# Source Code Interoperability based on Ontology

Camila Zacché de Aguiar
Federal University of Espírito Santo
Vitória, Brazil
camila.zacche.aguiar@gmail.com

Félix Zanetti
Federal University of Espírito Santo
Vitória, Brazil
feluzan@gmail.com

Vítor E. Silva Souza
Federal University of Espírito Santo
Vitória, Brazil
vitor.souza@ufes.br

## ABSTRACT

The different ways of representing a source code in different programming languages create a heterogeneous context. In addition, the use of multiple programming languages in a single source code (polyglot programming) brings a wide choice of terms from different languages, libraries and structures. These facts prevent the direct exchange of information between source codes of different programming languages, requiring specialized knowledge of the programming languages involved. In this article, we present an ontology-based method for source code interoperability that provides an alternative to mitigate heterogeneity problems, aiming to semantically represent the source code written in different programming languages and apply it from different perspectives in a unified way. In this sense, the method is applied in a lab experiment with the objective of validating its methodological aspects, instantiating their respective phases in different subdomains (object orientation and object/relational mapping) and programming languages (Java and Python) in the code smells detection perspective. In addition, the code smell detector produced is evaluated with a set of real-world software projects written in Java and Python.

## CCS CONCEPTS

• **Software and its engineering** → **Interoperability**; • **Information systems** → *Semantic web description languages.*

## KEYWORDS

interoperabity, ontology, applied ontology, source code, code smell detection

**ACM Reference Format:**
Camila Zacché de Aguiar, Félix Zanetti, and Vítor E. Silva Souza. 2021. Source Code Interoperability based on Ontology. In *XVII Brazilian Symposium on Information Systems (SBSI 2021), June 7–10, 2021, Uberlândia, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3466933.3466951

## 1 INTRODUCTION

The different ways of representing a source code in different programming languages create a context of heterogeneity. Based on this concept [14], we define two different types of heterogeneity in source code: *syntactic heterogeneity*, when different syntaxes are attributed to corresponding concepts — for example, both Eiffel's *frozen* syntax and the C++ *final* syntax correspond to a non-extensible class in both programming languages — and *semantic heterogeneity*, when corresponding concepts have different interpretations — for example, the concept of private visibility is interpreted in Java as a restriction of access to members of a certain class by members of external classes, while interpreted in C++, additionally, as a restriction of access to members of a class inherited by its subclasses.

This scenario is part of the daily life of software developers who work with different programming languages in the coding of several source codes or even with multiple programming languages in a single source code. Hence, they need adequate tools to support the differences in these programming languages. With the semantic interoperability of the source code, the tools can operate under a single and shared semantic abstraction layer, contributing to the application of standardized functionalities, reuse, evolution and expansion to different programming languages.

To achieve semantic interoperability, heterogeneity must be eliminated, adopting solutions capable of guaranteeing uniform interpretations. This can be obtained with the support of ontologies built from formal methods and theories in order to clarify concepts and articulate their representations [8]. In this context, ontologies can be a consistent representation of the real world within a context, according to the interpretation of reality. Thus, source codes written in programming languages that share a common consistent commitment can significantly interoperate with each other.

In the context of this article, an ontology is an explicit and formal specification of a shared conceptualization [15], understanding *explicit* as being clear and well-defined; *formal* as computer readable; *conceptualization* as an abstract model of the real world according to a purpose; and *shared* as being based on consensual knowledge of a community on a domain. When referencing ontology of the source code, we are talking about the ontological commitments assumed in a consensual way from different programming languages in order to define concepts of this domain.

An ontology should be built according to a method that supports the development of *Reference Ontologies* — when the main goal is to represent knowledge mainly for human consumption, preserving the clarity and precision of the semantic nature of the entities and their interrelationships in the domain — and *Operational Ontologies* — when the main goal is to ensure computational properties in machine-readable model artifacts. An operational ontology written in the Web Ontology Language (OWL)[1] or the Resource Description Framework (RDF)[2] can formally describe the semantics of a domain through classes, properties and relations, allowing expressiveness, flexibility representing data as triples (subject-predicate-object) and efficiency in supporting reasoning and inference. Thus, although the source code is commonly represented in the syntactic structure of the language grammar, a semantic representation allows for


*SBSI 2021, June 7–10, 2021, Uberlândia, Brazil*

ACM ISBN 978-1-4503-8491-9/21/06…$15.00
https://doi.org/10.1145/3466933.3466951

[1] https://www.w3.org/OWL.
[2] https://www.w3.org/RDF.

new insights, brings a greater understanding of its concepts and relationships and mainly independence of the individual structure of the programming language.

In this paper, we present the OSCIN (Ontology-based Source Code Interoperability) method that aims to semantically represent the source code written in different programming languages and apply it from different perspectives in a unified way. Combining a syntactic analysis of the involved programming languages with an ontological representation of the subdomain, the application perspective from that particular subdomain can be applied in source code files of any of the selected programming languages, instantiated for the ontology from the source code.

The method is applied in a lab experiment in order to validate its methodological aspects, instantiating their respective phases for the detection of code smell in three different scenarios: object-oriented code in Java, object-oriented code in Python and object/relational mapping code in JPA/Java. Therefore, different subdomains (object orientation and object/relational mapping) and programming languages (Java and Python) in the code smells perspective are addressed. In addition, the produced code smell detector is evaluated with a set of real-world software projects written in Java and Python.

