An Ontology-based Approach to Analyzing the Occurrence of Code Smells in Software

Luis Paulo da Silva Carvalho[1,3], Renato Novais[2,4], Laís do Nascimento Salvador[3,4] and Manoel Gomes de Mendonça Neto[3,4]

[1] *Federal Institute of Bahia, Av. Amazonas 3150 - Zabelê, Vitória da Conquista, Brazil*

[2] *Federal Institute of Bahia, Rua Emídio dos Santos, s/n, Salvador, Brazil*

[3] *Federal University of Bahia, Av. Ademar de Barros, s/n - Campus de Ondina, Salvador, Brazil*

[4] *Fraunhofer Project Center for Software and Systems Engineering at UFBA, Av. Paralela, Salvador, Brazil*

Keywords: Ontology, Reasoner, Code Smells, Ontocean.

Abstract: Code Smells indicate potential flaws in software design that can lead to costly consequences. To mitigate the bad effects of Code Smells, it is necessary to detect and fix defective code. Programmatic processing of Code Smells is not new. Previous works have focused on detection and representation to support the analysis of faulty software. However, such works are based on a syntactic operation, without taking advantage on semantic properties of the software. On the other hand, there are several ways to provide semantic support in software development as a whole. Ontologies, for example, have recently been usedl. The application of ontologies for inferring semantic mechanisms to aid software engineers in dealing with smells may be of great value. As little attention has been given to this, we propose an ontology-based approach to analyze the occurrence of Code Smells in software projects. First, we present a comprehensive ontology that is capable of representing Code Smells and their association with software projects. We also introduce a tool that can manipulate our ontology in order to provide processing of Code Smells as it mines software source-code. Finally, we conducted an initial evaluation of our approach in a real usage scenario with two large open-source software repositories.

1 INTRODUCTION

Code Smells are defined as metaphors that describe patterns associated with bad design and bad programming practices (Van Emden and Moonen, 2002). They can lead to issues in software maintenance (Olbrich et al., 2010). One well-known example of Code Smell is *God Class*, which is related to poorly designed classes that tend to concentrate functionalities and relegate other classes to minor roles (Smith and Williams, 2000). Common negative impacts of Code Smells include (Tufano et al., 2015): (a) increasing in both software change and fault proneness; and (b) reduced understandability and maintainability of the software. Thus, it is important to investigate the occurrence of Code Smells to prevent their bad effects in software development.

Previous works in this area can be divided into two main branches: (a) detection and visualization of Code Smells, *e.g.*, (Moha et al., 2010)(Fenske et al., 2015), which rely on metrics obtained from the software's source-code to spot occurrences of Code Smells; and (b) analysis of the effects of Code Smells, *e.g.*, (Chatzigeorgiou and Manakos, 2014)(Palomba et al., 2014), which perform a statistical analysis of the occurrence of Code Smells and provide insights about their consequences in software development.

Related works are based on syntactic operations without taking advantage of the semantic properties of the software. They show that previous efforts have given little attention to the association of semantics with information obtained from the detection of Code Smells. On the other hand, there have been examples to provide semantic support in software. Ontologies have been used as an effective tool to this end, as can be seen in recent works, *e.g.*, (Civili et al., 2013), (Sivaraman, 2014), and (Daraio et al., 2016). Approaches to detection, visualization and analysis of Code Smells may be enriched by the use of ontologies along with inferring semantic mechanisms to aid software engineers in dealing with defective code.

Carvalho, L., Novais, R., Salvador, L. and Neto, M.
An Ontology-based Approach to Analyzing the Occurrence of Code Smells in Software.
DOI: 10.5220/0006359901550165
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 2*, pages 155-165
ISBN: 978-989-758-248-6

Rules and reasoners are key features related to ontologies that can produce semantic information related to Code Smells. The association of such features with detection tools, *e.g.*, tools that detect Code Smells, can help software engineers to manage their occurrences. For instance, a reasoner can be used to reveal developers who add Code Smells to a piece of software. Once identified, the developers can be trained in object-oriented programming techniques to prevent further additions of smells to software. Furthermore, ontologies are able to incorporate other conceptualizations in such a way as to allow the expansion of the initially intended analysis (according to principles related to the open-world assumption (Djurić et al., 2005)). For example, in this work we conducted a case study into two types of Code Smells (God Classes and Brain Methods), however the resulting ontology can be expanded, later on, to add other types of smells.

In this context, we propose an ontology-based approach to support the detection and analysis of Code Smells in software projects. Our **ONTO**logy for **C**ode sm**E**lls **AN**alysis (ONTOCEAN) is capable of representing the knowledge necessary to keep track of Code Smells and to evaluate their impact on software.

We have associated our ontology with a tool: **O**ntology-driven **C**ode sm**E**lls **AN**alyzer (OCEAN). OCEAN is a source-code mining automation designed to benefit from the information that Code Smells can offer. OCEAN is integrated with Visminer (Mendes et al., 2015), which is an API that furnishes a set of functionalities that are capable of performing the automated calculation of software metrics and detection of Code Smells. As a proof of concept, we also conducted a study to demonstrate the execution of OCEAN to mine large open-source software repositories: JUnit and Log4j. The intention is to showcase OCEAN's functionalities and the use of mined data to populate instances of our ontology with the purpose of enabling the inferrence of information related to anomalous code artifacts.

As a result we were able to: (a) produce an ontology, ONTOCEAN, suited to represent information related to Code Smells and to provide ways to detect their presence and evaluate their impact by activating reasoners upon semantic rules; (b) make available a new tool, OCEAN, that is capable of populating ONTOCEAN with information extracted from software projects.

The remainder of this paper is structured as follows: Section 2 introduces our ontology, ONTOCEAN, and all of its conceptualizations; Section 4 explains how our tool, OCEAN, was created and how it is integrated with ONTOCEAN; Section 5 describes the tests executed to evaluate the applicability of ONTOCEAN; conclusions and future work are discussed in Section 7.

