Received 21 June 2023, accepted 18 July 2023, date of publication 1 August 2023, date of current version 19 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3300575

RESEARCH ARTICLE

An Ontology-Based Approach to Reduce the Negative Impact of Code Smells in Software Development Projects

IVIAN L. CASTELLANO[1], GILBERTO FERNANDO CASTRO AGUILAR[2,3], NEMURY SILEGA[4], TAHIR KAMAL[5], MEHDHAR S. A. M. AL-GAASHANI[6], (Graduate Student Member, IEEE), NAGWAN ABDEL SAMEE[7], AND MAALI ALABDULHAFITH[7]

[1]Oficina de Estudios de Posgrado, International University of Andalusia, 41092 Seville, Spain
[2]Facultad de Ingeniería, Universidad Católica de Santiago de Guayaquil, Guayaquil 090109, Ecuador
[3]Facultad de Ciencias Matemáticas y Físicas, Universidad de Guayaquil, Guayaquil 090511, Ecuador
[4]Department of System Analysis and Telecommunications, Southern Federal University, 347900 Taganrog, Russia
[5]College of Computer Science and Technology, Zhejiang Normal University, Zhejiang 321017, China
[6]College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China
[7]Department of Information Technology, College of Computer and Information Sciences, Princess Nourah bint Abdulrahman University, Riyadh 11671, Saudi Arabia

Corresponding author: Maali Alabdulhafith (MIAlabdulhafith@pnu.edu.sa)

This work was funded by Princess Nourah bint Abdulrahman University Researchers Supporting Project number (PNURSP2023R407), Princess Nourah bint Abdulrahman University, Riyadh, Saudi Arabia.

ABSTRACT The quality of software systems may be seriously impacted by specific types of source code anomalies. For example, poor programming practices result in Code Smells (CSs), which are a specific type of source code anomalies. They lead to architectural problems that consequently impact some significant software quality attributes, such as maintainability, portability, and reuse. To reduce the risk of introducing CSs and alleviate their consequences, the knowledge and skills of developers and architects is essential. On the other hand, ontologies, which are an artificial intelligence technique, have been used as a solution to deal with different software engineering challenges. Hence, the aim of this paper is to describe an ontological approach to representing and analyzing code smells. Since ontologies are a formal language based on description logics, this approach may contribute to formally analyzing the information about code smells, for example, to detect inconsistencies or infer new knowledge with the support of a reasoner. In addition, this proposal may support the training of software developers by providing the most relevant information on code smells. This ontology can also be a means of representing the knowledge on CSs from different sources (documents in natural language, relational databases, HTML documents, etc.). Therefore, it could be a valuable knowledge base to support the struggle of software developers and architects either to avoid CSs or to detect and remove them. The ontology was developed following a sound methodology. The well-known tool Protégé was used to manage the ontology and it was validated by using different techniques. An experiment was conducted to demonstrate the applicability of the ontology and evaluate its impact on speeding up the analysis of CSs.

INDEX TERMS Code smells, knowledge representation, ontology, reasoning, software quality.

I. INTRODUCTION

Ensure software quality is a relevant goal of software engineering. Maintainability is one of the quality characteristics defined by ISO/ IEC 25010 [1]. It is defined as the degree of effectiveness and efficiency with which a product can be modified by the intended maintainers [1]. Software evolution and maintenance costs increase as systems become more complex and larger [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

The maintenance and evolution of systems are usually hindered by structural problems in the source code that were not properly addressed in early stages of the development process. These structural problems were called as code smells (CSs) by Kent Beck [3]. Software is not prevented from operating by CSs, but they indicate when fundamental design principles are ignored or design best practices are not used. As a result, they affect the maintainability, extend the time required to develop the software, and raise the risk of making mistakes in later stages of the software life cycle. The lack of knowledge of software architects and programmers about CSs is a factor that leads to these structural problems in the source code [4]. Hence, identifying and adopting approaches to improve the knowledge of software architects and programmers about CSs is essential to prevent anomalies in the source code.

Looking for alternatives to address the aforementioned issues, we found that the scientific community accepts the adoption of ontologies as a feasible alternative to manage knowledge in different domains [4], [5], [6], [7]. By using ontologies to represent semantically information, it is possible to check for consistency and use reasoners to automatically deduce new knowledge. Some ontology-based approaches have been developed to represent, analyze and share knowledge about CSs [5], [6], [7]. Hence, based on the benefits of adopting ontologies extensively reported in the literature, this paper aims to describe an ontology-based proposal to represent, analyze and share knowledge related to CSs. Especially, this ontology may be a valuable means to support the training of software architects and programmers. The information regarding CSs can be found in heterogeneous sources such as scientific articles [3], [8], websites, databases, etc. Consequently, this ontology will make it possible to examine the consistency of the information contained in diverse sources. In addition, the systematic usage of this ontology will allow the information represented to be extended and thus become a valuable knowledge base. This knowledge base could be exploited to support the work of those professionals involved in software development projects.

The structure of this paper is as follows. Section II analyzes the related works. In Section III some basic concepts regarding CSs and ontologies are presented. Section IV introduces an ontology to represent the concepts related to CSs. In Section V, the results of the ontology evaluation are analyzed. Finally, we present the conclusions and future works.

