Reference Ontology Specification Document

Ontology: Object-Oriented Code Ontology (OOC-O)

1. Introduction

This document presents the requirements of the Object Oriented Code Ontology and is organized as follows: Section 2 contains the ontology purpose and its intended uses; Section 3 presents the domain for which the ontology is being constructed; Section 4 presents the reference ontology itself, including architecture (modularization) and description of the competency questions, OntoUML conceptual models, axioms (informal and formal), dictionary of terms and evaluation of the each sub-ontology considered in the architecture.

2. Purpose and Intended Uses of Ontology

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. That is, we intend to apply the ontology as a meta language of OO programming languages in a polyglot programming development environment and in the context of semantic interoperability among object/relational mapping frameworks.

3. Domain Description

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 contained constructs, and, generally, such semantics must preserve the meanings of the constructs across programs [1]. 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 [2]. To the best of 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.

Object-oriented (OO) programming is dened 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.

4. Reference Ontology

This section introduces the Object-Oriented Code Ontology (OOC-O). The subsections present their competency questions, conceptual model in OntoUML, axioms, dictionary of terms and preliminary evaluation of the ontology.

4.1. Modularization

Figure 4.1.1 shows the sub-ontologies identified in this context, which are described in Table 4.1.1.

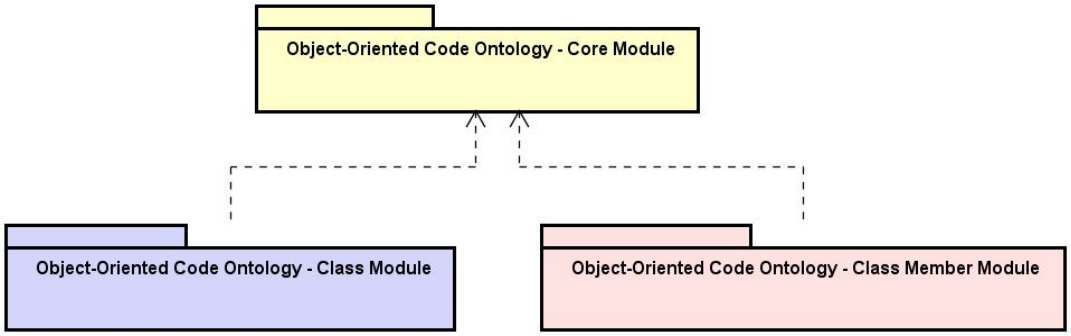

Figure 4.1.1 Object-Oriented Code Ontology Architecture.

Table 4.1.1 Sub-ontologies of the OOC-O.

Sub-ontology	Description
Object-Oriented Code Ontology Core Module	Ontology that describes the fundamental concepts of code in object-oriented programming languages
Object-Oriented Code Ontology Class Module	Ontology that describes the fundamental concepts related to class in object-oriented programming languages
Object-Oriented Code Ontology Class Member Module	Ontology that describes the fundamental concepts related to member (attribute and method) of the class in object-oriented programming languages

4.2. Object-Oriented Code Ontology - Core Module

4.2.1. Competency Question

For functional requirements, we have iteratively defined the competency questions (CQs) presented in Table 4.2.1.

Table 4.2.1 OOC-O Core Module competency question.

ID	Competency Question
CQ1	How an OO source code is represented?
CQ2	What makes up an OO source code?
CQ3	How are classes logically organized in an OO source code?
CQ4	What is the visibility of an element present in an OO source code?
CQ5	What is the identifier of an element present in an OO source code?
CQ6	What elements compose a class?
CQ7	What is the return type of a method?
CQ8	What is the value type of a variable?
CQ9	What is the mutability of a variable?
CQ10	What types makes up an OO source code?

4.2.2. Conceptual Model

The OntoUML diagram is presented in Figure 4.2.1 and the definitions of their terms is presented in the Dictionary of Terms (Table 4.2.2).