2 ONTOLOGICAL REPRESENTATION OF CODE SMELLS

One of the most widely known definition of what ontologies means is provided by (Gruber, 1993): "an ontology is an explicit specification of a conceptualisation". 'Conceptualisation' represents an abstract model of some aspect of the world and takes the form of a collection of concepts and their respective attributes and the relationships between the concepts. Ontologies have attracted attention in several areas of expertise, such as knowledge representation and management, information integration and semantic web. To be effectively applied, ontologies must be combined with reasoners. Reasoners ensure the quality of the ontology and are important to exploit the structures encapsulated in it (Staab and Studer, 2013). Our work is consistent with this view because it not only presents an ontology but it also makes use of a reasoner to expand the capacity of our ontology to generate semantic knowledge related to Code Smells.

(Chandrasekaran et al., 1999) pointed out some advantages of the use of ontologies: (a) Ontologies are fitting resources to represent the vocabulary of a domain of knowledge - as we have modeled Code Smells as an ontology we are providing a vocabulary related to bad coding; (b) Ontologies enable knowledge sharing - the resulting vocabulary can be (re)used by other works to understand phenomenon associated with bad software development techniques; (c) Ontologies contribute to the expansion of knowledge - this can be the case with a Code Smells ontology as it is used to base other ontologies which add concepts to the vocabulary we have provided. We are positive that the relationship between ontologies and Code Smells can offer these advantages.

According to the definition of ontology and its application in the representation of concepts related to Code Smells, we have ranked a set of requirements, *i.e.*, as we want our solution to have the aforementioned advantages we decided to transform them into requirements. Thus, in order to evaluate our work we managed to fulfil:

- **REQ1** – the resulting ontology must contain conceptualisations to represent Code Smells and items of object-oriented programming which the smells are related to;

- **REQ2** – the ontology has to be furnished with semantic rules to support the inferring of information associated with Code Smells;
- **REQ3** – it must be compatible with reasoners so that they can be invoked to produce the inferred information.

These requirements were used to evaluate related works and our own.

REQ1 is fulfilled by the items found in Figure 1 which illustrates the taxonomy of our proposed ontology, ONTOCEAN. 'A' points to VCS-related information; 'B' encloses Code Smells and sub-types: Method-oriented and Class-oriented Code Smells; 'C' contains classes to represent Metrics and its subtypes: Method-oriented and Class-oriented Metrics; and 'D' points to object-oriented Classes and Methods to be associated with the calculated metrics and detected Code Smells[1].

The implementation of requirements **REQ2** and **REQ3** are explained in the sections bellow.

An important part of the ontology is the set of Code Smells-related conceptualizations. They are used to represent the anomalies found in the source-code. The 'Commit' class, for instance, is semantically associated with the 'Clazz' class to create lists of classes which are affected by commits. A 'ClassOrientedSmell' instance can be associated with a 'Clazz' to indicate the occurrence of a class-like Code Smell (*e.g.*, 'GodClass') found in the source-code. Associations (*i.e.*, object properties relating the classes of the ontology) can be created to represent the software metrics (instances of classes inherited from the 'Metric' class) which take part in the production rules used to detect Code Smells. We have also added classes to enclose the projects' metadata: a list of commits ('Commit') and a committer ('Committer') and the repository ('Repository') from which the project is mined.

ONTOCEAN's classes are obtained from three main sources: (a) the mined data found in project's VCS; (b) calculated by OCEAN after processing the mined data; (c) the inferred information produced by reasoners. Table 1 describes the classes and groups them according to the way how they are created/produced.

It is also relevant to point out that the set of conceptualizations we have embedded in the ontology is not exhaustive; *i.e.*, despite the fact that it suffices for OCEAN, it can be expanded to include extra sets of metrics and Code Smells.

[1]An expanded version of ONTOCEAN classes and their relationships can be found at the end of the article

3 SEMANTIC INFERRING OF CODE SMELLS

In this section we explain how we have embedded semantic rules into ONTOCEAN with the purpose of enabling the retrieval of Code Smell-related information by inferring routines. By doing so, we were able to fulfil requirement **REQ2**. The intention behind inserting Code Smells' production rules into ONTOCEAN is to enable the ontology to point out instances of defective code with the help of no other tool than a reasoner. Thus, the ontology is self-sufficient in terms of automatic retrieval of Code Smells.

Code Smells can be obtained by detection strategies. For instance, in order to mine God Classes, such strategies can test candidate classes against the rule (Lanza and Marinescu, 2010)(Olbrich et al., 2010),

$$GC(C) = \begin{cases} 1, & ((WMC(C) > 47) \\ & \wedge(TCC(C) < 0.3)) \\ & \wedge(ATFD(C) > 5), \\ 0, & otherwise, \end{cases} \tag{1}$$

where C is the class being inspected; WMC represents the 'Weighted Method Count', which is the sum of the CC ('Cyclomatic Complexity') (McCabe, 1976) of all methods contained in C; TCC stands for 'Tight Class Cohesion' and counts the number of directly connected methods of C; and ATFD, 'Access to Foreign Data', is the number of attributes of foreign classes accessed by C. WMC, TCC and ATFD are software metrics that can be used by detection tools to find God Classes. With the purpose of replicating the rule we have added axioms to ONTOCEAN. The result is shown in Figure 2. 'A' represents the top ontological class, Clazz, indicating that only the individuals that are classes (object-oriented classes) must be considered. 'B' restricts the measured value of the WMC metric to greater than 47. 'C' constrains the TCC metric to values lower than 0.3. 'D' limits the ATFD metrics to values greater than 5.

