OOC-O: a Reference Ontology on Object-Oriented Code

Camila Zacché de Aguiar, Ricardo de Almeida Falbo, and Vítor E. Silva Souza

Ontology & Conceptual Modeling Research Group (NEMO)
Federal University of Espírito Santo, Brazil
camila.zacche.aguiar@gmail.com, {falbo, vitorsouza}@inf.ufes.br
http://nemo.inf.ufes.br/

Abstract. With the rise of polyglot programming, different programming languages with different constructs have been combined in the same software development projects. However, to our knowledge, no axiomatization demonstrating the existential commitments of a language have been presented, nor is there effort to adopt a consensual conceptualization between languages, in particular object-oriented ones. In this paper, we propose OOC-O, a reference ontology on Object-Oriented Code whose purpose is to identify and represent the fundamental concepts present in OO source code. The ontology is based on UFO, was developed according to the SABiO method, verified according to its competency questions and validated by instantiation of concepts in OO code form and a process of harmonization among popular object-oriented languages.

Keywords: Object-Oriented Ontology · Polyglot Programming · Object-Oriented Programming Language.

1 Introduction

A Programming Language is defined by a formal grammar, however there must also be a meaning for each construct of the language. Programs have their meanings given by the semantics of their constructs which, generally, must be preserved across programs. Without the semantics of constructs, it would be difficult to verify if the code represents what it was designed to do. In general, a programming language is presented through its syntax containing some informal explanation of its semantics [27]. To the best of our knowledge, no axiomatization demonstrating the existential commitments of object-oriented (OO) constructs of a language have been presented, nor is there effort to adopt a consensual conceptualization of object-oriented constructs between languages.

Thus, in this paper we propose OOC-O, a reference ontology on Object-Oriented Code whose purpose is to identify and represent the fundamental concepts present in OO source code. This reference ontology is based on UFO [14] and was developed according to the SABiO method [11], in a modular way to foster its reuse. Ontology verification was guided by competency questions, whereas its validation consisted of both instantiating its concepts in OO code form and

by harmonizing popular OO languages using the ontology as interlanguage. The latter resulted from the ontology capture process, whose objective was to reduce semantic and syntactic conflicts between languages.

Although OOC-O is applicable in several contexts, it is being built in the context of polyglot programming, i.e., different programming languages with different constructs combined in the same software development project. If on the one hand the combination of different programming languages with specific responsibilities can reduce the effort to implement solutions [12], on the other hand, the effort to implement an algorithm may differ between programming languages depending on its constructs [24]. In this context, OOC-O has been used as support for both programmers to understand different syntaxes and semantics of object-oriented constructs, as well as for integrated development tools to interoperate different languages. The ontology has already been used to migrate classes with object/relational mappings from one language to another [30] and is currently being used in an effort to produce a unified solution for identifying smells in OO source code. Furthermore, OOC-O is part of a larger effort of creating an ontology network on software development frameworks.[1]

The remainder of this paper is organized as follows. Section 2 discusses briefly the main concepts found in most OO programming languages as well as the ontological foundations used for developing OOC-O. Section 3 presents OOC-O. Section 4 addresses ontology verification and validation. Section 5 discusses related works. Finally, Section 6 concludes the paper.

2 Baseline

Object-oriented (OO) programming is defined as a software implementation method in which programs are organized as cooperative collections of **objects**, each of which representing an instance of some **class**, and whose classes are members of a hierarchy of classes linked by **inheritance relationships**. A class serves as a template from which objects can be created. It is a defined type that determines the data structures (**attributes**) and **methods** associated with that type. In order for the attributes and methods of a class to be used in defining a new class, **inheritance** is applied as a means of creating abstractions.

Abstraction is the mechanism of representing only the essential characteristics, ignoring the irrelevant details as a way of hiding implementation. To hide data, *encapsulation* applies a packaging of methods and attributes accessible or modifiable only via the interface. Moreover, abstraction can be defined by *polymorphism*, attributing the ability to take on many forms and by *genericity*, attributing the ability to take several types independently of the structure.

Abstraction, encapsulation, inheritance and polymorphism are the main principles of object orientation [7]. In other words, if any of these elements is missing, you have something less than an OO language [5]. Thus, we consider an OO programming language as a tool that supports these four fundamental principles:

[1] `https://nemo.inf.ufes.br/projects/sfwon/`

Abstraction is realized in a OO code by means of **classes** containing attributes and methods; *Encapsulation* is implemented by **accessor methods** hiding internal information of the class, avoiding direct access to its attributes, and by **element visibility** avoiding unwanted access to these elements; *Inheritance* is directly represented as a relation between a **subclass** that inherits characteristics from a **superclass**; and, finally, *Polymorphism* takes place via the concepts of **method override**, in which a method declaration in the subclass modifies the method declared in the superclass, **abstract class**, whose abstract methods are implemented according to the subclass that inherits them, and **generic class/method**, whose definition can be used by different data types.