II. RELATED WORK

Numerous authors argue the advantages of discovering CSs at an early stage in software development. Paiva and others [9] present a study done on two Java-developed programs: MobileMedia and Health Watcher. God Class, God Method, and Feature Envy were the three types of CS that were examined using the JSpIRIT tool. On the other hand, research was conducted to assess six tools: iPlasma, Together, JDeodorant, PMD, DECOR [10]. SourceForge was used to analyze systems developed in Java. Additionally, Liu [11] evaluated the ability of the tools Checkstyle, Infusion, PMD, iPlasma, DECOR, and others to detect CSs. The features of SonarQube are described by Bastias and López [12], [13] for analyzing several quality attributes and identifying technological risks brought on by CSs [14].

The aforementioned methods show how hard scientists are working to address CSs-related problems. There are certain limitations, nonetheless, such as the fact that only a limited number of CSs are handled, CSs detection rate is low for various tools, or only particular CSs types are addressed.

The scientific community is interested in dealing with issues related to how to represent, analyze, and share knowledge about CSs. Adopting tools for communication and knowledge sharing regarding CSs is essential for educating software developers. Several authors argue the notable benefits of adopting ontology-based approaches to manage the knowledge related to CSs. An ontological representation, for instance, was used by Luo et al. [7] to formally express the ideas of anti-patterns, CSs, refactoring, and their interactions. Da Silva et al. [6] describes an ontology capable of representing the knowledge necessary to keep track of Code Smells and to evaluate their impact on software. An ontology that collects and organizes the knowledge about design in object-oriented micro-architectures is introduced by Garzás and Piattini [5]. They discovered several pertinent terms, including patterns, refactoring, bad smells, and good practices.

Although they only consider a small subset of CSs, these approaches demonstrate the pertinence of ontology-based proposals to manage the knowledge related to CSs. The effect of CSs on software quality is also not fully examined. Furthermore, there is a lack of empirical data showing how these methods affect the professional knowledge of those working on software development projects.

III. BACKGROUND

A. CODE SMELLS

In terms of reusability, understandability, and maintainability, CSs can reduce software quality. While a CS does not stop the software from working, it can delay its development or cause faults to appear in later stages of the process [15]. Because appropriate practices aren't followed, CSs reveal violations of key design principles, which have a detrimental effect on the quality of software [16].

Multiple authors have examined various aspects that lead to design and architectural gaps [17], [18]. For instance, having large classes and methods including many parameters are common issues that make the code hard to read and tighten up the coupling. The system maintainability is negatively impacted by these issues.

For information on state of the art in this field, we reviewed the literature on CSs [3], [4], [8], [10]. A list of CSs was created and categorized according to their granularity and similarity. We discovered that the CSs put forth by Lanza and Marinescu [8] as well as Fowler and others [3] are the most addressed by the scientific community.

Mäntylä and Lassenius [17], [18] classify CSs into groups based on how similar they are. Granularity or scope are factors considered to classify CSs. In that direction, the structural and lexical levels [12] are considered.

B. ONTOLOGY

In the domain of computing, an ontology is composed of classes (or concepts), properties (slots or roles), and restrictions on properties (or facets) [19]. An ontology and a collection of individual instances of classes make up a knowledge base. Studies of ontology-based approaches to information sharing and communication have been extensively published in the scientific literature [19], [20], [21], [22], [23].

Some relevant languages to represent ontologies are XML Schema, Ontolingua, RDF (Resource Description Framework), RDF Schema (RDF-S), OWL, and OWL 2. The fact that OWL 2 is founded on description logics makes it possible to use reasoners to automatically verify the consistency of the represented models. A wide range of operators, such as intersection, union, and negation, are included. Protégé, a popular tool for creating ontologies and serving as a broad foundation for knowledge representation, supports OWL [24]. Pellet is an effective reasoner that may be combined with Protégé to draw conclusions, examine formal logical properties, and verify the coherence of the ontology.

Adopting a sound methodology that guides the development process is essential to creating a high-quality ontology. Particularly, we adopted the guide recommended by Noy and McGuinness, which is a well-known methodology [25]. It is one of the methodologies for creating ontologies that is most frequently used or cited [26]. The methodology proposes the following steps: Determine the domain and scope of the ontology, consider reusing existing ontologies, enumerate important terms in the ontology, define the classes and the class hierarchy, define the properties of classes (called relationships or slots), define the facets and/or restrictions on slots and finally create instances.

IV. ONTOLOGICAL MODEL TO REPRESENT, ANALYZE AND SHARE KNOWLEDGE ABOUT CODE SMELLS

Following the steps defined in the methodology of Noy and McGuinness, an ontology to manage the knowledge related to CSs was obtained. The most important outcomes are described below.

A. DETERMINE THE DOMAIN AND SCOPE OF THE ONTOLOGY

This ontology aims to describe various concepts related to CSs, such as the main features of CSs, CSs classifications, potentially affected design principles or quality features which belong to different quality standards. In addition, some architectural anomalies that may be found in classes and methods of a system are described as well. Likewise, the metrics used by some tools to identify anomalies are included.

The ontology represents CSs knowledge, making it a helpful tool for training software specialists. Specially, those specialists responsible for the identification and solution of CSs. The following competency questions must be answered by the ontology:

Q1: What is the classification of a code smell?