ONTOCEAN is able to represent method-oriented Code Smells as well. The Brain Method Code Smell is applicable to methods (of classes) that concentrate logic, a characteristic that must be avoided since they tend to increase the complexity of software (Fowler, 1997). The occurrences of Brain Method can be detected by applying the rule (Lanza and Marinescu, 2010),

$$BM(M) = \begin{cases} 1, & ((MLOC(M) > 65) \\ & \wedge(CC(M) > 0.24) \\ & \wedge(MN(M) > 5) \\ & \wedge(NOAV(M) > 8)) \\ 0, & otherwise, \end{cases} \tag{2}$$

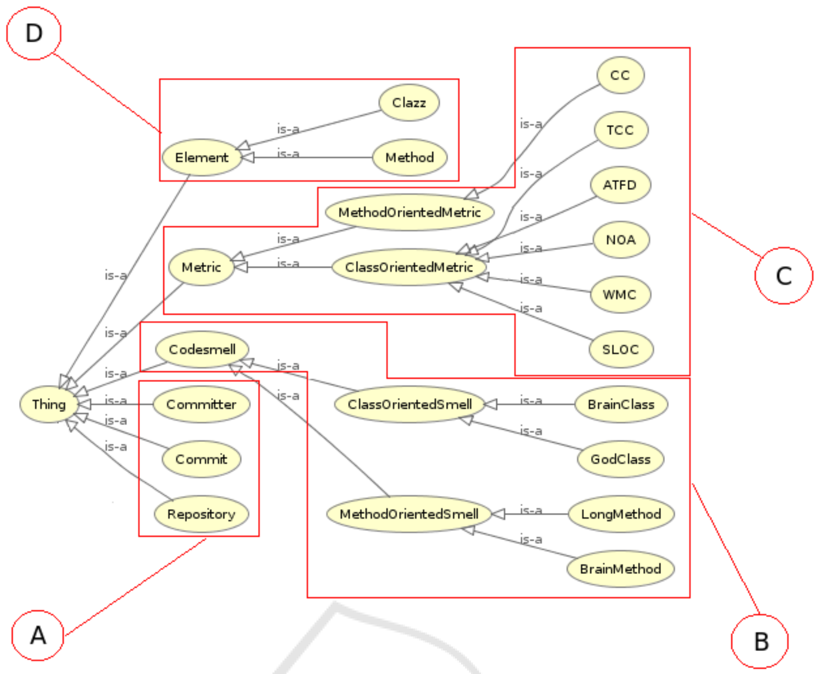


Figure 1: The taxonomy of ONTOCEAN.

Table 1: Origins of ONTOCEAN's conceptualizations.

Origin	Class	Sub-classes	Description
From VCS	Repository		Software's GIT repository
	Commit		Set of modifications on software's artefacts
	Committer		Developer that pushes commits to the repository
	Clazz		Object-oriented classes from source-code
	Method		Methods contained in classes
Calculated	Metrics	ClassOrientedMetric	Metrics that apply to object-oriented classes
		MethodOrientedMetric	Metrics that apply to methods of classes
From reasoners	Code Smell	ClassOrientedSmell	Smells embedded in object-oriented classes
		MethodOrientedSmell	Smells embedded in methods of classes
	Propagators		Committers who introduce smells in software

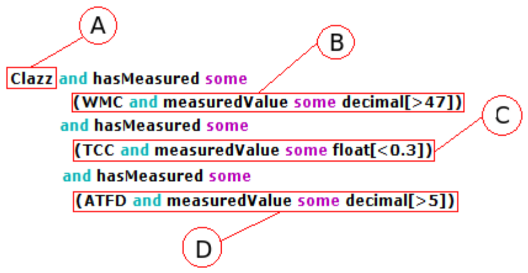
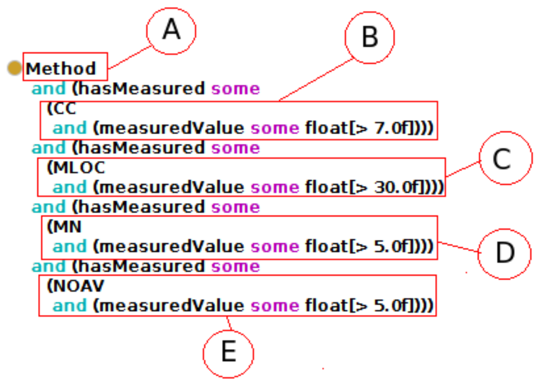

```
Clazz and hasMeasured some
    (WMC and measuredValue some decimal[>47])
    and hasMeasured some
    (TCC and measuredValue some float[<0.3])
    and hasMeasured some
    (ATFD and measuredValue some decimal[>5])
```

Figure 2: Translating the GC detection rule as an ontological axiom.

where M is the method being evaluated; MLOC represents 'Method's Lines Of Code', which is the number of lines of code in the method's body; CC counts the number of all the linearly independent paths of execution within the method; MN, 'Max Nesting', indicates the maximum number of nested conditional tests in the method; NOAV counts the 'Number of Accessed Variables' to indicate the value of the variables accessed directly by the method.

The axiom equivalent to the Brain Method detection rule is shown in Figure 3. Like the God Class detection axiom, it comprises a set of metrics and thresholds. 'A' points to the ontological class, Method, used to filter out individuals that are not methods.'B' includes methods that have a CC greater than 7. 'C' grants that M is a Brain Method only if its MLOC is above 30. 'D' constrains the maximum nesting in M to greater than 5. 'E' limits the number of accessed variables accessed by M to a value surpassing 5.

A reasoner (Abburu, 2012) can use the axioms presented in Figures 2 and 3 to perform semantic in-



```
Method
 and (hasMeasured some
  (CC
   and (measuredValue some float[> 7.0f])))
 and (hasMeasured some
  (MLOC
   and (measuredValue some float[> 30.0f])))
 and (hasMeasured some
  (MN
   and (measuredValue some float[> 5.0f])))
 and (hasMeasured some
  (NOAV
   and (measuredValue some float[> 5.0f])))
```

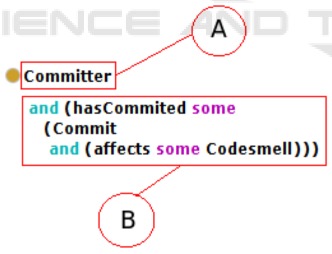
Figure 3: Translating the Brain Method detection rule as an ontological axiom.

```
<EquivalentClasses>
  <Class IRI="http://...#GodClass"/>
  <ObjectIntersectionOf>
    <Class IRI="http://...#Clazz"/>
    <ObjectUnionOf>
      <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasMeasured"/>
        <ObjectIntersectionOf>
          <Class IRI="http://.../metrics#TCC"/>
          <DataSomeValuesFrom>
            <DataProperty IRI="http://.../metrics#measuredValue"/>
            <DatatypeRestriction>
              <Datatype abbreviatedIRI="xsd:float"/>
              <FacetRestriction facet="&xsd;maxExclusive">
                <Literal datatypeIRI="&xsd;float">0.3</Literal>
              </FacetRestriction>
            </DatatypeRestriction>
          </DataSomeValuesFrom>
        </ObjectIntersectionOf>
      </ObjectSomeValuesFrom>
      ...
```

Figure 5: OWL excerpt.

ferring. As a result, occurrences of God Classes and Brain Methods can be retrieved. The retrieval is performed automatically, reducing the effort required by software engineers to handle Code Smells detection in projects.