Considering the range of existing languages, we selected languages that provide constructs for the basic OO principles discussed above in order to form the baseline of our research, namely: Smalltalk, Eiffel, C++, Java and Python. The selection took into account the first two OO programming languages ever proposed and the three currently most popular OO languages according to the TIOBE[2] IEEE Spectrum[3] and Redmonk[4] indexes.

In order to build an ontology on OO source code, we followed a systematic approach for building ontologies named SABiO [11], a method that considers activities for the development of reference ontologies and to its implementation as operational ontologies. In this paper, we developed only the reference ontology and, therefore, only the early stages of SABiO were performed. In Purpose Identification and Requirements Elicitation, we identify the purpose and intended uses of the ontology, define its functional requirements, by means of Competency Questions, and also non-functional ones (NFRs), and decompose the ontology into appropriate modules. Ontology Capture and Formalization phase follows, aiming at objectively recording the domain conceptualization based on an ontological analysis using a foundation ontology and representing it in a graphic model.

In addition, SABiO suggests five support processes, applied as follows: Knowledge Acquisition, to gather domain knowledge reliably through specialists and bibliographic material; Reuse, to take advantage of conceptualizations already established for the domain; Documentation, to record the results of the development process by means of a Reference Ontology Specification; Configuration Management, to control changes, versions, and delivery by means of a repository; and Evaluation, to evaluate the suitability of the ontology by means of verification, ensuring that the ontology satisfies its requirements, and validation, ensuring that the ontology is able to represent real world situations.

For building our conceptual models, we used the OntoUML modeling language, which is based on the UML 2.0 class diagram and incorporates important foundational distinctions made by the Unified Foundational Ontology (UFO) [14]. Such distinctions are made explicit in the model by means of UML class stereotypes, summarized as follows: «category», a rigid type whose instances share common intrinsic properties but obey different principles of identity (non-

[2] tiobe.com, January 2019.

[3] spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages, July 2018.

[4] redmonk.com/sogrady/2019/03/20/language-rankings-1-19/, January 2019.

sortal, rigid entities); «kind», a rigid sortal type that is formed by distinct parts (functional complex) and supplies an identity principle for its instances; «subkind», a rigid sortal type whose instances inherit an identity principle from a kind; «role», an anti-rigid sortal type whose specialization condition is given by extrinsic (relational) properties; «relator», a concept connecting other concepts, and thus existentially dependent on them; and «quality», a type whose instances represent intrinsic properties of an individual associated with a quality structure. This choice is motivated by UFO having a modeling language with stereotypes covering the domain studied and the availability of an ontology network on software engineering represented in such language, facilitating integration and reuse.

3 Object-Oriented Code Ontology (OOC-O)

The Object-Oriented Code Ontology (OOC-O) aims to identify and represent the semantics of the entities present at compile time in object-oriented (OO) code. Given such scope, even though objects are the fundamental constructs in OO programming and messages are responsible for exchanges between objects, they are not covered by OOC-O, since they exist only at runtime. The intention is to use the ontology to assist the understanding of different programming languages and to support the development of tools that work with these languages, in the context of polyglot programming and object-oriented frameworks.

We elicited the following non-functional requirements for OOC-O: NFR1 – be modular or embedded in a modular framework to facilitate reuse of other ontologies and, consequently, its own reuse by other ontologies; and NFR2 – be based on well-known sources from the literature. In response to **NFR1** and to facilitate viewing, we decomposed the ontology into three modules, namely: OOC-O Core (an overview of the main concepts), OOC-O Class (detailing concepts derived from Class) and OOC-O Class Members (detailing concepts derived from Class Members, i.e., Methods and Attributes). Moreover, we integrated OOC-O into the Software Engineering Ontology Network (SEON) [23], to reuse relevant concepts, as well as SEON's grounding in UFO. Two ontologies from SEON were reused: the Software Process Ontology (SPO) [19] and the Software Ontology (SwO) [8]. Along the paper, fragments of these reused ontologies in OOC-O are preceded by the corresponding acronyms (SPO:: and SwO::, respectively) and highlighted using different colors. Regarding **NFR2**, ontology capture was supported by a process of knowledge acquisition that used consolidated sources of knowledge referring to the five programming languages selected in this research, including books [15, 25, 18, 17, 22, 28] and standards [13, 9].

For functional requirements, we have iteratively defined twenty five competency questions (CQs) detailed in OOC-O's Reference Ontology Specification document [2], for instance: **CQ1:** What makes up an OO source code? **CQ2:** What is the visibility of an element present in an OO source code? **CQ3:** How are classes logically organized in an OO source code? **CQ4:** What elements compose a class? **CQ5:** Which are the parent classes of a class? **CQ6:** What is a root class? **CQ7:** What are the variables of a method? **CQ8:** What is the mutability

of a variable? **CQ9:** What types of classes are present in an OO source code? **CQ10:** What types of methods are present in an OO source code?