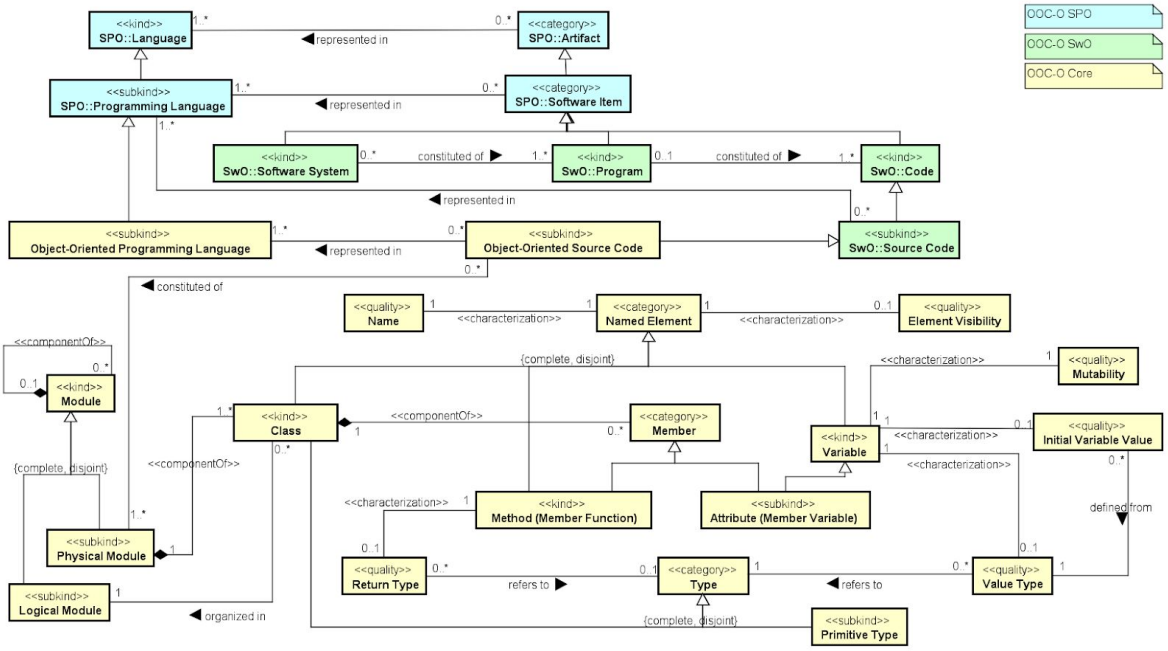

Figure 4.2.1 OntoUML Diagram of the OOC-O Core Module.

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: ∀ m1;m2 : Module; PhysicalModule(m1) ^ componentOf(m1;m2) ! PhysicalModule(m2) (A1) and ∀ m1;m2 : Module; LogicalModule(m1) ^ componentOf(m1;m2) ! LogicalModule(m2) (A2).

Classes are composed of Members, be it a Method (Member Function), function that belongs to the class and provides a way to dene the behavior of an object, being invoked when a message is received by the object [3]; or be it an Attribute (Member

Variable), variable that belongs to the class and provides a way to dene the state of its objects. Classes, Methods and Variables are Named Elements characterized by a unique Name and a Visibility, which denes 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.

4.2.3. Dictionary of Terms

The definition of the terms present in OOC-O Core Module is presented in Table 4.2.2.

Table 4.2.2 Dictionary of Terms of the OOC-O Core Module.