Other axioms can be added to ONTOCEAN to expand its capacity to produce relevant information. For instance, the axiom contained in Figure 4 is capable of identifying developers (*i.e.*, committers) who create and propagate problematic code. 'A' points to the ontological class, Committer, used to filter out individuals that are not developers. 'B' grants that only committers who make modifications that insert Code Smells are listed as smell propagators.



```
Committer
 and (hasCommited some
  (Commit
   and (affects some Codesmell)))
```

Figure 4: Axiomatic rule to identify Code Smells propagators.

All of the proposed axioms are tested against real projects in Section 5.

ONTOCEAN was created and defined using the second generation of Web Ontology Language (OWL2)[2]. The language was used to model Code Smells and respective axioms. The document exhibited in Figure 5 is an excerpt from the God Class detection axiom which is part of ONTOCEAN's source-code. It shows a rule of equivalence between a God Class and a Class (God Class is a sub-type of Class) to which a value of the TCC is measured and no-greater than 0.3.

As OWL is an open format, ONTOCEAN can be adapted to meet specific needs. The thresholds of the metrics that take part in the smells detection rules can be adjusted. For instance, the ontology can be modified to specify a threshold for TCC different from the initial default (0.3).

Considering that we were able to represent Code Smell detection equations (the ones illustrated in 1 and 2) as semantic rules (figures 2 and 3 respectively) the fulfilment of requirement **REQ2** is granted. In Section 5 we will use the rules to extract instances of Code Smells from the source-code of a software project.

4 OCEAN, A TOOL TO POPULATE ONTOCEAN

OCEAN is composed of a set of components that perform both the source-code mining and the production of ontological individuals that represent Code smells. OCEAN's component-based architecture is illustrated in Figure 6.

OCEAN relies on Visminer API to extract metrics from the source-code of software. Visminer offers a collection of functionalities and automations that are capable of translating source-code Object-Oriented Classes into Abstract Syntax Trees (AST). Java Development Tools (JDT)[3] is the library used by Visminer to create ASTs from source-code. Each AST is analysed to obtain metrics that are evaluated against Code smells production rules, such as 1 and 2. Along with the production of ASTs, Commits are retrieved from GIT repositories and they come associated with other relevant information: the Committer who pushed the commit, the affected files and how they were affected (*e.g.*, lines that were added and/or

[2]https://www.w3.org/TR/owl2-overview/

[3]http://www.eclipse.org/jdt/

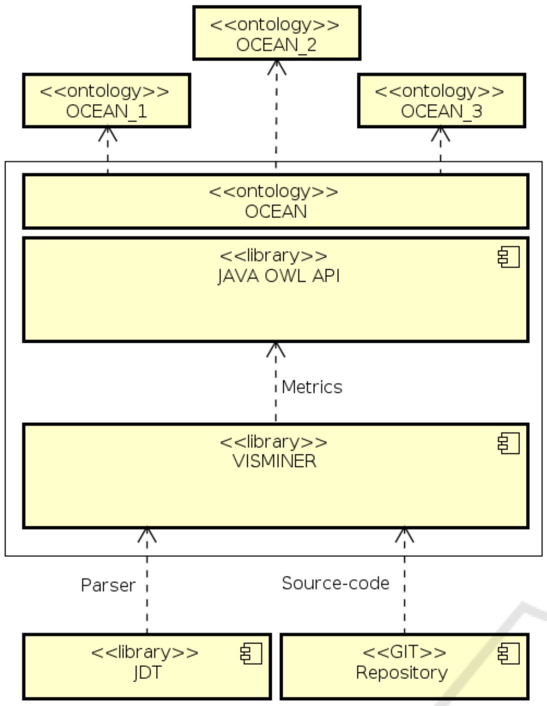


Figure 6: The componentized architecture of OCEAN.

removed), the commit's message and the date-and-time when the commit was sent to the repository.

Java OWL API[4] is another library used by OCEAN. It is responsible for translating Visminer's output (VCS-related information and software metrics) into ontological individuals: Code Smells and Code Smells propagators. As OCEAN runs, an instance of ONTOCEAN is populated by the addition of information extracted from the targeted GIT repository. As a result a new instance of ONTOCEAN is created to encapsulate the individuals of the software project under analysis. Thus, OCEAN provides an exclusive instance of its ontology for every new piece of software being investigated.

We have decided to create a new instance of ONTOCEAN for every new project. The purpose is to isolate the mined/inferred information from the ontological items that define our vocabulary about Code Smells. That is to say, ONTOCEAN is a TBox[5] and it represents our chosen classes to represent Code Smells and related information. Each instance of ONTOCEAN is a new ABox[6] that encapsulates TBox-compliant statements about the vocabulary. Therefore, it is possible to inherit different ABoxes (for each different project being analysed) from the same unique TBox. While ONTOCEAN's TBox works as a template, we minimize the number of individuals to be processed by reasoners by dispersing them into different ABoxes.

At the end of the mining, instances of the classes contained in the ONTOCEAN are produced. Figure 7 illustrates the instances and how they are connected to each other. An instance of 'Repository' is associated with its commits by the 'hasCommit' object property. The 'hasCommitted' property points a 'Committer' to the commits he/she has pushed to the repository. A 'commit' can affect (add/modify) several classes ('ClazzA' and 'ClazzB'), and this relationship is represented by the 'affects' object property. A 'Clazz' may contain one or more 'Method' ('MethodA' and 'MethodB') and this relationship is represented by the 'contains' property. The 'hasMeasured' identifies either a 'Clazz' or a 'Method' associated with specific metrics ('ClassOrientedMetric' and 'MethodOrientedMetric'). 'ClassOrientedSmell' and 'MethodOrientedSmell' contain the rules that produce individuals that are equivalent to 'Clazz' and 'Method' to which the smells's detection rules apply.

