is presented in Figure 4.

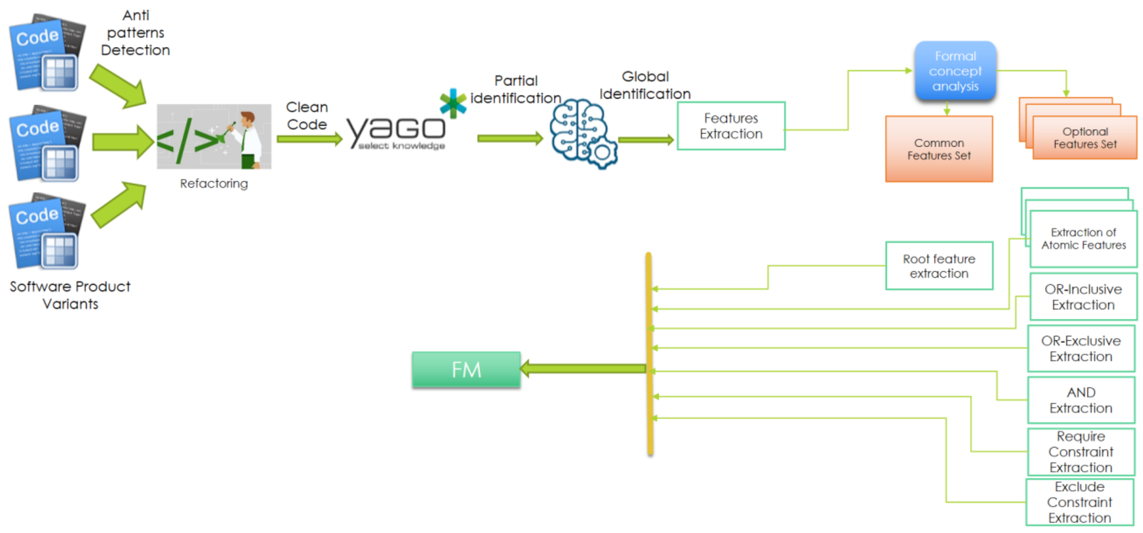

Figure 4. Proposed Approach.

Our approach is using the bottom-up strategy in the software product line construction. The process for this strategy begins with the definition of the shared resources and development of the generic core assets. The shared resources and the generic core assets are directly identified from the source code products. Then, the SPL's feature model is obtained through the reverse engineering. But, before the obtaining of this feature model, we need a process using the semantic knowledge base, ontologies, and artificial intelligence to detect and extract the common core assets and custom assets from the source code after the refactoring process.

5. CONCLUSIONS AND FUTURE WORKS

Developing from scratch or "reinventing the wheels" can be problematic for the company that tries to offer many products in a restricted time interval. Software reuse can be a solution for this problem. Reusing the existing features is an important challenge in software engineering. Software Product Line is one of the technique used to ensure the success of this challenge by defining some common core assets and custom assets. The obtained products can contain reuse the common core assets and some of the custom assets. These parts or assets can include some problems in their source code more known as Code Smells. These problems can propagate between the different products.

To resolve the problem related to code smells present in the source code, the anomalies should be detected and identified. Then, a refactoring process can improve the internal structure of software system by revising its content using some restructuration techniques and correcting the code anomalies.

In this paper, we present an approach adopting the bottom-up strategy which combines refactoring to eliminate code smells and reverse engineering to propagate modifications to the design level. Our purpose is to obtain a software product line model free from code smells. Our input is the source code for some product variants containing code anomalies to be resolved using the refactoring. The identification of the common core assets and custom assets is the next step using the artificial intelligence and ontologies. Finally, the reverse engineering is used to redesign the SPL's feature model.

Our future works will be the refinement of the different parts of the approach. Also, we will choose the appropriate tools to use in our prototype by comparing the proposed tools in the literature, identifying their fields of activity, evaluating the programming languages concerned by these tools, identifying the studied code smells, detecting only or detecting and correcting, identifying the way to combine many of these tools…

REFERENCES

[1] Rincón, L. F. et al., 2014. An ontological rule-based approach for analyzing dead and false optional features in feature models. In Electronic Notes in Theoretical Computer Sciences, Vol. 302, pp 111–132.

[2] Jacobson, I. et al., 1992. Object-oriented software engineering: a use case driven approach. Addison-Wesley, USA.

[3] Xue, Y., 2011. Reengineering legacy software products into software product line based on automatic variability analysis. Proceedings of the 33rd International Conference on Software Engineering, New York, USA, pp. 1114–1117.