During ontology capture and formalization, we performed ontological analysis based on UFO, representing OOC-O in OntoUML. Such process was conducted iteratively, in order to address different aspects/refinements at each iteration, and interactively, so domain experts and ontology engineers could discuss the conceptualization of the domain in OntoUML. Finally, to ensure consensual understanding of the domain, the concepts were defined in a dictionary of terms and mapped to the concepts of each selected programming language, detailed in a technical report [1]. In what follows, we present the three modules of OOC-O. More details of the ontology can be found in its specification document [2].

3.1 OOC-O Core Module

Figure 1 shows the core concepts of OOC-O and how they integrate with SEON through the SPO and SwO ontologies.

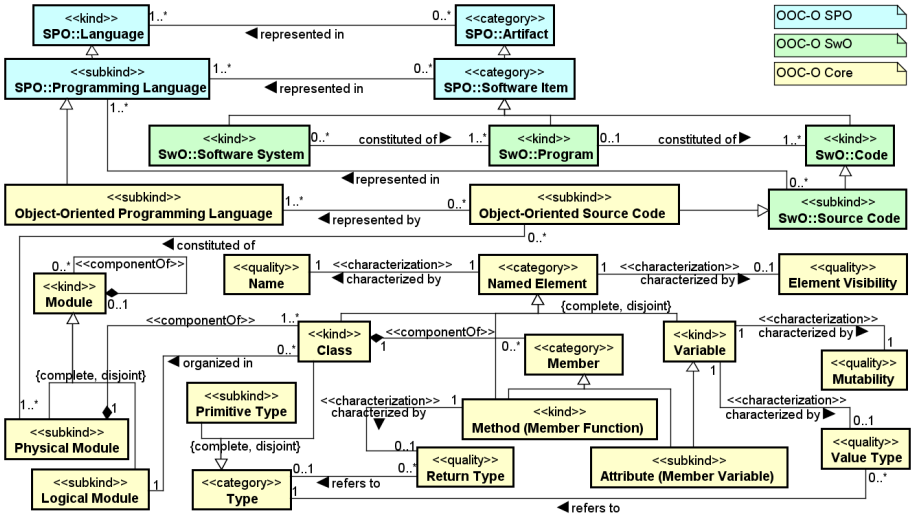

Fig. 1. Object-Oriented Code Ontology: Core module.

SPO establishes a common conceptualization on the software process domain (processes, activities, resources, people, artifacts, procedures, etc.). We reuse the concept of software **Artifact**, object consumed or produced during the software process, which is *represented in* a **Language**, a set of symbols used for encoding and decoding information. A software artifact can be, among other things, a **Software Item** such as a piece of software produced during the software process.

SwO further specializes this concept: a **Software System** is a Software Item that aims at satisfying a system specification. It is *constituted of* **Programs**,

which are Software Items that aim at producing a certain result through execution on a computer, in a particular way, given by a program specification. In turn, Programs are *constituted of* **Code**, a Software Item representing a set of computer instructions and data definitions which are *represented in* a **Programming Language** as a **Source Code**.

OOC-O is anchored in the concept of **Object-Oriented Source Code**, a Source Code specialization *represented in* an **Object-Oriented Programming Language**. Such code is *constituted of* **Physical Modules**, i.e., physical units in which the physical files (ex: `.java`) are stored (e.g., a directory in the file system). Physical Modules are *composed of* **Classes** *organized in* **Logical Modules**, i.e., packages or namespaces that group classes and allow programmers to control dependencies, visibility, etc. Both **Modules** (Physical or Logical) can be *decomposed in* their respective sub-Modules. However, decomposition can only take place among modules of the same type, i.e., $\forall m_1, m_2 : Module, PhysicalModule(m_1) \wedge componentOf(m_1, m_2) \rightarrow PhysicalModule(m_2)$ (**A1**) and $\forall m_1, m_2 : Module$, $LogicalModule(m_1) \wedge componentOf(m_1, m_2) \rightarrow LogicalModule(m_2)$ (**A2**).

Classes are *composed of* **Members**, be it a **Method (Member Function)**, function that belongs to the class and provides a way to define the behavior of an object, being invoked when a message is received by the object [18]; or be it an **Attribute (Member Variable)**, variable that belongs to the class and provides a way to define the state of its objects. Classes, Methods and Variables are **Named Elements** *characterized by* a unique **Name** and a **Visibility**, which defines the access type to the element. Attribute is a subtype of **Variable**, item of information located in the memory whose assigned value can be changed or not according to its **Mutability**. Analogously, a Method has a **Return Type**, whose values *refer to* the **Types** of information that the language is capable of manipulating, whether a **Primitive Type**, predefined by the language through a reserved word; or a Class, predefined or not.