5 EVALUATING ONTOCEAN IN USAGE SCENARIO EXAMPLES

To evaluate ONTOCEAN in usage examples, we used it to store information mined by OCEAN from two projects found in GITHUB: JUNIT[7], which is a software testing framework and LOG4J[8], a logging API. Both JUNIT and LOG4J have been applied in the creation of much software, making them a relevant choice for testing. They are public open source projects with access to their source-code which makes the generation of ASTs to obtain software metrics possible. Figure 8 shows OCEAN being executed. 'A' points to the open source project being processed. 'B' shows the list of suspicious software artifacts found. 'C' is the output of the execution: a new instance of our ontology.

Considering JUNIT, OCEAN scanned a total amount of 2299 commits, resulting in the detection of God Classes. Their presence was later confirmed by the execution of a reasoner, HermiT OWL Reasoner(Horrocks et al., 2012), in the Protege[9] environment. It was revealed that all the instances of God Class were related to only one specific class, "org.junit.runners.ParentRunner". In this case, we as-

[4]http://owlapi.sourceforge.net/
[5]Terminological Box
[6]Assertional Box

[7]https://github.com/junit-team/junit4
[8]https://github.com/apache/log4j
[9]http://protege.stanford.edu/

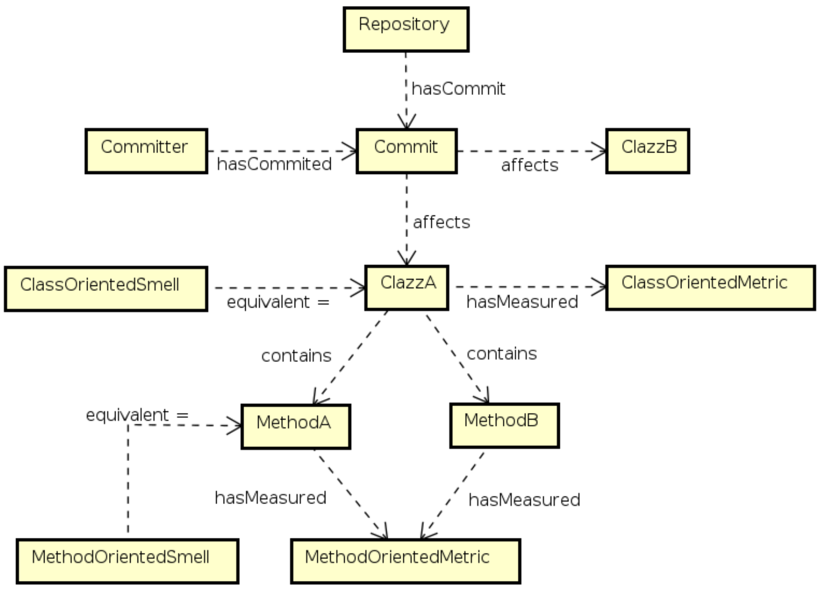
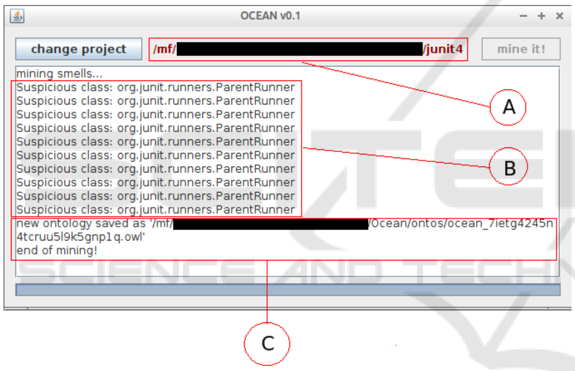


Figure 7: Mined instances of ONTOCEAN's classes.



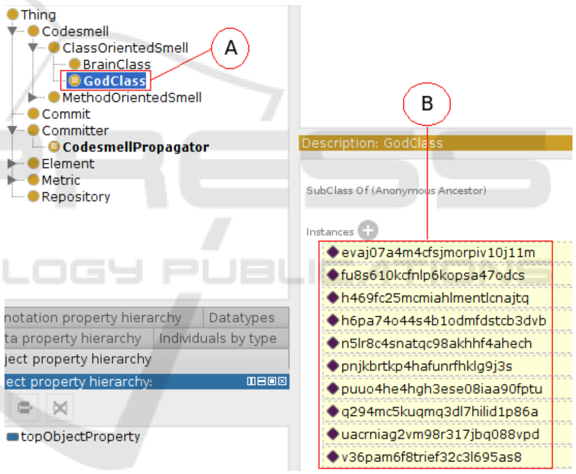
```
/mf/[redacted]/junit4
mining smells...
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
Suspicious class: org.junit.runners.ParentRunner
new ontology saved as '/mf/[redacted]/Ocean/ontos/ocean_7ietg4245n
4tcruu5l9k5gnp1q.owl'
end of mining!
```

Figure 8: Executing OCEAN to mine a project.

sumed that the God Classes were introduced in the past and had remained that way throughout the software evolution, *i.e.*, the same class that produced the instance of God Class was found in early commits, but the project was not revised to solve the anomaly later on. The inferred instances of God Classes are exhibited in Figure 9. 'A' is the ontological class that represents God Class to which the rule exhibited in Figure 2 is attached. 'B' points to the inferred instances of God Class.