The rest of the paper is organized as follows: Section 2 describes the proposed method, illustrating it with a running example. Section 3 reports on lab experiments in which we applied the method ourselves under a few different contexts. An evaluation of the method using real-world software projects and the threats to validity of our experiments are then discussed in sections 4 and 5, respectively. In Section 6, related works are briefly reviewed. Finally, Section 7 concludes the paper.

## 2 THE OSCIN METHOD

In this section we present OSCIN — Ontology-based Source Code Interoperability, a method which is able to semantically represent source code written in different programming languages and apply it from different perspectives in a unified way.

Figure 1 shows an overview of the method. It starts with the *Specification Phase*, when three aspects that define the context in which the method is to be applied are identified. The **subdomain** indicates the portion of the more general domain of source code in which we are interested, e.g., object-oriented (OO) code, code that uses object/relational mapping (ORM), presentation code in Android apps, etc. The **programming language** identified in this phase is the one we intend to analyze, e.g., OO code in Java, ORM code in Python, Android presentation code in Kotlin, and so on. Finally, the **application perspective** tells us which type of application will be performed. In this paper, we focus in the detection of code smells, but other perspectives can be applied, such as source code measurement, model verification, etc.

In the *Subdomain Semantic Abstraction Phase*, the conceptualizations of the identified subdomain are represented in a semantic model, i.e., an **ontology**. In the *Language Syntactic Abstraction Phase*, a **syntactic analyzer** of the identified programming language is produced, i.e., a program that can read code in the language and produce a syntactic structure based on the language grammar

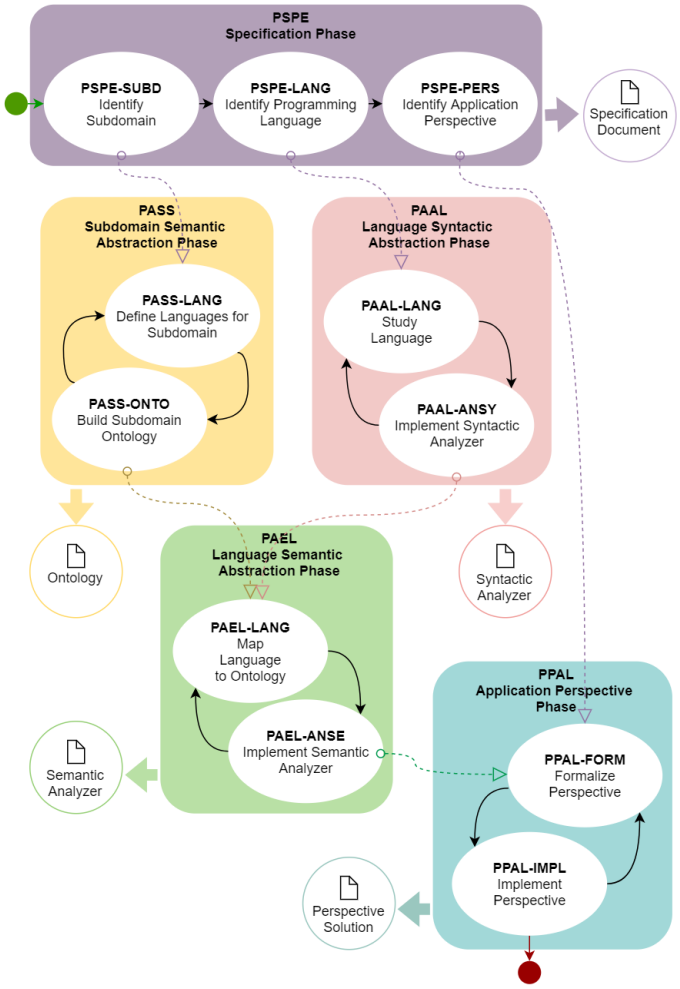


**Figure 1: Overview of the OSCIN method.**

rules. In the *Language Semantic Abstraction Phase*, a **semantic analyzer** for the language and subdomain is produced, i.e., a program that creates instances of the subdomain's ontology from the language's syntactic structure. In the *Application Perspective Phase*, a **perspective solution** for the selected perspective and subdomain is built, i.e., a program that can perform the intended perspective (e.g., code smell detection) based on the ontology instances produced by the semantic analyzer.

Once the method has been fully executed once, it supports the evolution and inclusion of new subdomains, programming languages and perspectives. For instance, if a code smell detector for OO code in Java has been produced using OSCIN, to build one for Python we can reuse the ontology and the application and only need to produce new syntactic and semantic analyzers. Analogously, if we now wanted a smell detector for ORM code in Java, we could reuse the Java syntactic analyzer but we would need a new ontology and new semantic and code analyzers. In that sense, entire phases can be skipped if the artifact to be produced already exists. The following subsections describe OSCIN phases in more detail.

To illustrate the method, we present a running example in which developers want to detect code smells in the presentation layer of Android apps [1].

## 2.1 Specification Phase

This phase aims to identify the characteristics that will be considered for the source code interoperability, that is, subdomain, programming language and analysis perspective.

The software engineer extracts information from the stakeholders about the subdomain, programming language and application perspective to be covered in the method. For instance, code written in an object-oriented fashion should be analyzed quite differently than one written using functional programming, the same goes for Web applications vs. standalone software, code that uses object/relational mapping or not, and so on. Such context must be clearly identified to guide the semantic representation of the source code and its code analysis.

This phase produces as **output** a Specification Document indicating the chosen subdomain, programming language and perspective. In our **running example**, the subdomain of *code for the presentation layer of Android apps* would be specified, together with *Kotlin* as the language and code smell as the perspective.