3.2 OOC-O Class Module

The purpose of the OOC-O Class module is to represent the relevant concepts present in OO programming languages with respect to classes. Hence, OOC-O Class module, shown in Figure 2, is centered on the Class concept already presented in OOC-O Core earlier.

Every Class must either be a **Concrete Class**, implemented class that can and intends to have instances, or an **Abstract Class**, incompletely implemented class whose descendants will use as a basis for further refinement [9]. Abstract class, in contrast to Concrete Class, should not have instances and should be an Extendable Class. Further, every class must be either an **Extendable Class**, class available to be extended through Inheritance, or **Non-Extendable Class**, the opposite.

An Extendable Class can assume the **Superclass** role when relating to a Class that assumes the **Subclass** role in an **Inheritance** relationship: $\forall c_1, c_2 : Class, i : Inheritance, inheritsIn(c_1, i) \wedge inheritedFrom(c_2, i) \rightarrow subClassOf(c_1, c_2)$ (**A3**).

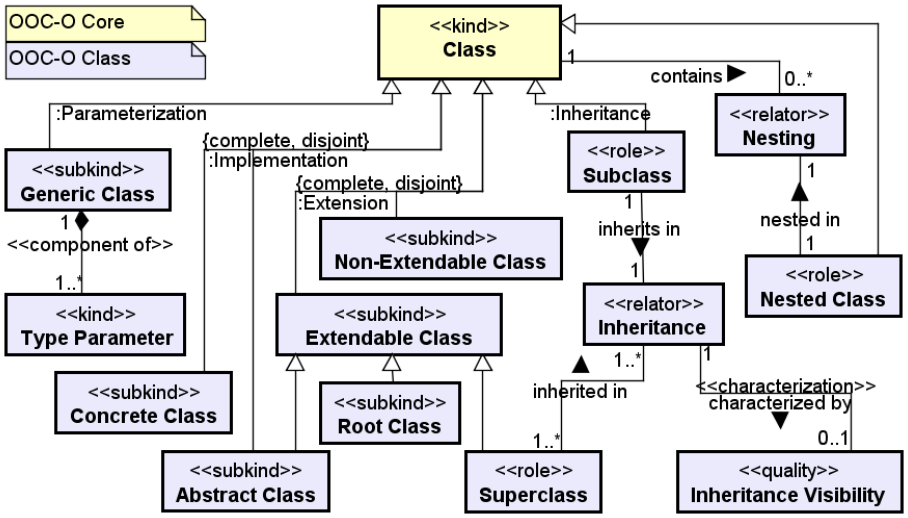

Fig. 2. Object-Oriented Code Ontology: Class module.

The relationship between a Superclass and a Subclass is established mainly by the existence of a "is-a" relation between them [26].

In this context, **Inheritance Visibility** can be set to limit the Subclass permission on the members of the Superclass. The Extendable Class inherited by all classes directly or indirectly in an OO code is known as **Root Class** [9] and introduces several general-purpose resources. When present, the Root Class is a common ancestor for all other existing classes, i.e., $\forall c : Class, r : RootClass, c \neq r \rightarrow descendantOf(c, r)$ (**A4**), where $descendantOf$ is defined in terms of the $subClassOf$ predicate introduced above, according to the following axioms: $\forall c_1, c_2 : Class, subclassOf(c_1, c_2) \rightarrow descendantOf(c_1, c_2)$ (**A5**) and $\forall c_1, c_2, c_3 : Class, subclassOf(c_1, c_2) \wedge descendantOf(c_2, c_3) \rightarrow descendantOf(c_1, c_3)$ (**A6**).

Finally, a Class can also assume the **Nested Class** role when relating to another Class by means of its declaration being within the body of that Class [13] (we refer to this as **Nesting**). Furthermore, a Class can be a **Generic Class**, when it describes a template for a possible set of types [9]. A Generic Class is *composed of* **Type Parameters**, which are identifiers that specify generic type names whose instances must define recognized types that will replace the Type Parameter at runtime.

3.3 OOC-O Class Members Module

The purpose of the OOC-O Class Members module is to represent the relevant concepts present in OO programming languages with respect to the component members of the classes. As methods and attributes are the key components of a class, OOC-O Class Members module, shown in Figure 3, is centered on the

concepts of Method (Member Function) and Attribute (Member Variable) already presented in OOC-O Core.