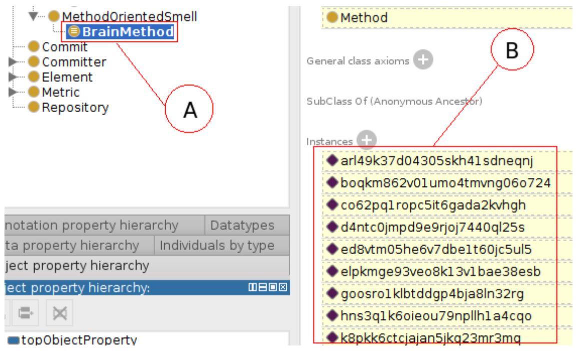
After mining 3561 commits from LOG4J, the HermiT reasoner revealed two types of Code Smells: God Classes and Brain Methods. Figure 10 shows the instances of Brain Methods that were found. 'A' is the ontological class that represents Brain Method. All the instances were produced by the rule shown in Figure 3. 'B' points to the instances of Brain Method.

ONTOCEAN is also capable of representing de-



```
evaj07a4m4cfsjmorpiv10j11m
fu8s610kcfnlp6kopsa47odcs
h469fc25mcmiahlmentlcnajtq
h6pa74o44s4b1odmfdstcb3dvb
n5lr8c4snatqc98akhhf4ahech
pnjkbrtkp4hafunrfhklg9j3s
puuo4he4hgh3ese08iaa90fptu
q294mc5kuqmq3dl7hilid1p86a
uacrniag2vm98r317jbq088vpd
v36pam6f8trief32c3l695as8
```

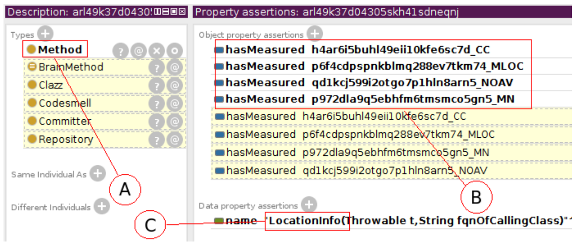
Figure 9: JUNIT's inferred God Classes.



```
arl49k37d04305skh41sdneqnj
boqkm862v01umo4tmvng06o724
co62pq1ropc5it6gada2kvhgh
d4ntc0jmpd9e9rjoj7440ql25s
ed8vtm05he6v7dbe1t60jc5ul5
elpkmge93veo8k13v1bae38esb
goosro1klbtddgp4bja8ln32rg
hns3q1k6oieou79npllh1a4cqo
k8pkk6ctcjajan5jkq23mr3mq
```

Figure 10: LOG4J's inferred Brain Methods.

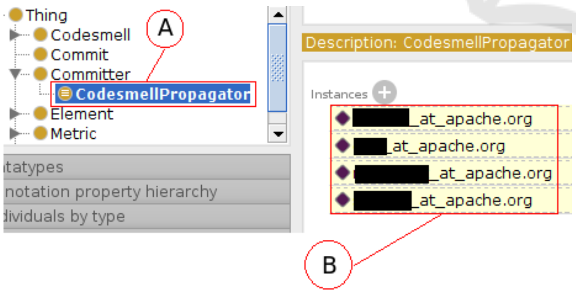
tailed information about Brain Methods. Figure 11 shows data and object properties that were automatically created by OCEAN. The properties enable the precise location of the methods that are Brain Methods. 'A' is the ontological class from which Brain Method instances are inherited: Method. 'B' points to object properties that represent the metrics that took part in the detection: MN (Max Nesting), CC, NOAV and MLOC (according to the rule shown in Figure 3. 'C' is the data property that identifies the method that was detected as a Brain Method: 'LocationInfo' method.



```
hasMeasured h4ar6i5buhl49eii10kfe6sc7d_CC
hasMeasured p6f4cdpspnkblmq288ev7tkm74_MLOC
hasMeasured qd1kcj599i2otgo7p1hln8arn5_NOAV
hasMeasured p972dla9q5ebhfm6tmsmco5gn5_MN
hasMeasured h4ar6i5buhl49eii10kfe6sc7d_CC
hasMeasured p6f4cdpspnkblmq288ev7tkm74_MLOC
hasMeasured p972dla9q5ebhfm6tmsmco5gn5_MN
hasMeasured qd1kcj599i2otgo7p1hln8arn5_NOAV
name 'LocationInfo(Throwable t,String fqnOfCallingClass)"
```

Figure 11: LOG4J's inferred Brain Methods.

The mining of both projects has also produced occurrences of Code Smell propagators. Such information may be interesting in decision making. For instance, the identified developers can be trained in techniques and good practices to combat the occurrence of Code Smell. Figure 12 shows the list of Code Smells propagators found in LOG4J. 'A' is the ontological class that represents Code Smells Propagators produced by the rule defined in Figure 4. 'B' points to the inferred instances of propagators, each instance identified by an email, which is the way each developer is uniquely identified in GITHUB.



```
[redacted]_at_apache.org
[redacted]_at_apache.org
[redacted]_at_apache.org
[redacted]_at_apache.org
```

Figure 12: LOG4J's inferred Code Smells propagators.

By combining ONTOCEAN with a reasoner and evaluating it against software projects we have managed to fulfil requirement **REQ3**. The inferred instances of artifacts affected by Code Smells were manually checked to dissipate any doubt regarding the detection of problematic code, *i.e.*, we are positive that the association of ONTOCEAN with a reasoner is capable of extracting Code Smells from the source-code of software projects.

6 RELATED WORK

To the best of our knowledge, the ontologies proposed by previous works do not fulfil our requirements (as stated in Section 2). Luo et al. (Luo et al., 2010), for instance, developed an ontology to encapsulate Code smells and to evaluate their impact on software. However they do not provide the resources to enable the inferring of Code Smells (**REQ2**). For the same reason we discarded the ontological solution proposed by Cheng and Liao (Cheng and Liao, 2007).

Other works focus on partial data. This is the case, for instance, with the ontology created by Martin and Olsina (de los Angeles Martin and Olsina, 2003). While they cover the representation of metrics, how such metrics can be used to detect smells is not explained. Thus, **REQ1** is not fulfilled.

Approaches proposed by (Kiefer et al., 2007), (Rießet al., 2010), and (Tappolet et al., 2010) are similar to our solution. They created ontologies to represent software elements (*e.g.*, packages, classes, variables, Code Smells) and proposed the use of iSPARQL to detect instances of smells from mined source-code. Reasoning, when applied, was not used to retrieve smells, so **REQ2** and **REQ3** are not fulfilled.