## 2.2 Subdomain Semantic Abstraction Phase

This phase aims to represent a subdomain in an abstraction of semantic resources, in the form of an ontology. Therefore, this phase deals with existing conceptual differences between different programming languages. In this case, it is possible to observe: (i) different languages with identical tokens and meanings, such as the `class` token adopted by both Java and C++ to represent an abstract data type; (ii) different languages with identical tokens and different meanings, such as the `private` token, adopted in Java to represent that a member is accessible only in the class itself, and adopted in C++ to also represent private inheritance; and (iii) different languages with different tokens and identical meanings, such as the `frozen` and `final` tokens adopted by C++ and Java to represent a non-extensible class. Note that this phase produces a single common artifact for the representation of the previously identified subdomain, i.e., the subdomain represented in the ontology is unique and independent of the programming language.

The ontology engineer, assisted by the domain expert, identifies the relevant programming languages to be considered in the development of the ontology of the selected subdomain. The suggestion is to identify the programming languages for the subdomain of interest according to the following guidelines: (i) *Relevance*, search for programming languages by creation date and, from that list, consider which of the languages were relevant to introduce this subdomain in the programming community; and (ii) *Popularity*, search for programming languages in at least two indexes (adoption rankings) of the programming community. From this list, calculate the average of the programming languages according to these indexes and identify which of the ranked languages address this subdomain.

Following, the ontology engineer extracts knowledge of the subdomain from the domain expert and from the relevant programming languages for the subdomain, representing it in a reference ontology and an operational ontology. The knowledge of the subdomain must be extracted from the languages defined so that the concepts represented in the ontology reflect the consensus reached between these languages, validated by the domain expert and following an ontology engineering method. Therefore, the ontology should not represent specific properties of a single language or even of the chosen perspective, but the shared conceptualization of the subdomain. Later, code analysis is performed by querying the ontology.

The quality and coverage of the ontology are key features for the quality of the application of the perspective. For this reason, it is highly recommended that the ontology is supported by a foundational ontology, to adopt an ontology engineering method that covers from reference to operational ontologies and provide semantic rules to support information inference. Further, the reuse of concepts already established in existing ontologies is encouraged and can bring advantages related to the maturity of the ontology and the time devoted to the application of the method. On one hand, the reference ontology must be expressive and accurately represent the subdomain, on the other hand, the operational ontology must consider the performance of reasoning and inference.

This phase consumes as **input** the subdomain identified in the Specification Phase (cf. Section 2.1) and produces as **output** reference and operational ontologies for that subdomain. In our **running example**, an ontology on the presentation layer of Android apps would be developed, preferably reusing ontologies about the Android platform in general and about object-oriented code. The reference ontology could be an OntoUML [7] model and its specification document, whereas the operational ontology would be an OWL file based on the reference model.

## 2.3 Language Syntactic Abstraction Phase

This phase aims to represent source code written in the programming language into an abstraction of syntactic features, i.e., one that considers the structure and shape of the source code. Existing parsers could be reused (and we suggest they are), but even in this case the *Study Language* activity remains necessary, since this knowledge will be used in the next phase.

The software engineer understands the syntactic structure of the programming language, i.e., its tokens and grammar. Since the definition of tokens and grammar results from the language itself, we suggest that this phase be carried out from the documentation of the language as well as its specification.

Following, the software engineer reuses or implements a syntactic analyser for the programming language. Such analyzer is basically a parser (from a compiler), able to map the characters of a source code in the language to the set of tokens of that language and then map to the grammar of that language in order to build its syntactic structure, e.g., an abstract syntax tree (AST). We highly recommend using an available parser, but the software engineer could decide to build a syntactic analyzer from scratch.

This phase consumes as **input** the programming language identified in the Specification Phase (cf. Section 2.1) and produces as **output** the syntactic analyzer of the language, which can later be used to analyze any source code written in this language. In our **running example**, ANTLR[3] would be used to generate a parser based on the existing Kotlin grammar.

[3] https://www.antlr.org/

## 2.4 Language Semantic Abstraction Phase

This phase aims to represent the source code into an abstraction of semantic features, i.e., one that considers the meaning of the source code. Although the previous phase allows us to know the elements that make up a source code, it does not explain the role these elements play in the code. For example, the `private` operand in Java is represented by the *classModifier* operator in the syntactic abstraction phase but that does not mean that the role of that operator is to modify the visibility of a class. Although the name of the operator is suggestive, it only tells us that it is a leaf node of the *classModifier* node in the AST.

Therefore, this phase intends to assign semantics to the elements of a programming language, noting that a token can represent, in a given programming language: (i) a single, independent meaning of grammar, e.g., the `class` token that represents an abstract data type in the Java language; or (ii) different meanings depending on the grammar, e.g., the `final` token which can represent a non-extensible class, a non-overridable method or a constant (non-modifiable variable) in Java, depending on where it is used.

The software engineer maps the syntax of a programming language to the meaning of the subdomain, in order to assign semantics to the source code. This mapping is carried out in a conceptual way linking the concepts defined in the ontology developed in the Subdomain Semantic Abstraction phase with symbols of the programming language identified in the Language Syntactic Abstraction phase. Note that: (i) not all symbols of the programming language will be mapped to the ontology and may correspond to exactly one, several or no semantic concepts of the ontology; and (ii) not all concepts of the ontology will be mapped to the programming language and may correspond to exactly one, several or no symbols of the language.