Term	Definition
Attribute (Member Variable)	Variable that belongs to a class.
Class	An abstract-definition element in the OO programming language to express such definitions [5], that is, class is an abstract data type and a mechanism for defining an abstract data type in a program [4]. Class describes the attributes of its objects as well as the methods they can execute [6].
Element Visibility	Access type defined to control the level of visibility of variables, methods, and classes [4]
Initial Variable Value	Initial value set in the declaration of a variable or defined by a default value that depends on its data type [6].
Logical Module	Logical unit in which physical files are organized by language.
Member	Element which makes up a class, defined as a variable or method.
Method (Member Function)	Function of a class that provides a way to define the behavior of a object [7], invoked when a message is received by the object [3]. Every method is created in memory only once so that objects share their instance. Every method is composed of a name, zero or more parameters and a return.
Module	Unit of organization of the physical files of a program.
Mutability	Characteristic that allows or not the content or state of a variable to be changed after it has been initialized.
Name	Unique identifier of object-oriented elements
Named Element	Elements that are defined by an name identifier and that stores values, functions, or classes [8]
Physical Module	Physical unit in which the physical files (ex: .java) are stored by language. The operating system does not allow two files with the same name in the same file directory, so it is necessary to organize the files in different modules.

Primitive Type	Type that is pre-defined by the language and named by its reserved word [9]. Primitive values do not share status with other primitive values.
Return Type	Value type returned by the method. Some methods can only act on the object's state and do not pass a return value to the program [5]
Type	Type of information that the language is capable of handling, composed of a domain of possible values and a set of possible operations that can be performed on those values [10]. Type is classified in Class and Primitive.
Variable	Information item located in specific memory, identified by a symbolic name. Every variable has an value, which is the value stored in the memory location that it represents [6]. In statically typed languages, variables are declared with an associated type [9], but in dynamic languages, no.
Value Type	In statically typed languages, variables are declared with an associated type [9], in dynamic languages, no.

4.2.4. Ontology Verification

The Table 4.2.3 presents the verification of competency questions listing necessary elements of the ontology to answer each competency question (CQ).

Table 4.2.3 Verification of the OOC-O Core Module.

ID	Competency Question
CQ1	**How an OO source code is represented?** Object-Oriented Source Code *represented in* Object-Oriented Programming Language.
CQ2	**What makes up an OO source code?** Object-Oriented Source Code *constituted of* Physical Module; Class *component of* Physical Module; Member *component of* Class; Method (Member Function) and Attribute (Member Variable) *subtype of* Member.
CQ3	**How are classes logically organized in an OO source code?** Class *organized in* Logical Module.
CQ4	**What is the visibility of an element present in an OO source code?** Named Element *characterized by* Element Visibility.
CQ5	**What is the identifier of an element present in an OO source code?** Named Element *characterized by* Name.
CQ6	**What elements compose a class?** Member *component of* Class; Attribute (Member Variable) and Method (Member Function) *subtype of* Member.
CQ7	**What is the return type of a method?** Method (Member Function) *characterized by* Return Type.

CQ8	**What is the value type of a variable?** Variable *characterized by* Value Type.
CQ9	**What is the mutability of a variable?** Variable *characterized by* Mutability.
CQ10	**What types makes up an OO source code?** Class and Primitive Type *subtype of* Type.

4.2.5. Ontology Validation

The Table 4.2.4 presents the validation of the ontology instantiating their concepts. For this we use instances of the Java programming language.

Table 4.2.4 Validation of the OOC-O Core Module.

Term	Definition

Attribute (Member Variable)	**width** in:
	public class Rectangle{
	  private int width;
	}

Class	**Rectangle** in: public class Rectangle{}
Element Visibility	**public** in: public class Rectangle{}
Logical Module	No Related to Java Language

Member	**width** in:
	public class Rectangle{
	  private int width;
	}

Method (Member Function)	**perimeter** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Module	**com.shapes** in: package com.shapes; public class Rectangle{}

Mutability	**angle** in:
	public class Rectangle{
	  public final double angle = 90;
	}

Name	**Rectangle** in: public class Rectangle{}
Named Element	**Rectangle** in: public class Rectangle{}

Physical Module	**com.shapes** in: package com.shapes; public class Rectangle{}

Primitive Type	**int** in:
	public class Rectangle{
	  private int width;
	}

Return Type	**int** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Type	**int** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Variable	**width** in:
	public class Rectangle{
	  private int width;
	}

Value Type	**int** in:
	public class Rectangle{
	  private int width;
	}

4.3. Object-Oriented Code Ontology - Class Module

4.3.1. Competency Question

For functional requirements, we have iteratively defined the competency questions (CQs) presented in Table 4.3.1.