We did not revise works which analyse the presence of Code Smells without using ontologies. Our analysis of related works did not take into consideration works that do not output the detection of smells as an open format like ontologies. This is important because, as pointed out by a recent study conducted by Fernandes et al. (Fernandes et al., 2016), Code Smell detection tools have lacked ways to adequately export analysed smells to reusable formats.

7 CONCLUSION AND FUTURE WORK

It is important to mitigate or alleviate the problems caused by the presence of Code Smells in software projects. To do this, it is necessary to locate precisely the parts of the projects that have been affected by defective coding. Considering that little has been done to promote the use of ontologies and reasoners to perform such a task, we present a combination of an ontology (ONTOCEAN) and tools (OCEAN and reasoners) to support software engineers in dealing with Code Smells. In our understanding we have opened the possibility of expanding the manipulation of smells from the point of static analysis of code to a semantically enriched approach.

By outputting occurrences of Code Smells to ontologies we promote the reuse of this type of information in a way that it would not be possible had we chosen a static format as output (*e.g.*, a PDF-based report on the occurrences of Code Smells). Static outputs are not easily subject to automated reuse as ontologies are. As ontologies are written in a open machine-readable format they show potential for varied applications. In this respect, the information about Code Smells can be expanded and/or reinterpreted to meet other needs.

One may argue that the ontology can be replaced by other storage mechanism, *e.g.*, relational databases. Undoubtedly, using relational databases is the preferred strategy to deal with information storage, but we want to emphasize that the embedding of semantic rules (as the ones used to infer the Code Smells) could not be done as easily and intuitively as with the use of ontologies.

As we conducted tests on large software projects (as seen in Section 5) we realized that both the embedding of detection rules into ontologies and their later activation by reasoners can provide an automated way to spot Code Smells. We were also able to expand the set of axioms to process information beyond the detection of defective code only. This the case with identifying the developers who create and propagate Code Smells. As for our initial requirements, they were fulfilled:

1. **REQ1**: all of the information that is mined by Visminer API was successfully mapped as ontological classes. Thus, we were able to represent software projects (and related information) as well as Code Smells.
2. **REQ2**: although we have not tested a broader set of Code Smells detection rules, we believe that semantic axioms are able to represent them adequately. Plus, they are applicable in combination with reasoners to find instances of faulty code.
3. **REQ3**: granted, as we were able to activate a reasoner upon a populated instance of ONTOCEAN to collect Code Smell data.

An ontology-oriented approach such as ONTOCEAN in association with a source-mining tool such as OCEAN shows potential, if it is considered that: (a) important information associated with Code Smells can be represented as ontological entities; (b) it is possible to represent Code Smells detection rules as axioms to evaluate software metrics; (c) reasoners suche as HermiT can be used to create detection strategies using the axiomatic rules as input; (d) the variability of the data mined from software projects can be used to support decision making processes to guarantee the quality of coding tasks.

We have limited this work to experimenting on the use of an ontology to represent Code Smells and to make use of inferring mechanisms. However we would like to investigate further the advantages that an ontological approach to mine Code Smells can provide. It might be the case with using ONTOCEAN in the context of Semantic Web. Computational agents can collect Code Smell information from ontologies synchronized with projects maintained in GITHUB (or any other VCS) to provide services for software developers, *e.g.*, to either indicate or discard reuse of software parts/components based on the occurrence or absence of Code Smells; or to automatically infer and notify the actions of bad coding propagators to software projects supervisors and teammates.

It would also be interesting to implement variations of the Code Smell production rules or to make the indication of different values for the thresholds of metrics possible. Considering that ontologies are not static formats, the instances of ONTOCEAN can be evolved to encapsulate other rules. The new rules could overlap the default ones and replace the detection strategies, expanding the scenarios illustrated in this work.

It is also of our interest to embed the Hermit reasoner into OCEAN. All the examples of semantic inferring and presentation of the mined information described in this paper were performed by Protege, but OCEAN could be evolved to execute reasoners supported by JAVA OWL API and provide high-level visualizations of Code Smells.

We encourage the replication of our tests and further use of ONTOCEAN and OCEAN. With this purpose, we have made the following available for download and use:

1. ONTOCEAN's set of ontologies [10].
2. The source-code of OCEAN [11].
3. Ontologies obtained from the mining of JUNIT and LOG4J [12].

REFERENCES

Abburu, S. (2012). A survey on ontology reasoners and comparison. *International Journal of Computer Applications*, 57(17).

[10]https://www.dropbox.com/s/xbhcmx4negtkij1/ontocean.zip?dl=0

[11]https://www.dropbox.com/s/j2nn5t83s0ssezk/ocean.zip?dl=0

[12]https://www.dropbox.com/s/bqabnc8ctbscyp0/tests.zip?dl=0

Chandrasekaran, B., Josephson, J. R., and Benjamins, V. R. (1999). What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26.

Chatzigeorgiou, A. and Manakos, A. (2014). Investigating the evolution of code smells in object-oriented systems. *Innov. Syst. Softw. Eng.*, 10(1):3–18.

Cheng, Y.-P. and Liao, J.-R. (2007). An ontology-based taxonomy of bad code smells. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology, Phuket, Thailand*.

Civili, C., Console, M., De Giacomo, G., Lembo, D., Lenzerini, M., Lepore, L., Mancini, R., Poggi, A., Rosati, R., Ruzzi, M., Santarelli, V., and Savo, D. F. (2013). Mastro studio: Managing ontology-based data access applications. *Proc. VLDB Endow.*, 6(12):1314–1317.

Daraio, C., Lenzerini, M., Leporelli, C., Naggar, P., Bonaccorsi, A., and Bartolucci, A. (2016). The advantages of an ontology-based data management approach: openness, interoperability and data quality. *Scientometrics*, pages 1–15.

de los Angeles Martin, M. and Olsina, L. (2003). Towards an ontology for software metrics and indicators as the foundation for a cataloging web system. In *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices (IEEE Cat. No.03EX726)*, pages 103–113.

Djurić, D., Gašević, D., and Devedžić, V. (2005). Ontology modeling and mda. *Journal of Object technology*, 4(1):109–128.