Q2: What quality features are impacted by a code smell?

Q3: What metrics can be used to discover a specific code smell?

Q4: What tools help with the identification of a particular code smell?

B. CONSIDER REUSING EXISTING ONTOLOGIES

Reusing ontologies is a good practice to speed up the development of a new one. For example, the concepts Metric, Clazz, and Method were reused from the ontology ON-TOCEAN [6]. Classes like Bloaters, Encapsulators, and Dispensable proposed by Luo [7] were reused as well.

In addition, we searched in the repositories DAML [27] and DBpedia [28]. In these repositories, a wide number of ontologies are published. However, these ontologies are not focused on representing knowledge about CSs.

C. LIST THE RELEVANT TERMS OF THE ONTOLOGY

Based on the literature review, some relevant terms were identified. The identified concepts are shown in Table 1. We focused particularly on the quality standard ISO 25000. This standard outlines the quality characteristics and sub-characteristics of a software. Hence, these characteristics and sub-characteristics may be impacted by CSs. Consequently, specific ISO 25000 concepts are also incorporated into our ontology.

D. DEFINE THE CLASSES AND THE CLASS HIERARCHY

Classes, properties, restrictions, and instances are the main components of an ontology. Fig. 1 shows the class hierarchy where 22 classes were specified and represented in Protégé. According to the type of properties, a class in an ontology can be classified into defined or primitive. Classes with only necessary conditions are classified as Primitive, whilst defined classes have necessary and sufficient conditions [30]. Fig. 1 illustrates the class *Clazz* that subsumes five disjoint subclasses (highlighted in red box).

E. DEFINE THE PROPERTIES

Properties (object properties and datatype properties) are the other key component in an ontology. Object properties are relationships between two individuals. Hence, instances of different classes can be related by means of object properties. On the other hand, datatype properties describe relationships between an individual and data values. Therefore, to link an individual to an XML Schema Datatype value or a rdf literal, a datatype property is used [30].

We identified 38 object properties in our ontology. Table 2 shows some object properties as well as their domain and range. Whilst Fig. 2 depicts several relationships between them. For instance, the property *isClassificationOf* links

TABLE 1. Relevant concepts to be represented in the ontology.

Concept	Description
Type of code smell	It is a structural issue with a piece of software that requires reorganization to increase quality [6, 7].
Classifications of code smell	To classify a CS according to its features.
Code agglomeration	A collection of CSs that are connected to one another. For instance, method calls or inheritance [28] can be used to determine the relationship between two CSs. [29].
Architectural anomaly	Symptom of degradation in the architecture of a software because CSs are gradually added to it [29].
Quality standard	International standards for quality, such as ISO 25000 [13].
Quality characteristic	It establishes software product quality requirements [13]. Maintainability, for instance.
Quality sub-characteristic	Within the quality characteristics are quality sub-characteristics [13]. Reusability, for instance, is a component of maintainability.
Design element	Principles, good practices, or design patterns. For example, Coupling and Cohesion.
Tool	Technologies that support the identification of CSs.
Metric	Metrics that are applied by the CSs identification tools.
System	Software that might contain architectural flaws as a result of CSs.
Component	A piece of code that wraps a system functionality and is available through common interfaces.
Class	It is used to describe one or more objects in object-oriented programming. It acts as a template for creating particular objects inside of a program.
Method	In object-oriented programming, a function or procedure.
Programming language	A formal language that contains rules to create sequences that direct the physical and logical actions of computers to generate different types of data.

individuals of the classes *ClassificationTypeCodeSmell* and *TypeCodeSmell*; the property *affectsQualityCharacteristic* links individuals of the classes *TypeCodeSmell* and *Quality-Characteristic*; whilst the property *canBeIdentifiedByMetric* links individuals of the classes *TypeCodeSmell* and *Metric*. similarly, the property *canBeIdentifiedByTool* links individuals of the classees *TypeCodeSmell* and *Tool*.

In an ontology, there is an inverse property for each object property. It means that if a property relates individual A to individual B, then its inverse property relates individual B to individual A. Fig. 2 depicts that *isClassifiedBy* is the inverse property of *isClassificationOf*, *isQuality-CharacteristicAffectedBy* is the inverse of *affectsQuality-Characteristic*, *allowIdentifyCodeSmell* is the inverse of *canBeIdentifiedByMetric*, and *helpsToIdentifyCodeSmell* is the inverse of *canBeIdentifiedByTool*.