The software engineer implements the semantic analyzer for the programming language mapped, i.e., a program that is able to produce instances of the operational ontology from source code files written in that language. Typically, visitor methods are used to walk the AST looking for the syntactic structure that corresponds to the concept of the ontology. Then, one must implement the instantiation process of the ontology.

This phase consumes as **input** the reference ontology elaborated in the Subdomain Semantic Abstraction phase (cf. Section 2.2) and the grammar from the Language Syntactic Abstraction phase (cf. Section 2.3) and produces as **output** a semantic analyser of this language, which can be used later to analyze any source code written in that language for the given subdomain. In our **running example**, a spreadsheet would map language tokens to ontology concepts and visitor methods for the Kotlin parser generated with ANTLR would be implemented in order to produce RDF triples with instances of the operational ontology (i.e., an OWL file).

## 2.5 Application Perspective Phase

This phase aims to implement an application that, according to the semantic representation of the subdomain (i.e., the ontology), reads code in the chosen programming language and provides an application according to the perspective. In our case, a code smells detector.

The software engineer, assisted by the ontology engineer, formalizes the characteristics of the perspective, following the definition established in the Specification Phase. Each perspective has its own application in the method, being able to consider different points of the source code in different ways. For instance, for the bad smell perspective, one can formalize smells in the form of a semantic query, that is, queries elaborated on the terms of the subdomain ontology in a query language applicable on the operational ontology. In addition, for the formalization of each smell, it is suggested to use rules already established in the literature, if available. Since most of these rules are not based on an ontology, this activity may require adaptations or construction of new rules.

Following, the software engineer implements the application in order to identify the instances of that ontology that correspond to the perspective. Therefore, a single ontology is used to identify different perspectives of the code and a single implementation is used to identify the same perspective in different programming languages. Since the formalization is represented from the ontology, it is easily customized, updated and it is independent of the programming language. On the other hand, the knowledge of the subdomain ontology is indispensable to deal with the perspective.

This phase consumes as **input** the operational ontology of the subdomain built in the Subdomain Semantic Abstraction phase (cf. Section 2.2) and the perspective identified in the Specification Phase (cf. Section 2.1) and produces as **output** the application, which can be used later to any source code in the specified context (subdomain, programming language, perspective). In our **running example**, some sort of OWL API would be used to implement a program that reads Kotlin files using the parser, produces instances of the operational ontology and runs the queries over such instances, identifying which points in the code present any of the specified smells.

# 3 APPLYING THE OSCIN METHOD

To assess the methodological aspects of OSCIN, we performed a laboratory experiment for detecting code smells in three different scenarios, involving three researchers as domain experts, two of whom play the role of software engineer and ontology engineer and one of them of stakeholder: object-oriented code in Java, object-oriented code in Python and object/relational mapping code in JPA/Java. The experiment, thus, addresses different subdomains (object orientation, object/relational mapping) and programming languages (Java, Python) in the code smells perspective.

The following subsections report on each scenario. All the artifacts produced during the application of OSCIN are available at the project website.[4]

## 3.1 Object Orientation smells in Java

In the *Specification Phase*, the software engineer interviewed the stakeholders and specified the subdomain of Object Orientation (OO) — defined as "a software implementing method in which programs are organized as cooperative collections of objects, each representing an instance of some class and whose classes are members of a class hierarchy linked by inheritance relationships".

The software engineer also specified Java as the programming language and, in the perspective of code smells, specified a simple smell to be detected as a proof-of-concept: *Long Parameter List* [3,

[4] https://nemo.inf.ufes.br/projetos/oscin/

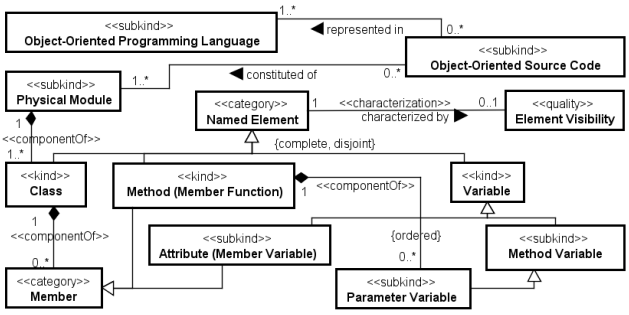


**Figure 2: Fragment of the OOC-O reference ontology [4].**

10, 11], a smell that appears when a method has a very long list of parameters which makes it difficult to use and understand. The long list of parameters is a smell because everything a method needs is passed through parameters so that the list of parameters becomes inconsistent, difficult to manipulate and constantly changing.

Next, in the *Subdomain Semantic Abstraction Phase*, the software engineer selected five programming languages for the OO subdomain according to their relevance and popularity in the TIOBE,[5] IEEE Spectrum,[6] and REDMONK[7] indexes. Thus, Java, Python and C++ were selected for their *popularity*. Smalltalk and Eiffel were also selected due to their *relevance*.

Once the languages were selected, the SABiO [5] ontology engineering method was applied for the development of the Object-Oriented Code Ontology (OOC-O) (detailed in [4]), both in reference and operational versions. The ontology uses the UFO [9] foundation ontology for ontological analysis, the OntoUML [7] language for graphical representation at the reference level and the OWL language for codification at the operational level. Figure 2 shows a fragment of the OOC-O reference ontology and Listing 1 a fragment of the operational ontology.