Fenske, W., Schulze, S., Meyer, D., and Saake, G. (2015). When code smells twice as much: Metric-based detection of variability-aware code smells. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 171–180.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 18. ACM.

Fowler, M. (1997). Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland.*

Gruber, T. R. (1993). A translation approach to portable ontology specifications. 5:199–220.

Horrocks, I., Motik, B., and Wang, Z. (2012). The hermit owl reasoner. In *ORE*.

Kiefer, C., Bernstein, A., and Tappolet, J. (2007). Mining software repositories with isparol and a software evolution ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 10. IEEE Computer Society.

Lanza, M. and Marinescu, R. (2010). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Incorporated, 1st edition.

Luo, Y., Hoss, A., and Carver, D. L. (2010). An ontological identification of relationships between anti-patterns and code smells. In *Aerospace Conference, 2010 IEEE*, pages 1–10. IEEE.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Mendes, T. S., Almeida, D., Alves, N., Spnola, R., Novais, R., and Mendona, M. (2015). Visminertd an open source tool to support the monitoring of the technical debt evolution using software visualization. In *17th International Conference on Enterprise Information Systems*. ICEIS.

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

Olbrich, S. M., Cruzes, D. S., and Sjøberg, D. I. (2010). Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. (2014). Do they really smell bad? a study on developers' perception of bad code smells. *ICSME*, 14:101–110.

Rieß, C., Heino, N., Tramp, S., and Auer, S. (2010). Evopat - pattern-based evolution and refactoring of rdf knowledge bases. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 647–662, Berlin, Heidelberg. Springer-Verlag.

Sivaraman, K. (2014). Effective web based e-learning. *Middle-East Journal of Scientific Research*, 19(8):1024–1027.

Smith, C. U. and Williams, L. G. (2000). Software performance antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136, New York, NY, USA. ACM.

Staab, S. and Studer, R. (2013). *Handbook on ontologies*. Springer Science & Business Media.

Tappolet, J., Kiefer, C., and Bernstein, A. (2010). Semantic web enabled software analysis. *Web Semant.*, 8(2-3):225–240.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 403–414, Piscataway, NJ, USA. IEEE Press.

Van Emden, E. and Moonen, L. (2002). Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE.

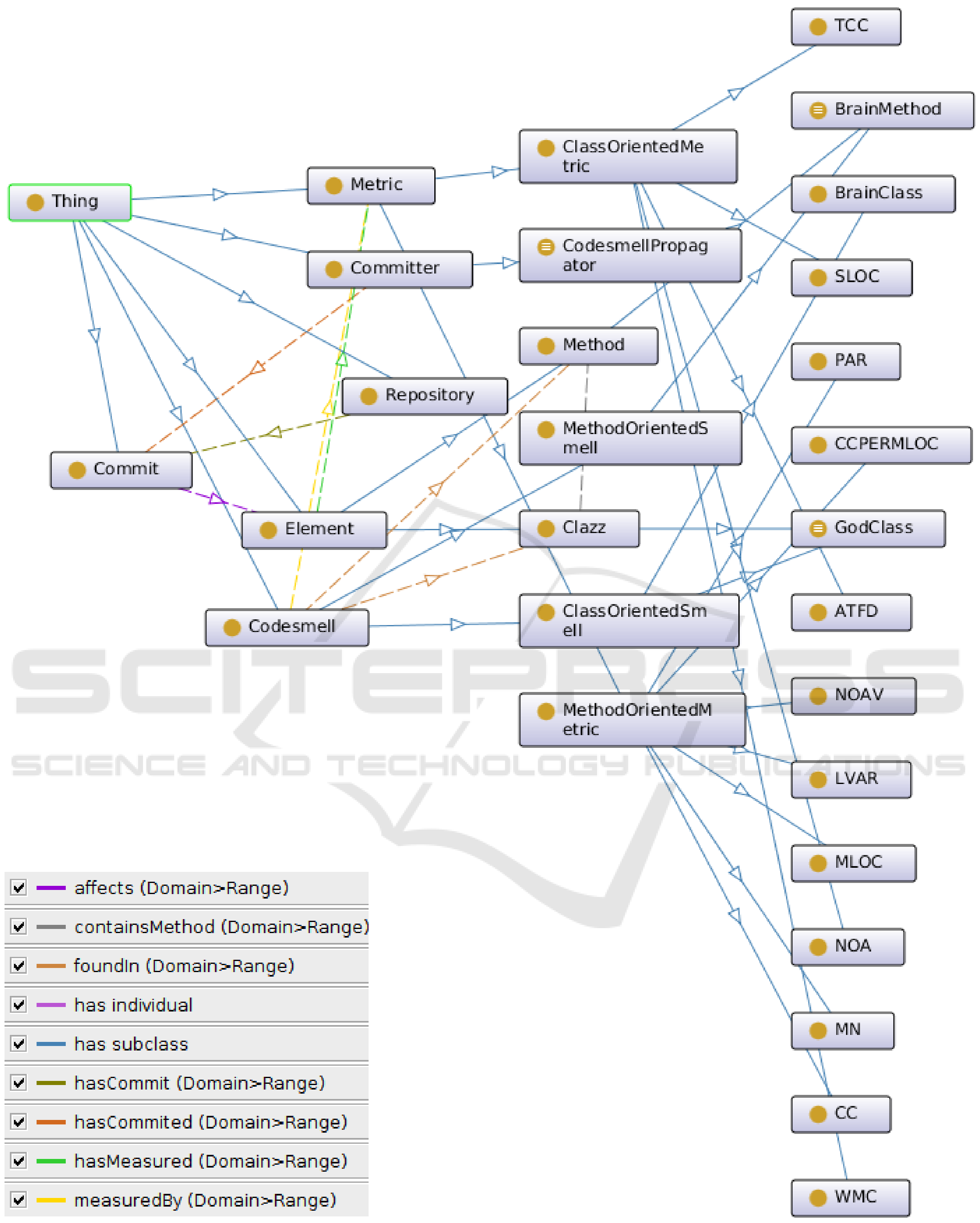


Figure 13: Full display of ONTOCEAN's elements.