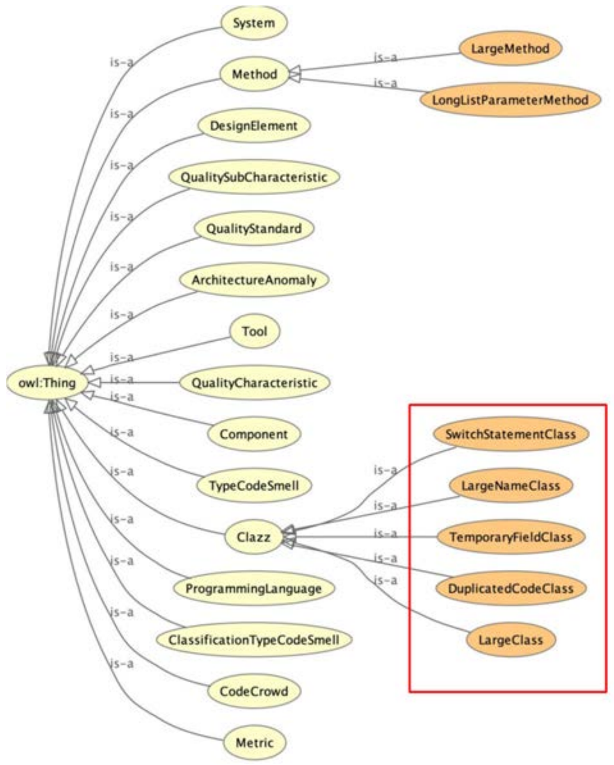


FIGURE 1. Classes of the ontology to represent code smells.

TABLE 2. Set of object properties with domain and range.

Domain	Property	Range
TypeCodeSmell	*affectsDesignElement*	DesignElement
	affectsQualityCharacteristic	QualityCharacteristic
	affectsComponent	Component
ClassificationTypeCodeSmell	*isClassificationOf*	TypeCodeSmell
QualityStandard	*containsQualityCharacteristic*	QualityCharacteristic
QualityCharacteristic	*containsSubQualityCharacteristic*	QualitySubCharacteristic
Tool	*identifyCodeSmell*	TypeCodeSmell
	supportsLanguage	ProgrammingLanguage
	Uses	Metric
Metric	*allowIdentifyCodeSmell*	TypeCodeSmell
Component	*hasMethod*	Method
	hasClass	Class
System	*HasComponent*	Component
	isDevelopWithLanguage	ProgrammingLanguage
Class	*hasMethod*	Method

Some datatype properties were specified in the ontology to complete the concept descriptions. The domain and range of these properties are depicted in Table 3.

F. DEFINE FACETS AND/OR RE-STRICTIONS ON SLOTS

The expressivity richness of OWL allows defining different types of restrictions. For example, cardinal restrictions are

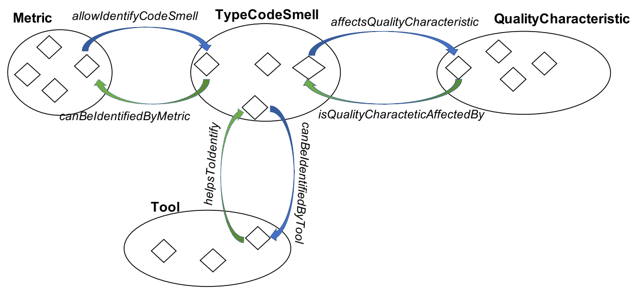
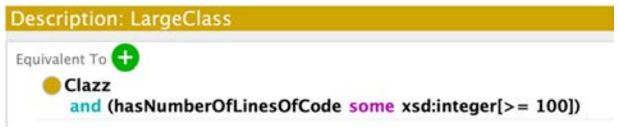


FIGURE 2. A sample of object properties.

TABLE 3. Data properties included in the ontology.

Domain	Data property	Range
Metric	*metricCharacteristic*	String
	isMetricScope	String
Method	*hasNumberOfLinesOfCode*	Integer
	hasNumberOfParameter	Integer
	hasTemporaryField	Boolean
Clazz	*hasSwitchStatement*	Boolean
	hasDuplicatedCode	Boolean
	hasNumberOfMethod	Integer
TypeCodeSmell	*typeOfGranularity*	String
Clazz	*className*	String
Method	*methodName*	String
System	*systemName*	String
	descripction	String
	developmentCenter	String

used to specify the number of relationships that an individual may participate in for a given property. Fig. 3 shows a cardinal restriction to classify the instances of the class **Large_Class**. Based on this statement, a reasoner will classify automatically as **Large_Class** the instances of the class Clazz with more than 100 code lines.


Description: LargeClass

Equivalent To

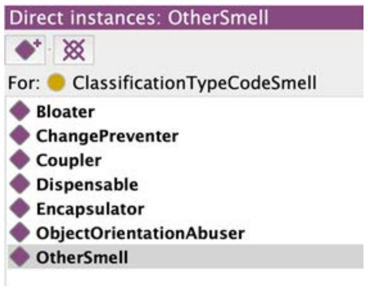
Clazz
and (hasNumberOfLinesOfCode some xsd:integer[>= 100])

FIGURE 3. A cardinal restriction for the class Large_Class.

G. DEFINE INSTANCES

To illustrate the ontology applicability, we used it to specify the knowledge related to CSs in a project to develop a software. Fig. 4 and Appendix depict instances of the classes *TypeCodeSmell* and *ClasificationTypeCodeSmell,* respectively.

We defined some concepts associated with a software development project to illustrate how the ontology could be used to represent CSs. Fig. 4 shows instances of the class *TypeCodeSmell.* Whilst Appendix shows instances of the class *ClasificationTypeCodeSmell*. This version of the ontology includes 96 individuals.