**Listing 1: Fragment of the OOC-O operational ontology [4].**

```
<owl:Class rdf:about="http://ooc-o#Class">
  <rdfs:subClassOf rdf:resource="http://ooc-o#Named_Element"/>
  <rdfs:subClassOf rdf:resource="http://ooc-o#Type"/>
  <owl:disjointWith rdf:resource="http://ooc-o#Member_Function"/
      >
  <owl:disjointWith rdf:resource="http://ooc-o#Primitive_Type"/>
  <owl:disjointWith rdf:resource="http://ooc-o#Variable"/>
</owl:Class>
<owl:Class rdf:about="http://ooc-o#Member_Function">
  <rdfs:subClassOf rdf:resource="http://ooc-o#Member"/>
  <rdfs:subClassOf rdf:resource="http://ooc-o#Named_Element"/>
  <owl:disjointWith rdf:resource="http://ooc-o#Member_Variable"/
      >
  <owl:disjointWith rdf:resource="http://ooc-o#Variable"/>
</owl:Class>
<owl:ObjectProperty rdf:about="http://ooc-o#componentOfClass">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
      FunctionalProperty"/>
  <rdfs:domain rdf:resource="http://ooc-o#Member"/>
  <rdfs:range rdf:resource="http://ooc-o#Class"/>
</owl:ObjectProperty>
```

[5]https://www.tiobe.com/tiobe-index/.
[6]https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020.
[7]https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/.

Afterwards, in the *Language Syntactic Abstraction Phase*, the software engineer studied the Java programming language based on its specification [6]. Then, a syntactic analyzer was implemented in Java reusing the ANTLR parser generator and one of its available grammar files for Java.

Next, in the *Language Semantic Abstraction Phase*, the software engineer prepared a mapping table between the concepts of OOC-O and the syntactic structures of the Java programming language. For instance: Class is mapped to *identifier* terminal symbol that makes up a *ClassDeclaration* non-terminal; Element Visibility is mapped to *public*, *private* or *protected* terminal symbol that makes up a *ClassModifier* non-terminal symbol of a *ClassDeclaration*; and Abstract Class is mapped to *abstract* terminal symbol that makes up a *ClassModifier* non-terminal symbol of a *ClassDeclaration*. Then, a semantic analyzer was implemented in Java, again based on ANTLR and using the Jena framework[8] to generate ontology instances. The mapping is implemented by looking for the syntactic structures of Java that correspond to each concept of the OOC-O ontology, representing those structures in an OWL ontology (RDF triples).

Next, in the *Application Perspective Phase*, the ontology engineer formalized the *Long Parameter List* code smell in a SPARQL query using the concepts of OOC-O, as shown in Listing 2. Such formalization considers the case of more than six parameters, hence the query searches for member functions (line 4) of a class (lines 3 and 6) whose number of parameter variables (lines 5 and 7) is greater than six (line 10).

**Listing 2: Long Parameter List smell as a SPARQL query.**

```
1 SELECT ?class ?method (count(?variable) as ?countVariable)
2 WHERE {
3   ?class rdf:type ooc-o:Class .
4   ?method rdf:type ooc-o:Member_Function  .
5   ?variable rdf:type ooc-o:Parameter_Variable .
6   ?method ooc-o:componentOfClass ?class .
7   ?variable ooc-o:componentOfMemberFunction ?method
8  }
9 GROUP BY ?class ?method
10 HAVING (?countVariable > 6)
```

Then, a code analyzer was implemented in Java reusing the Jena framework. The analyzer loads the instantiated ontology and applies SPARQL queries to detect smells in the source code instantiated by the ontology. The query results are represented in the form of RDF nodes and correspond to the parts of the source code that "smell".

## 3.2 Object Orientation smells in Python

To detect object-oriented smells in Python, much of the work done in the previous scenario was reused. In the *Specification Phase*, only the programming language was changed to Python. The *Subdomain Semantic Abstraction Phase* was skipped, as OOC-O would be reused in this scenario.

Then, in the *Language Syntactic Abstraction Phase*, the software engineer studied the Python specification[9] and, analogous to the Java scenario, implemented a parser using ANTLR and an available grammar file for Python.

[8]https://jena.apache.org
[9]https://docs.python.org/3/reference/index.html

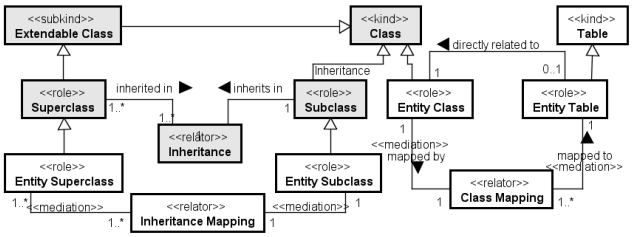


**Figure 3: Fragment of the ORM-O reference ontology [18].**

Also as before, in the *Language Semantic Abstraction Phase* a mapping table between OOC-O and Python was prepared. For instance: Class is mapped to *name* terminal symbol that makes up a *classdef* non-terminal; Element Visibility is not mapped; and Member Function is mapped to *name* terminal symbol that makes up a *funcdef* non-terminal symbol.

Then, a semantic analyzer was implemented in Java, again based on ANTLR and Jena. Once that was done, the *Application Perspective Phase* could be skipped, as the code analyzer for the Long Parameter List smell had already been built in the previous scenario.