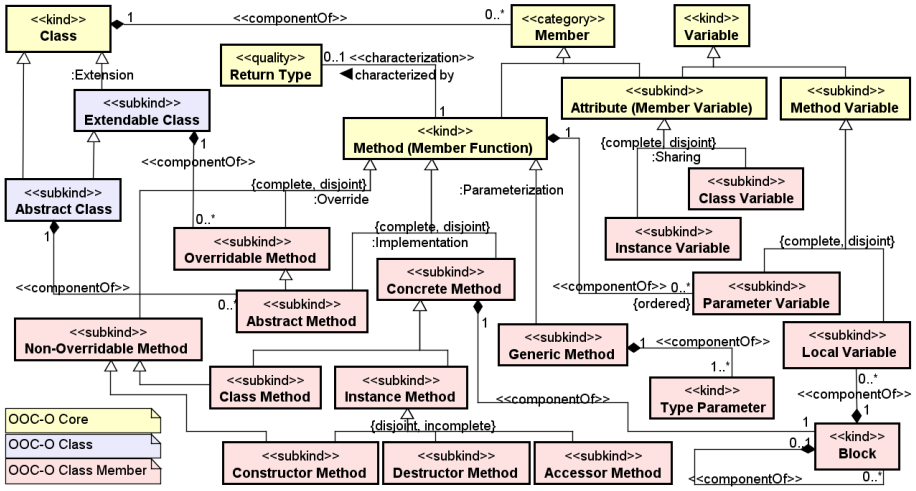

Fig. 3. Object-Oriented Code Ontology: Class Members.

Every Method of a Class must be either a **Concrete Method**, implemented in its own (concrete or abstract) Class by means of Blocks; or an **Abstract Method**, belonging to an Abstract Class and implemented (or "made concrete") only in its Subclasses. A Concrete Method can be specialized according to its execution context, either in the context of the class, invoked by the class in a **Class Method**, or in the context of the object, invoked by the object in an **Instance Method**. An Instance Method can be specialized in **Accessor Method**, which provides an interface between the internal data of the object and the external world [15], in **Constructor Method**, which specifies how an object should be created and initialized, or in **Destructor Method**, which is responsible for cleaning unusable objects. Return Type cannot characterize neither a Constructor nor a Destructor Method: $\forall rt : ReturnType, m : Method, characterization(rt, m) \rightarrow \neg ConstructorMethod(m) \wedge \neg DestructorMethod(m)$ (**A7**).

Further, every Method must be either an **Overridable Method**, method belonging to an Extendable Class that can be overwritten in descendant classes [25], such as an Abstract Method declared in an Abstract Class to be implemented by Subclasses; or a **Non-Overridable Method**, method that can be inherited but is not allowed to be overwritten in descendant classes, such as Class Methods and Constructor Methods. A Method can also be a **Generic Method** when describing a template for a possible set of methods *composed of* one or more **Type Parameters**.

Variables, in turn, can be associated with methods, i.e., be a **Method Variable**, or classes, i.e., an **Attribute (Member Variable)**. In an indirect way, Method Variable is member of a Class, since a Class is *composed of* Methods. Method Variable can be a **Parameter Variable** declared within the signature of a Method or **Local Variable** declared within a Block. Part-of relations among Methods, Blocks and Local Variables are transitive in the following ways: $\forall v : LocalVariable, b_1, b_2 : Block, componentOf(v, b_1) \wedge componentOf(b_1, b_2) \rightarrow componentOf(v, b_2)$ (**A8**) and $\forall v : LocalVariable, b : Block, m : ConcreteMethod, componentOf(v, b) \wedge componentOf(b, m) \rightarrow componentOf(v, m)$ (**A9**). An Attribute can be a **Class Variable** when shared by all objects of the Class or an **Instance Variable** when it represents the particular state of each object.

4 Evaluation

The evaluation of a reference ontology comprises activities of verification and validation. For ontology **verification**, SABiO suggests identifying whether the elements that make up the ontology are able to answer the competency questions raised. Table 1 presents the results for some of the raised CQs (cf. Section 3), showing which concepts and relations are used to answer a CQ.

Table 1. Results for OOC-O verification.

ID	Competency Question	Axiom
CQ1	Object-Oriented Source Code *constituted of* Physical Module; Class *component of* Physical Module.	
CQ2	Named Element *characterized by* Element Visibility	
CQ3	Class *organized in* Logical Module	
CQ4	Member *component of* Class; Attribute (Member Variable) and Method (Member Function) *subtype of* Member	
CQ5	Subclass *subtype of* Class; Subclass *inherits in* Inheritance; Superclass *inherited in* Inheritance	A3, A5, A6
CQ6	Extendable Class *subtype of* Class; Root Class *subtype of* Extendable Class; Subclass *subtype of* Class; Subclass *inherits in* Inheritance; Superclass *inherited in* Inheritance.	A3, A4, A5, A6
CQ7	Parameter Variable *component of* Method; Local Variable *component of* Block; Block *component of* Block; Block *component of* Concrete Method; Concrete Method *subtype of* Method	A8, A9
CQ8	Variable *characterized by* Mutability	
CQ9	Generic Class *subtype of* Class; Concrete Class *subtype of* Class; Abstract Class *subtype of* Class; Non-Extendable Class *subtype of* Class; Extendable Class *subtype of* Class	
CQ10	Generic Method *subtype of* Method; Concrete Method *subtype of* Method; Abstract Method *subtype of* Method; Overridable Method *subtype of* Method; Non-Overridable Method *subtype of* Method	

For ontology **validation**, the ontology should be instantiated to check if it is able to represent real world situations. For this, we use the same OO code fragment written in the selected languages to instantiate the concepts of the ontology. Table 2 shows some results of the OOC-O instantiation. It is worthy to say that since there are orthogonal generalization sets that are disjoint and complete (e.g., *:Implementation* and *:Extension* in Class concept), each concept instance

Table 2. Results for OOC-O instantiation.