Direct instances: OtherSmell

For: ClassificationTypeCodeSmell

- Bloater
- ChangePreventer
- Coupler
- Dispensable
- Encapsulator
- ObjectOrientationAbuser
- OtherSmell

FIGURE 4. Instances of the class *ClasificationTypeCodeSmell.*

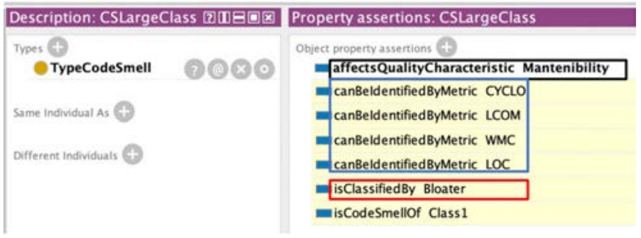
V. RESULTS OF THE EVALUATION PROCESS

The two main steps to evaluate an ontology are: checking the consistency and verifying that it meets the defined requirements. The consistency checking enables confirming that the ontology is free of contradictions. By using a reasoner, it is possible to evaluate the consistency and classify the individuals automatically. Specifically, reasoner Pellet was used throughout the development process of our ontology. Thus, we demonstrated that the ontology satisfies the requirements for a formal system.

To verify that the ontology meets the defined requirements, we evaluated how the competence questions are answered. To illustrate how the competence questions are answered, the ontology was populated, and the reasoner Pellet was applied. The results of this evaluation are detailed below:

Q1: ¿What is the classification of a code smell?

First of all, we selected in Protégé the individual *CSLargeClass,* which is an instance of the class **TypeCodeSmell**. The value of the object property *isClassifiedBy* in Fig. 5 illustrates how the reasoner classified the selected individual as a **Bloater** (red box).


Description: CSLargeClass

Types
TypeCodeSmell

Same Individual As

Different Individuals

Property assertions: CSLargeClass

Object property assertions

affectsQualityCharacteristic Mantenibility
canBeIdentifiedByMetric CYCLO
canBeIdentifiedByMetric LCOM
canBeIdentifiedByMetric WMC
canBeIdentifiedByMetric LOC
isClassifiedBy Bloater
isCodeSmellOf Class1

FIGURE 5. Inferences for CSLargeClass, an instance of the class TypeCodeSmell.

Q2: What quality features are impacted by a code smell?

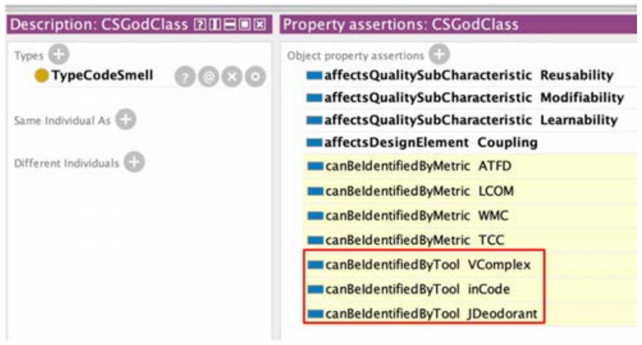
When a **TypeCodeSmell** is chosen, numerous quality characteristics may be associated by means of the property *affectsQualityCharacteristic* because a CS may affect one or more quality characteristics. Fig. 5 illustrates how the specific **CSLargeClass**, a subclass of **TypeCodeSmell**, influences the quality characteristic **Maintainability** (object property *affectsQualityCharacteristic*) (black box).

Q3: What metrics can be used to discover a specific code smell?

This information can be obtained by choosing an instance of the **TypeCodeSmell** class and checking the values of the property *canBeIdentifiedByMetric*. The individual **CSLargeClass** can be identified by metrics such as CYCLO, LCOM, WMC, and LOC, as shown in Fig. 5 (highlighted in the blue box).

P4: ¿What tools can be used to identify a specific code smell?

The value of the property *canBeIdentifiedByTool* will indicate the tools that can identify a specific CS. For example, Fig. 6 (highlighted in red box) shows that ***CSGodClass*** *canBeIdentifiedByTool JDeodorant, VComplex,* and *inCode*.



TypeCodeSmell

affectsQualitySubCharacteristic Reusability
affectsQualitySubCharacteristic Modifiability
affectsQualitySubCharacteristic Learnability
affectsDesignElement Coupling
canBeIdentifiedByMetric ATFD
canBeIdentifiedByMetric LCOM
canBeIdentifiedByMetric WMC
canBeIdentifiedByMetric TCC
canBeIdentifiedByTool VComplex
canBeIdentifiedByTool inCode
canBeIdentifiedByTool JDeodorant

FIGURE 6. Inferences for CSGodClass, an instance of the class TypeCodeSmell.

VI. ASSESSING THE IMPACT OF THE ONTOLOGICAL MODEL

To demonstrate with empirical evidences the impact of our proposal, we conducted an experiment. Specifically, we are interested in assessing whether the application of this ontological approach speeds up the analysis of knowledge related to CSs. We conducted a quasi-experiment with ten software architects and developers who have been involved in different software projects. We measured the time that the participants needed to analyze information regarding CSs. Firstly, they used different means to find information about CS. Secondly, the participants used our approach.

To carry out this experiment, we used the competence questions that were defined to specify the scope of the ontology:

Q1: What is the classification of a code smell?

Q2: What quality features are impacted by a code smell?

Q3: What metrics can be used to discover a specific code smell?