[4] Ouali, S. et al., 2012. From Intentions to Software Design using an Intentional Software Product Line Meta-Model. Proceeding of the 8th International Conference on Innovations in Information Technology, Al Ain, UAE.

[5] Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

[6] Fernandes, E. and Figueiredo, E., 2017. Detecting Code Anomalies in Software Product Lines". Proceedings of 7th Brazilian Conference on Software: Theory and Practice, Maringa, Brazil, pp. 49–55.

[7] Zhang, M. et al., 2011. Code Bad Smells: A Review of Current Knowledge. In Journal of Software Maintenance and Evolution: Research and Practice, Wiley Online Library, pp. 179-202.

[8] Apel, S. et al., 2013. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer.

[9] Vale, G. et al., 2014. Bad Smells in Software Product Lines: A Systematic Review. Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), Brazil, pp. 84-94.

[10] Abilio, R. et al., 2015. Detecting Code Smells in Software Product Lines-An Exploratory Study. Proceeding of the 12th International Conference on Information Technology - New Generations, pp. 433–438.

[11] Ouali, S. et al., 2011. Framework for evolving software product line. In International Journal of Software Engineering & Applications, Vol. 2, No. 2, pp. 34-51.

[12] Weiss, D. M. and LAI, C. T. R., 1999. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley.

[13] Pohl, K. and Metzger, A., 2006. Software Product Line testing. In Communication of the ACM, pp78-81.

[14] Czarnecki, K. and Eisenecker, W., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.

[15] Thiel, S. and Hein, A., 2002. Modeling and Using Product Line Variability in Automotive Systems. In IEEE Software Vol.19, No.4, pp. 66-72.

[16] Hotta, K. et al., 2012. An Empirical Study on the Impact of Duplicate Code. Advances in Software Engineering.

[17] Chikofsky, E. J. and Cross, J. H., 1990. Reverse engineering and design recovery: A taxonomy. IN IEEE Software, Vol.7, pp. 13–17.

[18] Moha, N. et al., 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. Transactions on Software Engineering, Vol. 36, No. 1, pp. 20-36.

[19] Sjoberg, D. et al., 2013. Quantifying the effect of code smells on maintenance effort. Software Engineering IEEE Transactions, Vol. 39, No. 8, pp. 1144–1156.

[20] Van Emden, E. and Moonen, L., 2002. Java quality assurance by detecting code smells. Proceeding Working Conf. Reverse Engineering, IEEE Computer Society Press, pp. 97—107.

[21] Marinescu, C. et al., 2005. Iplasma: An integrated platform for quality assessment of object-oriented design. Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary.

[22] Liu, X. and Zhang, C., 2016. DT: a detection tool to automatically detect code smell in software project. Proceedings of the 2016 4th International Conference on Machinery, Materials and Information Technology Applications, vol. 71, pp. 681–684.

[23] Fontana, F. et al., 2012. Automatic Detection of Bad Smells in Code: An Experimental Assessment. In Journal of Object Technology.

[24] Campbell, D. and Miller, M., 2008. Designing refactoring tools for developers. Proceedings of the 2nd Workshop on Refactoring Tools, New York, NY, USA.

[25] De Andrade, H. S. et al., 2014. Architectural bad smells in software product lines. Proceedings of the 1st International Conference Dependable and Secure Cloud Computing Architecture, pp. 1–6.

[26] Vale, G. and Figueiredo, E., 2015. A Method to Derive Metric Thresholds for Software Product Lines. Proceedings 29th Brazilian Symposium on Software Engineering, pp. 110–119.

[27] Regulwar, G. B. and Tugnayat, R. M., 2012. Bad Smelling Concept in Software Refactoring. International Proceedings of Economics Development and Research, Vol.45, pp. 56–61.

[28] Rebele, T. et al., 2016. YAGO: a Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames. Proceeding of the 15th International Semantic Web Conference, Kobe, Japan.

[29] Demsar, J. et al., 2004. Orange: From experimental machine learning to interactive data mining. Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy.

[30] Ebrahimi, K. S. et al., 2013. Combining modality specific deep neural networks for emotion recognition in video. Proceedings of the 15th ACM on International conference on multimodal interaction, Sydney, Australia.

AUTHORS

Sami Ouali holds a PhD in Computer Sciences issued from the University of Manouba, Tunisia. He is actually an assistant professor in the University of technology and Applied Sciences (UTAS), College of Applied Sciences lbri, Oman. He is a member of the RIADI labs, Tunisia. His research interests lie in the areas of computer sciences, software engineering and software product line.