Language Code	**OOC-O Instance**
Smalltalk Code	Polygon = Concrete Class & Extendable Class & Subclass
`Object subclass: #Polygon`	Object = Superclass & Root Class
`  instanceVariableNames: 'side'`	side = Instance Variable
`  perimeter ...`	perimeter = Instance Method & Overridable Method
Eiffel Code	
`class Polygon`	Polygon = Concrete Class & Extendable Class & Subclass
`feature{ANY}`	side = Instance Variable
`  perimeter() is  do ... end`	INTEGER = Value Type
`feature{NONE}`	perimeter = Instance Method & Non-Overridable Method
`  side : INTEGER`	NONE and ANY = Element Visibility
`end`	
C++ Code	Polygon = Concrete Class & Extendable Class
`class Polygon{`	side = Instance Variable
`  private: int side;`	perimeter = Instance Method
`  public: void perimeter(){};`	private and public = Element Visibility
`};`	void and int = Value Type
Java Code	Polygon = Concrete Class & Extendable Class & Subclass
`public class Polygon{`	side = Instance Variable & Overridable Method
`  private int side;`	perimeter = Instance Method
`  public void perimeter(){};`	private and public = Element Visibility
`}`	void and int = Value Type
Python Code	Polygon = Concrete Class & Extendable Class & Subclass
`class Polygon:`	side = Instance Variable
`  side = None`	None = Initial Variable Value
`  def perimeter(): ...`	perimeter = Concrete Method & Overridable Method

(e.g., the `Polygon` class) is classified in at least each of these generalization sets (e.g., Concrete Class or Abstract Class, and Extendable Class or Non-Extendable Class). The complete table is available in a technical report [1].

From OOC-O's instantiation we can see that the code relative to class definition incorporates the semantics of **concrete** and **extendable** class in the ontology. Most languages, explicitly (Smalltalk) or implicitly (Eiffel, Java and Python), incorporates **subclass** semantics, since all classes are subclasses of the **root** class of these languages such as the `Object` class in Smalltalk, Java and Python, and the `Any` class in Eiffel (C++ does not have a root class). Code relative to method definition in different languages incorporates a highly variable semantics, including the semantics of **instance**, **concrete**, **overridable** and **non-overridable** methods in the ontology. The **element visibility** is either explicitly defined with keywords (`private` and `public` in Java and C++, and `none` and `any` in Eiffel) or is private by default (Smalltalk) or is public by default (Python). The **value type** is explicitly defined in some languages (Eiffel, C++, Java) and defined by the assigned value (Python) or defined as an object (Smalltalk) in others.

We also performed a harmonization between the elements of the selected languages and the concepts of OOC-O, applying equivalence relations. Table 3 shows some of these matches and the complete table is available in a technical report [1]. Although the OO principles are well established, the way they are handled in the programming languages is not uniform. Each language adopts different syntax and semantics for their constructs, resulting in different levels in which those principles are addressed. In this context, OOC-O can be used to support interoperability among them.

Table 3. Equivalence between selected OO programming languages and OOC-O.

Lang.	Language concept	OOC-O concept
Smalltalk	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Template	Generic Class
	Method	Concrete Method & Overridable Method
	Accessor Method	Accessor Method
	Instance Variable	Instance Variable
	Access	Element Visibility
Eiffel	Class	Concrete Class & Extendable Class
	Deferred Class	Abstract Class
	Frozen Class	Non-Extendable Class
	Generic Class	Generic Class
	Routine	Instance Method & Non-Overridable Method
	Routine Redefinition	Overridable Method
	Accessor Routine	Instance Method
	Attribute	Instance Variable
	Access	Element Visibility
C++	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Final Class	Non-Extendable Class
	Template	Generic Class
	Member Function	Instance Method
	Final Member Function	Non-Overridable Method
	Virtual Member Function	Overridable Method
	Accessor Member Function	Instance Method
	Instance Variable	Instance Variable
	Access Modifier	Element Visibility
Java	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Final Class	Non-Extendable Class
	Generic Class	Generic Class
	Method	Instance Method & Overridable Method
	Abstract Method	Abstract Method
	Final Method	Non-Overridable Method
	Accessor Method	Instance Method
	Instance Variable	Instance Variable
	Access Modifier	Element Visibility
Python	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Generic Class	Generic Class
	Method	Concrete Method & Overridable Method
	Accessor Method	Instance Method
	Data Attribute	Instance Variable