Q4: What tools help with the identification of a particular code smell?

Specifically, to answer these questions, we selected the CS *CSLargeClass* and the tool *VComplex*. Considering that the participants have different levels of knowledge about CSs, they searched for information from different sources such as scientific literature, work reports, databases, etc. Our preliminary hypothesis was that the application of our approach might reduce by 10 times the time necessary to analyze the knowledge regarding CSs. The second column in Table 4 shows the time that spent each participant to answer the questions without using our approach. Whilst the third column shows the time that spent each participant to answer the questions using our approach. The last row shows the average time for each observation. In this row, it is shown that for the first observation, the average time was 778 seconds whilst for the second observation, the average time was 68 seconds. Hence, the time by using our approach was reduced by 11.44 times. These statistics demonstrated our preliminary hypothesis. Thus, we have quantitative evidence that demonstrates the positive impact of our proposal. Specially, this ontological model can be a valuable means to support the training of software developers. Additionally, we are designing new experiments to assess the impact of our approach to enhance other variables, for example, the time to detect and fix CSs.

TABLE 4. Results of the experiment.

Participant	Observation 1 Time (seconds)	Observation 2 Time (seconds)
A1	910	94
A2	740	45
A3	613	58
A4	954	61
A5	970	64
A6	630	50
A7	765	77
A8	610	53
A9	850	88
A10	738	90
Average	**778**	**68**

VII. CONCLUSION AND FUTURE WORK

This paper presented an ontology-based approach to support the representation and analysis of the knowledge related to CSs. This approach addresses some of the gaps that were found in the related work review. Likewise, the literature review helped identify some types of CSs, tools for detecting detection, and ontologies developed to represent knowledge related to CSs. The development of the ontology was guided by a well-known methodology which contributed to ensuring its quality. The ontology was represented using OWL. Since this is a formal language based on description logics, it enables the automatic consistency checking of the knowledge. In addition, this ontology can be analyzed by a reasoner to infer new knowledge.

We conducted an experiment to demonstrate the benefits of applying our approach. The results of the experiment indicate that using this approach makes it possible to analyze knowledge related to CSs 10 times faster than by other means. In addition, this ontology covers not only specific information regarding CSs, but includes information related to software projects and other concepts related to software quality. Therefore, the analysis of the knowledge can be improved in terms of completeness and quality. As a result, this ontology may be a helpful tool to aid in the education of software architects and programmers because it includes a wide range of CS-related and other pertinent concepts.

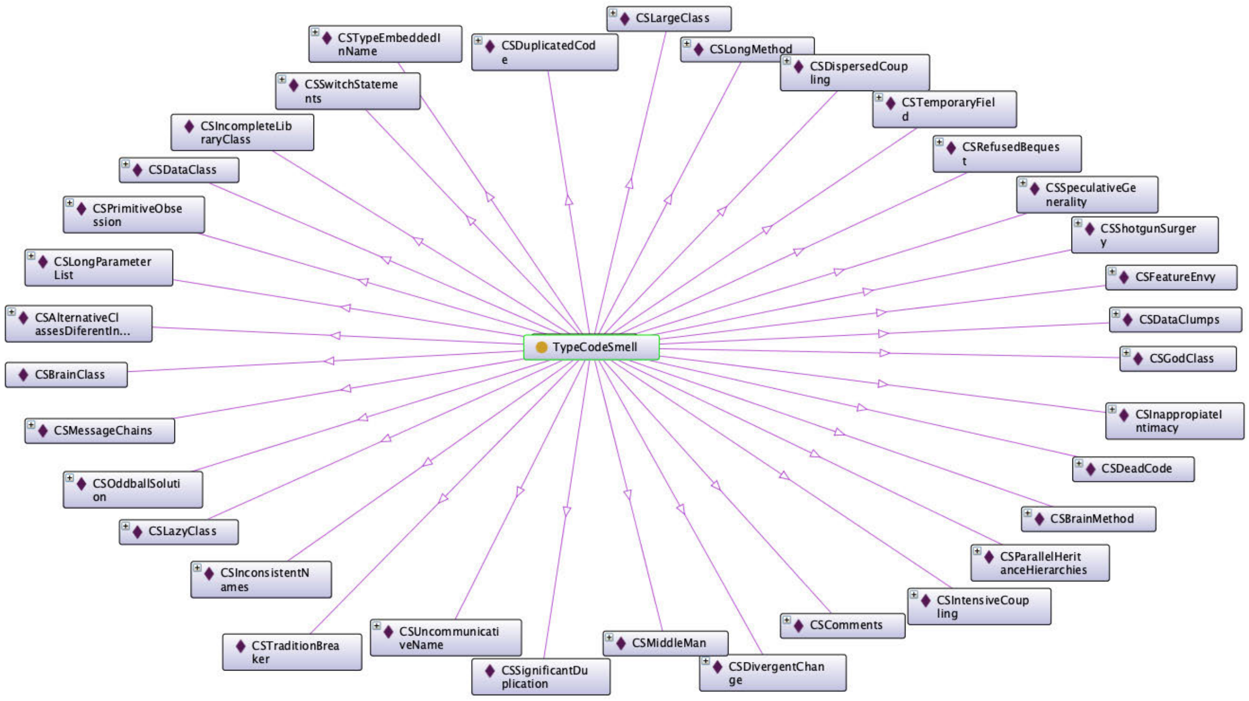


FIGURE 7. Instances of the class TypeCodeSmell.