Table 4.3.1 Competency question of the OOC-O Class Module.

ID	Competency Question
CQ1	What types of classes are present in an OO source code?
CQ2	What is a root class?
CQ3	What are type parameter of a generic class?
CQ4	What are the superclasses of a class?
CQ5	What is the visibility of an inheritance?
CQ6	What classes are present into a class?

4.3.2. Conceptual Model

The OntoUML diagram is presented in Figure 4.3.1 and the definitions of their terms is presented in the Dictionary of Terms (Table 4.3.2).

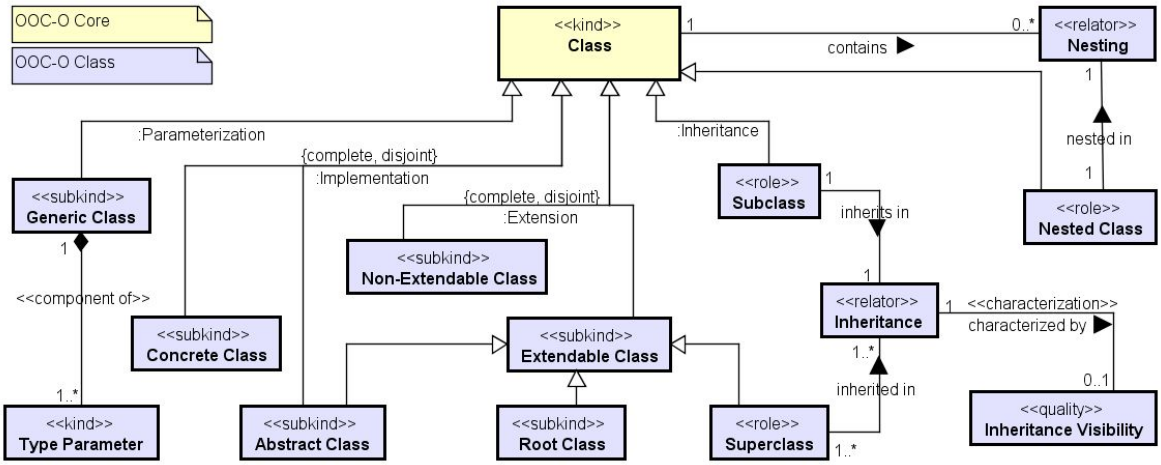

Figure 4.3.1 OntoUML Diagram of the OOC-O Class Module.

Every Class must either be a Concrete Class, implemented class that can and intends to have instances [11], or an Abstract Class, incompletely implemented class whose descendants will use as a basis for further refinement [12]. 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: ∀ c1; c2 : Class; i : Inheritance; inheritsIn(c1; i)^inheritedF rom(c2; i) ! subClassOf(c1; c2) (A3). The relationship between a Superclass and a Subclass is established mainly by the existence of a is-a relation between them [4].

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 [12] and introduces several general-purpose resources. When present, the Root Class is a common ancestor for all other existing classes, i.e., ∀ c : Class; r : RootClass; c ≠ r ! descendantOf(c; r) (A4), where descendantOf is defined in terms of the subClassOf predicate introduced above, according to the following axioms: ∀ c1; c2 : Class; subclassOf(c1; c2) ! descendantOf(c1; c2) (A5) and ∀ c1; c2; c3 : Class; subclassOf(c1; c2) ^ subClassOf(c2; c3) ! descendantOf(c1; c3) (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 [9] (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 [11]. A Generic Class is composed of Type Parameters, which are

identifiers that specify generic type names whose instances must dene recognized types that will replace the Type Parameter at runtime.

4.3.3. Dictionary of Terms

The definition of the terms present in OOC-O Class Module is presented in Table 4.3.2.