Therefore, **Abstraction** is represented by the class concept in the languages, being composed by members such as method in Smalltalk, Java and Python, or routine in Eiffel, or member function in C++, and by attribute in Eiffel, or data attribute in Python, or instance variable in Smalltalk, C++ and Java. **Inheritance** is represented by subclass in Smalltalk, Eiffel, Java and Python, or derived class in C++, and by superclass in Smalltalk, Eiffel, Java and Python, or base class in C++. **Encapsulation** is represented by access in Smalltalk and Eiffel, or access modifier in C++ and Java, and by the public visibility in Python. Encapsulation is represented also by accessor method in Smalltalk, however, the accessor method concept in Eiffel, C++, Java and Python is not equivalent to accessor method in the ontology because in these languages there is only a convention for treating an instance method as an accessor method. **Polymorphism** is represented by routine redefinition in Eiffel or virtual function in C++. Smalltalk, Java and

Python incorporate the semantics of overridable method to the method concept. Polymorphism is represented also by generic class/method in Eiffel, Java and Python, or template in Smalltalk and C++. Polymorphism is represented also by abstract class in Smalltalk, C++, Java and Python, or deferred class in Eiffel.

Finally, in a separate research effort [30], the OOC-O reference ontology presented in this paper was implemented in OWL, giving rise to its operational version OOC-OWL (also available in the aforementioned website). OOC-OWL was then used by ORM-OWL (Object/Relational Mapping Ontology) to instantiate source code with ORM annotations and migrate it from one language/framework to another using the ontology as an interlingua.

5 Related Works

Concepts that were originally developed by OO programming languages have appeared in many other areas such as database [3], development methodology [21], data analysis [6], and others. Therefore, there are several works that discuss and formalize fundamentals of programming language, discussing semantic theories to be applied in the definition of programming languages ontologies [27], or of the object orientation, using the ontological view to define the formal basis of the object notion [29] or introducing a new view on the roles in OO programming languages, such in the powerJava language extended from Java [4]. However, this research is interested in identifying and formalizing the relevant concepts in OO programming languages, little explored as far as we know.

Evermann & Wand [10] apply semantic mapping between ontological concepts of the BWW ontology and OO programming language constructs to assign semantics and rules in the context of software modeling. The BWW concepts (thing, property and functional schema) are mapped to UML concepts (object, class, attribute, attribute of 'ordinary' class and attribute of association class). Although the research has applied an ontological analysis to map object-oriented constructs, it covers only a small portion of that domain.

Kouneli et al. [16] apply an operational ontology of programming language for representing the knowledge delivered by a distance learning course on computer programming. Although the ontology is built following a methodology and sources of information of the Java language, it is not based on any foundation ontology. The concepts of the ontology are anchored in the Thing concept and hierarchically organized from Java Element (Class, Constructor, Data Type, Exception, Interface, Method (AbstractMethod, FinalMethod, ClassMethod and InstanceMethod), Object, Operator, Package, Statement, Thread and Variable (ClassVariable, InstanceVariable, LocalVariable and Parameter)), Keyword and Literal Value. Unlike OOC-O, this ontology only represents the Java programming language domain and incorporates non-object-oriented concepts such as exception, operator, statement, and thread.

Pastor et al. [20] elaborate the O3 reference ontology, inspired by BWW and the FRISCO framework to semantically map the concepts of the OO programming paradigm. The concepts of BWW (thing, property, substantial and

relation) are specialized for the concepts of the OO paradigm (class (generalization, specialization), domain (primitive type)), attribute (variable, constant), interface). Although the research has applied an ontological analysis to map object-orientation concepts, it covers only a small portion of that domain and incorporates non-object-oriented concepts such as constraint, service, relation, agent, server and others.

6 Final Considerations

This paper presents a reference ontology about the concepts of object-oriented programming code based on a foundation ontology. The OOC-O ontology is built according to an ontology engineering method and based on well-known data sources. Verification and validation activities were successfully accomplished, by answering competency questions, instantiating the ontology in fragments of OO code, harmonizing between object-oriented languages (Smalltalk, Eiffel, C++, Java and Python) and checking the coverage of the fundamental OO concepts.

The OOC-O ontology is not intended to represent principles or philosophies of object orientation, but rather the semantic representation of OO programming language code. To the best of our knowledge, we have not found any related work that covers the proposed domain in depth. Finally, in future work, we intend to use the ontology at the foundation of tools in a polyglot programming development environment and in the context of semantic interoperability among different object-oriented frameworks.