We focus the future work on extending the ontology by including new concepts about the software development life cycle and their relation to CSs. In addition, the systematic application of this ontology in real environments will contribute to expanding the information that it represents as well as identifying new opportunities to improve it.

APPENDIX
INSTANCES OF THE CLASS TypeCodeSmell

See Fig. 7.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Princess Nourah bint Abdulrahman University Researchers Supporting Project number (PNURSP2023R407), Princess Nourah bint Abdulrahman University, Riyadh, Saudi Arabia.

REFERENCES

[1] J. Estdale and E. Georgiadou, "Applying the ISO/IEC 25010 quality models to software product," in *Systems, Software and Services Process Improvement*. Cham, Switzerland: Springer, 2018.

[2] A. April and A. Abran, *Software Maintenance Management: Evaluation and Continuous Improvement*, vol. 67. Hoboken, NJ, USA: Wiley, 2012.

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

[4] A. Malavolta, "Análisis de detección de Code Smells para el lenguaje JavaScript," M.S. thesis, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Buenos Aires, Argentina, 2018.

[5] J. Garzas and M. Piattini, "An ontology for microarchitectural design knowledge," *IEEE Softw.*, vol. 22, no. 2, pp. 28–33, Mar. 2005.

[6] L. P. da Silva Carvalho, R. Novais, L. do Nascimento Salvador, and M. G. de Mendonça Neto, "An ontology-based approach to analyzing the occurrence of code smells in software," in *Proc. 19th Int. Conf. Enterprise Inf. Syst.*, 2017, pp. 155–165.

[7] Y. Luo, A. Hoss, and D. L. Carver, "An ontological identification of relationships between anti-patterns and code smells," in *Proc. IEEE Aerosp. Conf.*, Mar. 2010, pp. 1–10.

[8] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Berlin, Germany: Springer, 2007.

[9] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *J. Softw. Eng. Res. Develop.*, vol. 5, no. 1, p. 7, 2017.

[10] K. Alkharabsheh, Y. Crespo, J. Taboada, and E. Manso, "Comparación de herramientas de Detección de design smells," in *Proc. JISBD*, 2016, pp. 159–172.

[11] X. Liu and C. Zhang, "The detection of code smell on software development: A mapping study," in *Proc. 5th Int. Conf. Machinery, Mater. Comput. Technol.*, 2017, pp. 560–575.

[12] O. A. Bastias, "Código con 'mal olor': Un mapeo sistemático," *Revista Cubana de Ciencias Informáticas*, vol. 12, no. 4, pp. 156–176, 2018.

[13] J. V. López, "Auditoría mantenibilidad aplicaciones según la ISO/IEC 25000," M.S. thesis, Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain, 2015.

[14] J. P. O. Delgado, "Análisis de seguridad y calidad de aplicaciones (Sonarqube)," M.S. thesis, Departamento de Informática, Universidad Obterta de Cataluyna, Manizales, Colombia, 2015.

[15] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 306–315.

[16] K. AlKharabsheh, Y. Crespo, E. Manso, and J. Taboada, "Sobre el grado de acuerdo entre evaluadores en la detección de Design Smells," in *Proc. Jornadas de Ingeniería del Software y Bases de Datos, (JISBD)*, 2016, pp. 143–157.

[17] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Softw. Eng.*, vol. 11, no. 3, pp. 395–431, Jul. 2007.

[18] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proc. Int. Conf. Softw. Maintenance (ICSM)*, 2003, pp. 381–384.

[19] N. F. Noy and D. L. McGuinness, "Ontology development 101: A guide to creating your first ontology," Stanford Knowl. Syst. Lab., Tech. Rep. KSL-01-05, Stanford Med. Inform., Stanford, CA, USA, Tech. Rep. SMI-2001-0880, 2001.

[20] J. T. F. Breis, "Un entorno de integración de ontologías para el desarrollo de sistemas de gestión del conocimiento," Ph.D. thesis, Departamento de Ingeniería de la Información y las Comunicaciones, Universidad de Murcia, Murcia, España, 2003.

[21] C. Welty and N. Guarino, "Supporting ontological analysis of taxonomic relationships," *Data Knowl. Eng.*, vol. 39, no. 1, pp. 51–74, Oct. 2001.

[22] E. M. Beniaminov, "Ontology libraries on the web: Status and prospects," *Autom. Documentation Math. Linguistics*, vol. 52, no. 3, pp. 117–120, May 2018.

[23] J. D. S. Benjumea, "Ontologías para conceptualización de modelos de negocio," M.S. thesis, Facultad de Ingenierías, Universidad de Medellín, Medellín, Colombia, 2013.

[24] A. F. Hernández, "Modelo ontológico de recuperación de información para la toma de decisiones en gestión de proyectos," Ph.D. thesis, Departamento de Información y Comunicación, Universidad de Granada, Granada, España, 2016.

[25] M. C. Suárez-Figueroa, A. Gómez-Pérez, E. Motta, and A. Gangemi, "Ontology engineering in a networked world," in *Ontology Engineering in a Networked World*. Berlin, Germany: Springer, 2012, pp. 1–435.