## 3.3 Object/Relational Mapping smells in JPA

In this scenario, at the *Specification Phase* the subdomain changed to *Object/Relational Mapping* (ORM), which, as specified by the software engineer, "defines how communication between object-oriented and relational paradigms is performed, automating the persistence of object data in database tables and columns and the construction of objects from data retrieved by database queries". Java was specified as programming language again but, more specifically, the Java Persistence API (JPA) standard was considered.

The perspective remained that of detecting code smells, but the software engineer specified an ORM smell to be detected, namely *Multi Directed Table*: smell that appears when entity classes that are not part of the same hierarchy share the same database table. This does not cause compilation or execution errors, but allows solutions that violate the database rules, i.e., have some columns in the table refer to attributes of a class and other columns to attributes of another, unrelated class.

In the *Subdomain Semantic Abstraction Phase*, ORM frameworks were selected for the development of the subdomain ontology. As a subdomain of Object-Orientation, the ontology engineer decided to focus on the three most popular languages from the OO scenario (C++, Java and Python). Thus, other than JPA, the ontology engineer selected ORM frameworks Django and SQLAlchemy according to their *popularity* for Python; and QxORM and ODB according to their *popularity* for C++. Again applying the SABiO [5] method, the Object/Relational Mapping Ontology (ORM-O) [18] was built based on OOC-O from the previous scenarios. Figure 3 shows a fragment of the ORM-O reference ontology (OOC-O concepts have gray background) and Listing 3 a fragment of the operational ontology.

Since the Java language had already been studied and mapped to OOC-O in a previous scenario, in the *Language Syntactic Abstraction* and *Language Semantic Abstraction* phases the software engineer

**Listing 3: Fragment of the ORM-O operational ontology [18].**

```
1  <owl:Class rdf:about="http://orm-o#Entity_Class">
2      <rdfs:subClassOf rdf:resource="http://ooc-o#OOC-O::Class"/>
3  </owl:Class>
4  <owl:Class rdf:about="http://orm-o#Class_Mapping"/>
5  <owl:Class rdf:about="http://orm-o#Entity_Table">
6      <rdfs:subClassOf rdf:resource="http://rdbs-o#Table"/>
7  </owl:Class>
8  <owl:ObjectProperty rdf:about="http://orm-o#directlyRelatedTo">
9      <rdfs:domain rdf:resource="http://orm-o#Entity_Table"/>
10     <rdfs:range rdf:resource="http://orm-o#Entity_Class"/>
11 </owl:ObjectProperty>
```

studied the JPA specification[10] and mapped it to ORM-O, using a different architecture from the OO/Java scenario (JavaParser[11] and OWLAPI[12] libraries). Since JPA uses Java annotations to perform object/relational mapping, once these annotations are found in the source code, concepts of ORM-O were instantiated along OOC-O ones.

Then, in the *Application Perspective Phase* the ontology engineer formalized the *Multi Directed Table* smell, as shown in Listing 4. It simply searches for an entity table (line 3) that is related (line 4) to more than one class (line 7).

**Listing 4: Formalization of the Multi Directed Table smell as a SPARQL query.**

```
1 SELECT ?table (count(?class) as ?countClasses)
2 WHERE {
3   ?table rdf:type orm-o:Entity_Table .
4   ?table orm-o:directly_related_to ?class  .
5  }
6 GROUP BY ?class
7 HAVING (?countClasses > 1)
```

Finally, a code analyzer was implemented in Java reusing the OWLAPI library. The analyzer loads the instantiated ontology and applies SPARQL queries to detect *Multi Directed Table* smells in Java code bases that use JPA.

# 4 EXPERIMENT AND RESULTS

Besides the applications presented in Section 3, we evaluated the results of applying the OSCIN method, i.e., the code analyzer that detects the *Long Parameter List* code smell produced in sections 3.1 and 3.2, to a set of real-world software projects written in Java and Python. From the Java Qualitas Corpus,[13] which is a large curated collection of open source Java projects [16] with different sizes and purposes, five open-source software projects were selected, namely: (i) the JUnit testing framework; (ii) the Log4j logging framework; (iii) the jEdit source code editor; (iv) the Jena linked data/Semantic Web framework; and (v) the Maven software project management and comprehension tool. From GitHub,[14] five open-source software projects written in Python were selected, namely: (i) the Requests HTTP library; (ii) the Pillow image library; (iii) the Numpy array

[10] http://jcp.org/en/jsr/detail?id=338
[11] https://www.javaparser.org
[12] http://owlcs.github.io/owlapi
[13] http://qualitascorpus.com
[14] https://github.com/psf/requests, https://github.com/python-pillow/Pillow, https://github.com/numpy/numpy, https://github.com/numpy/numpy, https://github.com/Theano/Theano

**Table 1: Results for detecting smells in real-world projects.**

| Project | Files | Classes | Methods | Smells | Time (s) |
|---|---|---|---|---|---|
| JUnit 5.0 | 1137 | 2496 | 8984 | 4 | 5101 |
| Log4j 2.0 | 503 | 982 | 3277 | 39 | 2944 |
| jEdit 4.3.2 | 519 | 1479 | 6830 | 48 | 3125 |
| Jena 2.6.3 | 914 | 1315 | 9900 | 6 | 5226 |
| Maven 3.0.5 | 761 | 1439 | 6048 | 48 | 4625 |
| Django 3.1 | 2674 | 7886 | 24995 | 119 | 35791 |
| Numpy 1.19.1 | 487 | 1480 | 10835 | 40 | 7616 |
| Pillow 7.2 | 272 | 266 | 2932 | 13 | 323 |
| Requests 2.24 | 35 | 76 | 608 | 6 | 95 |
| Theano 1.0.5 | 380 | 1268 | 9653 | 179 | 9065 |

processing library; (iv) the Django Web application framework; and (v) the Theano optimization compiler library.