References

1. Aguiar, C.Z.: Object-Oriented Code Ontology — Harmonization Document of Object-Oriented Programming Language, available on: `http://nemo.inf.ufes.br/projects/sfwon/`. Tech. rep., Federal University of Espírito Santo (2019)
2. Aguiar, C.Z.: Object-Oriented Code Ontology — Reference Ontology Specification Document, available on: `http://nemo.inf.ufes.br/projects/sfwon/`. Tech. rep., Federal University of Espírito Santo (2019)
3. Atkinson, M., Dewitt, D., Maier, D., Bancilhon, F., Dittrich, K., Zdonik, S.: The object-oriented database system manifesto. In: Deductive and object-oriented databases, pp. 223–240. Elsevier (1990)
4. Baldoni, M., Boella, G., Van Der Torre, L.: powerjava: ontologically founded roles in object oriented programming languages. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1414–1418. ACM (2006)
5. Booch, G.: Coming of age in an object-oriented world. IEEE Software **11**(6), 33–41 (1994)
6. Brun, R., Rademakers, F.: Root—an object oriented data analysis framework. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**(1-2), 81–86 (1997)
7. Conaway, C.F., Page-Jones, M., Constantine, L.L.: Fundamentals of object-oriented design in UML. Addison-Wesley Professional (2000)

8. Duarte, B.B., Leal, A.L.C., Falbo, R.d.A., Guizzardi, G., Guizzardi, R.S., Souza, V.E.S.: Ontological foundations for software requirements with a focus on requirements at runtime. Applied Ontology **13**(2), 73–105 (may 2018). https://doi.org/10.3233/AO-180197
9. Eiffel, E.: Eiffel: Analysis, design and programming language. ECMA Standard ECMA-367, ECMA (2006)
10. Evermann, J., Wand, Y.: Ontology based object-oriented domain modelling: fundamental concepts. Requirements engineering **10**(2), 146–160 (2005)
11. Falbo, R.A.: Sabio: Systematic approach for building ontologies. In: ONTO. COM/ODISE@ FOIS (2014)
12. Fjeldberg, H.C.: Polyglot programming. Ph.D. thesis, Master thesis, Norwegian University of Science and Technology, Trondheim/Norway (2008)
13. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D.: The java language specification: Java se 10 edition, 20 february 2018 (2018)
14. Guizzardi, G., Wagner, G.: A unified foundational ontology and some applications of it in business modeling. In: CAiSE Workshops (3). pp. 129–143 (2004)
15. Hunt, J.: Java and object orientation: an introduction. Springer Science & Business Media (2002)
16. Kouneli, A., Solomou, G., Pierrakeas, C., Kameas, A.: Modeling the knowledge domain of the java programming language as an ontology. In: International Conference on Web-Based Learning. pp. 152–159. Springer (2012)
17. Lafore, R.: Object-oriented programming in C++. Pearson Education (1997)
18. LaLonde, W.R., Pugh, J.R.: Inside smalltalk, vol. 2. Prentice Hall (1990)
19. de Oliveira Bringuente, A.C., de Almeida Falbo, R., Guizzardi, G.: Using a foundational ontology for reengineering a software process ontology. Journal of Information and Data Management **2**(3), 511 (2011)
20. Pastor, O.: Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orientado a Objetos. Ph.D. thesis, Tesis doctoral dirigida por Isidro Ramos, DSIC, Universitat Politècnica de ... (1992)
21. Pastor, O., Insfrán, E., Pelechano, V., Ramirez, S.: Linking Object-Oriented Conceptual Modeling with Object-Oriented Implementation in Java. In: Database and Expert Systems Applications, 8th International Conference, DEXA'97, Toulouse, France, September 1-5, 1997, Proceedings. pp. 132–141 (1997)
22. Phillips, D.: Python 3 object-oriented programming. Packt Publishing Ltd (2015)
23. Ruy, F.B., de Almeida Falbo, R., Barcellos, M.P., Costa, S.D., Guizzardi, G.: Seon: A software engineering ontology network. In: European Knowledge Acquisition Workshop. pp. 527–542. Springer (2016)
24. Schink, H., Broneske, D., Schröter, R., Fenske, W.: A tree-based approach to support refactoring in multi-language software applications. In: Proceedings of the 2nd International Conference on Advances and Trends in Software Engineering, Lisbon, Portugal. pp. 3–6 (2016)
25. Sebesta, R.W.: Concepts of programming languages. Boston: Pearson, (2012)
26. Tucker, A.B.: Programming languages> Principles and Paradigmas. Tata McGraw-Hill Education (2007)
27. Turner, R., Eden, A.H.: Towards a programming language ontology. Citeseer (2007)
28. Tyrrell, A., Tyrrell, A.: Eiffel Object-oriented Programming. Springer (1995)
29. Wand, Y.: A proposal for a formal model of objects. In: Object-oriented concepts, databases, and applications. pp. 537–559. ACM (1989)
30. Zanetti, F., Aguiar, C.Z., Souza, V.E.S.: Representacao ontologica de frameworks de mapeamento objeto/relacional. In: 12th Seminar on Ontology Research in Brazil (ONTOBRAS), *to appear* (2019)