[26] A. Sattar, E. Salwana, M. Nazir, M. Ahmad, and A. Kamil, "Comparative analysis of methodologies for domain ontology development: A systematic review," *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 5, pp. 99–108, 2020.

[27] DAML. (2004). *DAML Ontology Library*. Accessed: Jun. 1, 2023. [Online]. Available: http://www.daml.org/ontologies

[28] DBpedia. (2019). *D.M. Contributors*. Accessed: Jun. 1, 2023. [Online]. Available: http://mappings.dbpedia.org/index.php?title=Main_Page&oldid=53546

[29] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *J. Softw. Eng. Res. Develop.*, vol. 3, no. 1, p. 11, Dec. 2015.

[30] M. Horridge, S. Jupp, G. Moulton, A. Rector, R. Stevens, and C. Wroe, "A practical guide to building owl ontologies using protégé 4 and co-ode tools edition 1. 2," Univ. Manchester, Manchester, U.K., Tech. Rep., 2009, p. 107. [Online]. Available: http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf

IVIAN L. CASTELLANO received the degree and M.Sc. degrees from the University of Informatics Sciences, Habana, Cuba, in 2012 and 2021, respectively. She is currently pursuing the master's degree in economics, finances and computing with the International University of Andalusia, Huelva, Spain. She has published five papers in international conferences. Her research interests include software quality, ontology-driven engineering, and code smells.

GILBERTO FERNANDO CASTRO AGUILAR received the bachelor's degree from Universidad Católica de Santiago de Guayaquil (UCSG), the master's degree from Universidad de Guayaquil (UG), and the Ph.D. degree from the University of Informatics Sciences (UCI). He is a Principal Professor of the Faculty of Sciences Mathematics and Physics, UG, and the Faculty of Engineering, UCSG. He has been the Manager of revenue assurance projects with National Telecommunications Corporation (CNT EP), Ecuador. He was a Revenue Assurance Certified in Herzliya-Israel, for research in computer engineering, software technologies, and data mining applications to help in decision making. He has publications in magazines, presentations, and events of high national and international impact. He is a Tutor-Mentor of regional and Latin American innovation projects. His research interests include basic sciences, bio-knowledge, and industrial development. He is a member of the body of referees of academic publications, the scientific committee, and the international networks of projects and technology, RISEI and RIIPRO. He is also a member of the editorial board of UCSG, from 2022 to 2027; a Volunteer Mentor of the program "Sin Fronteras" Latin America, in 2022; a peer evaluator of the evaluation process for accreditation purposes of Universities and Polytechnic Schools of Ecuador; a member of the relevance project, educational model, and accompaniment for institutional accreditation of UG, in 2023; and a delegate alternate to the Superior University Council, UG.

NEMURY SILEGA received the degree and Ph.D. degrees from the University of Informatics Sciences, Habana, Cuba, in 2007 and 2014, respectively. Currently, he is a Researcher Leader with Southern Federal University, Taganrog, Russia. He has published more than 50 papers in international conferences and more than 20 in journals. His current research interests include model-driven development, ontology-driven engineering, and business process modeling. In 2015, he received the Erasmus Mundus Scholarship for a postdoctoral stay with the University of Granada.

TAHIR KAMAL received the degree from the University of Agriculture Faisalabad, Pakistan, and the Ph.D. degree from the Chongqing University of Post and Telecommunications, China, in 2021. Currently, he is a Lecturer with Zhejiang Normal University, China. He has published more than ten papers in international conferences and journals. His current research interests include model-driven development, ontology-driven engineering, business process modeling, agile development, software engineering, and machine learning.

MEHDHAR S. A. M. AL-GAASHANI (Graduate Student Member, IEEE) received the B.Sc. degree from Belgorod State Technological University (BSTU), Russia, and the M.Sc. degree from Don State Technical University, Russia. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing, China. His research interests include machine learning, image processing, and the IoT.

NAGWAN ABDEL SAMEE received the B.S. degree in computer engineering from Ein Shams University, Egypt, in 2000, and the M.S. degree in computer engineering and the Ph.D. degree in systems and biomedical engineering from Cairo University, Egypt, in 2008 and 2012, respectively. Since 2013, she has been an Assistant Professor with the Information Technology Department, CCIS, Princess Nourah bint Abdulrahman University, Riyadh, Saudi Arabia. Her research interests include data science, machine learning, bioinformatics, and parallel computing. Her awards and honors include the Takafull Prize (Innovation Project Track), the Princess Nourah Award in Innovation, the Mastery Award in Predictive Analytics (IBM), the Mastery Award in Big Data (IBM), and the Mastery Award in Cloud Computing (IBM).

MAALI ALABDULHAFITH was born in Saudi Arabia, in September 1985. She received the Ph.D. degree in computer science from Dalhousie University, Halifax, Canada, in 2017. In 2014, she joined the College of Computer and Information Science (CCIS), Princess Noura University (PNU), as a Lecturer and was promoted to an Assistant Professor, in 2018. Currently, she is the Director of data management and performance measurement with CCIS, overlooking and managing the strategy of the college. Her research interests include machine learning, data analytics, emerging wireless technology, and technology applications in health care.