Each software project was processed separately by the code smell detector and a summary of the results is presented in Table 1. It shows, for each project, the number of `.java` and `.py` files processed, the number of classes identified on those files, the number of methods identified on those classes, the number of *Long Parameter List* smells identified on those methods and the processing time in seconds.

The time spent by the code smell detector is proportional to the number of files processed considering the complexity and quality of the code, between 1 to 7 seconds per class in a modest setting machine (2.70GHz processor, 256GB SSD and 8GB RAM). This time is dedicated 12,06% for the code parser, 87,79% for the ontology instantiation and 0,15% for the smells detection, which is reasonable compared to other approaches and considering that our algorithm has not been optimized for performance. Note, also, that including other smells has a very low effect on the overall time.

The results show that a code smell detector produced by the OSCIN method can be applied to real-world software projects of considerable size. Since other smells might be formalized in more complex SPARQL queries, further tests using a more complete catalog of smells is recommended and subject of future work.

## 5 THREATS TO VALIDITY

We now discuss the main threats to the validity of our experiments, according to four types of validity: (i) internal, (ii) external, (iii) conclusion, and (iv) construct validity [17].

*Internal validity* threatens the conclusion of the experiment on a possible cause and effect relationship between the method and the result. The artifacts produced in the application of the method are totally related to its result and, therefore, the quality of the ontology and the query can be a threat to the result of the method since the ontology may not represent the concepts necessary for the detection of smells and the query is customizable according to the authors of the paper. To mitigate these threats, the method defines that the ontology must be built following an established ontology engineering method and that the query must be formalized using the concepts of the ontology and literature catalogs of smells, when applicable. In addition, the type of smell adopted in the experiment can be a threat to the representativeness of smells as other types of smells may require other detection strategies. This threat can be mitigated by adding more types of smells in future works.

*External Validity* threatens the generalization of the method for use in industry. The number of domains and programming languages used in the experiment may not be enough to generalize the method for any source code. To reduce the impact of this threat, we instantiated the method with two different source code subdomains and programming languages. Moreover, to eliminate the threat of analyzing toy software, we have selected software projects recognized by the community, with various purposes and sizes.

*Conclusion Validity* threatens the correct conclusion on the results. The reader's interpretation of the application of the method can be a threat and, therefore, the method was described with detailed instructions and running examples. To minimize threats related to the definition of the method as a standard, the method was used by two different subjects, although from the same research group and in an academic context. Thus, threats to the heterogeneity of the subjects should be addressed in future works.

*Construct Validity* threatens the generalization of the method. In order to eliminate the threat of projects influenced by the chosen perspective of code smells, the analyzed projects were captured from an open source repository, developed in a normal software development proccess and composed of different programmers. Moreover, the detection of smells was analyzed only quantitatively and threats to its accuracy and customization should also be addressed in future works.

## 6 RELATED WORKS

Very few papers propose a method to source code interoperability or source code analysis based on ontology. In this context, we find the DECOR method [12], which presents the steps to define a smells detection technique by specifying smells rules in a Domain Specific Language (DSL) to automatically generate detection algorithms. Code smells are specified in DSL rules that are reified in a smell model from an object-oriented metamodel and a parser. This model must be instantiated individually for each smell and from the visitor methods generate a specific detection algorithm for that smell using templates. On the other hand, the source code files are reified in the source code model based on reverse engineering. Then, the detection algorithms generated from code smell model are applied on the source code model. Although the DECOR method is able to detect different code smells on a high level abstraction using a DSL, we can observe that the method represents the source code in a syntactic model and code smells in a language-independent metamodel that generates specific algorithms. Taking a different view, OSCIN represents the source code in a semantic model (an ontology) and code smells in a query language on that model.

Other works do not propose a method for deriving code smell detectors, but instead a detector for specific types of smell. In particular, some of these works use ontologies to represent the source code or a common detection rule for different languages, as in iSPARQL [10] and OCEAN [2].

iSPARQL [10] presents software, bug and versioning ontologies for source code analysis built on OWL. The ontology is used as a means of storing the elements needed to analyze the software, which is done via iSPARQL, an engine that extends the SPARQL

language with some facilities for query execution. The work focuses on programming languages and metrics of object-orientation that are instantiated in the ontology through a parser in order to perform similarity measures on the source code. In turn, OCEAN [2] presents a set of components that allows the mining of source code and the production of ontological individuals that represent code smells. The ontology stores the source code elements and the software metrics from the Abstract Syntax Tree, i.e., the metrics are extracted from the AST. The work focuses on the Java language and on metrics of object orientation.

Similarly, the use of common detection rules is observed in MLSD [13], Multiple Language Smells Detector, which presents an intermediate representation of the source code as database in order to identify code smell from SQL queries. The source code is structured in tables from reverse engineering using parsing, regular expressions and software metrics. From the database, code smells are detected by applying SQL queries on the database schema, enabling application in multiple languages.

Although these works look similar to ours, they differ from ours by: (i) presenting a computational tool and not a complete method for the detection of smells; (ii) representing the source code as an ontology, limited to only a few object-oriented elements or as a database, without semantic characteristics; (iii) representing software metrics in the ontology or in the database calculated from the syntactic structure of the source code; and (iv) limiting the scope of the ontology-based works to the object-oriented paradigm or the Java language.

OSCIN differs from existing proposals by allowing code smells to be formalized as queries over a unified model that represent a given *source code* subdomain (e.g., the object-oriented source code subdomain, represented by OOC-O), which makes them applicable to code in any programming language, as long as it is included in the given subdomain and the necessary parsers have been built (e.g., Java, Python, etc.). The unified model (ontology) also makes it easier for us to define new smells and customize smells detection. Finally, new OSCIN executions can reuse artifacts produced previously.

## 7 CONCLUSIONS

This article presented OSCIN, an Ontology-based Source Code Interoperability method that aims to semantically represent the source code written in different programming languages and apply it from different perspectives in a unified way. We systematically presented the method in detail, illustrating it with a running example, and apply the method to: (i) Java and, with reuse, also Python, validating support for multiple programming languages with few changes; (ii) the object-oriented subdomain and, with reuse, also the object/relational mapping subdomain, validating support for multiple subdomains; (iii) definition of smells cataloged in the literature or customized by the stakeholder in queries over ontology concepts, validating the definition flexibility and customization.

Furthermore, the application presented in the article was applied to large and well-known software projects, processing about 7.682 files, 18.687 classes, 84.062 methods and 502 bad smells. Results show the importance of the characteristics that the OSCIN method proposes, as well as the differences with respect to other related works.

OSCIN allows the semantic representation of the source code (in essence), independence of the programming language and flexibility in the source code analysis, which differs from the related works. It is worth noting that the semantic representation of source code makes it possible to explore and evolve different views on the analysis of source code looking forward. Finally, future work intends to focus on the implementation of a tool that helps the programming community to apply the method in several domains and programming languages.

## REFERENCES


[1] Suelen Goularte Carvalho, Maurício Aniche, Júlio Veríssimo, Rafael S. Durelli, and Marco Aurélio Gerosa. 2019. An empirical catalog of code smells for the presentation layer of Android apps. *Empirical Software Engineering* 24, 6 (dec 2019), 3546–3586.

[2] Luis Paulo da Silva Carvalho, Renato Lima Novais, Laís do Nascimento Salvador, and Manoel Gomes de Mendonça Neto. 2017. An Ontology-based Approach to Analyzing the Occurrence of Code Smells in Software. In *Prof. of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS*. ScitePress, 155–165.

[3] Phongphan Danphitsanuphan and Thanitta Suwantada. 2012. Code smell detecting tool and code smell-structure bug relationship. In *2012 Spring Congress on Engineering and Technology*. IEEE, 1–5.

[4] Camila Zacché de Aguiar, Ricardo de Almeida Falbo, and Vítor E Silva Souza. 2019. OOC-O: A reference ontology on object-oriented code. In *International Conference on Conceptual Modeling*. Springer, 13–27.

[5] Ricardo A. Falbo. 2014. SABiO: Systematic Approach for Building Ontologies. In *Proc. of the 1st Joint Workshop ONTO.COM / ODISE on Ontologies in Conceptual Modeling and Information Systems Engineering*. CEUR, Rio de Janeiro, RJ, Brasil.

[6] J Gosling, B Joy, G Steele, G Bracha, A Buckley, and D Smith. 2018. The Java language specification: Java SE 10 edition, 20 February 2018.

[7] Giancarlo Guizzardi. 2005. *Ontological Foundations for Structural Conceptual Models*. PhD Thesis. University of Twente, The Netherlands.

[8] Giancarlo Guizzardi. 2019. Ontology, Ontologies and the "I" of FAIR Giancarlo. *Data Intelligence* 23, November (2019), 0–2. https://doi.org/10.1162/dint

[9] Giancarlo Guizzardi and Gerd Wagner. 2004. A Unified Foundational Ontology and some Applications of it in Business Modeling. In *Proc. of the 2004 Open InterOp Workshop on Enterprise Modelling and Ontologies for Interoperability*. CEUR.

[10] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. 2007. Analyzing software with iSPARQL. In *Proc. of the 3rd International Workshop on Semantic Web Enabled Software Engineering*. 1–15.

[11] Rohit Kumar and Jaspreet Singh. 2016. A unique code smell detection and refactoring scheme for evaluating software maintainability. *International Journal of Latest Trends in Engineering and Technology* 7 (2016), 421–436.

[12] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.

[13] Ghulam Rasool and Zeeshan Arshad. 2017. A lightweight approach for detection of code smells. *Arabian Journal for Science and Engineering* 42, 2 (2017), 483–506.

[14] Amit P. Sheth. 1999. Changing Focus on Interoperability in Information Systems: from Systems, Syntax, Structures to Semantics. *Interoperating Geographic Information Systems* (1999).

[15] Rudi Studer, V Richard Benjamins, and Dieter Fensel. 1998. Knowledge engineering: principles and methods. *Data & knowledge engineering* 25, 1-2 (1998), 161–197.

[16] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proc. of the 2010 Asia Pacific Software Engineering Conference (APSEC2010)*. IEEE, 336–345.

[17] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer.

[18] Félix Luiz Zanetti, Camila Zacche de Aguiar, and Vıtor E Silva Souza. 2019. Representacao ontologica de frameworks de mapeamento objeto/relacional. In *Proc. of the 12th Seminar on Ontology Research in Brazil (ONTOBRAS 2019). CEUR, Porto Alegre, RS, Brasil.*