Table 4.3.2 Dictionary of Terms of the OOC-O Class Module.

Term	Definition
Abstract Class	Class describing an incompletely implemented abstraction, whose descendants will use as the basis for further refinement [12].
Concrete Class	Class describing an completely implemented abstraction.
Extendable Class	Class that can be extended by descendant classes, that is, it can be inherited by other classes.
Generic Class	Parameterized class that does not describe a type but a template for a possible set of types [12].
Inheritance	Mechanism designed to facilitate, from the definition viewpoint, the hierarchical relationship between classes [5].
Inheritance Visibility	Access type assigned to the inheritance in order to limit the subclass's permission on superclass members.
Nested Class	Class whose declaration is inside the body of another class [9]
Nesting	Relationship between class and its nested class.
Non-Extendable Class	Complete class that can not be extended by descendant classes, that is, it can not be inherited by other classes.
Root Class	Class directly or indirectly inherited by all other classes [12], containing a set of general purpose resources.
Subclass	Class defined by inheritance with another class [13], inheriting its instance variables and methods.
Superclass	Extensible class that shares its characteristics with the subclasses derived from it
Type Parameter	Identifier that specifies a generic type name whose instance must set a recognized type to replate the identifier at runtime.

4.3.4. Ontology Verification

The Table 4.3.3 presents the verification of competency questions listing necessary elements of the ontology to answer each competency question (CQ).

Table 4.3.3 Verification of the OOC-O Class Module.

CQ1	**What types of classes are present in an OO source code?** 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
CQ2	**What is a root class?** Extendable Class *subtype of* Class; Root Class *subtype of* Extendable Class
CQ3	**What are type parameter of a generic class?** Type Parameter *component of* Generic Class
CQ4	**What are the superclasses of a class?** Subclass *subtype of* Class; Subclass *inherits in* Inheritance; Superclass *inherited in* Inheritance.
CQ5	**What is the visibility of an inheritance?** Inheritance characterized in Inheritance Visibility
CQ6	**What classes are present into a class?** Class *nest in* Nesting Nested Class *nested in* Nesting

4.3.5. Ontology Validation

The Table 4.3.4 presents the validation of the ontology instantiating their concepts. For this we use instances of the Java programming language.

Table 4.3.4 Validation of the OOC-O Class Module.

Term	**Instantiation**
Abstract Class	**Shape** in: public abstract class Shape{}
Concrete Class	**Rectangle** in: public class Rectangle{}
Extendable Class	**Rectangle** in: public class Rectangle{}
Generic Class	**GenericShape** in: public class GenericShape<T> {}
Inheritance	**extends** in: public class Polygon extends Shape{}
Inheritance Visibility	No Related to Java

Nested Class	**Point** in:
	public class Rectangle{
	  class Point{}
	}

Nesting	**class Point** in:
	public class Rectangle{
	  class point{}
	}

Non-Extendable Class	**Math** in: public final class Math{}
Root Class	**Object** in: public class Object {}
Subclass	**Polygon** in: public class Polygon extends Shape{}
Superclass	**Shape** in: public class Polygon extends Shape{}
Type Parameter	**T** in: public class GenericShapes<T> {}

4.4. Object-Oriented Code Ontology - Class Member Module

4.4.1. Competency Question

For functional requirements, we have iteratively defined the competency questions (CQs) presented in Table 4.4.1.

Table 4.4.1 Competency question of the OOC-O Class Member.

ID	Competency Question
CQ1	What types of methods are present in an class?
CQ2	What variables are present into a class?
CQ3	What are type parameter of a generic method?
CQ4	What is a class method?
CQ5	What is a class instance?
CQ6	What types of instance methods are present in an class?
CQ7	What is a constructor method?
CQ8	What blocks are present into a method?
CQ9	What variables are present into a method?

4.4.2. Conceptual Model

The OntoUML diagram is presented in Figure 4.4.1 and the definitions of their terms is presented in the Dictionary of Terms (Table 4.4.2).

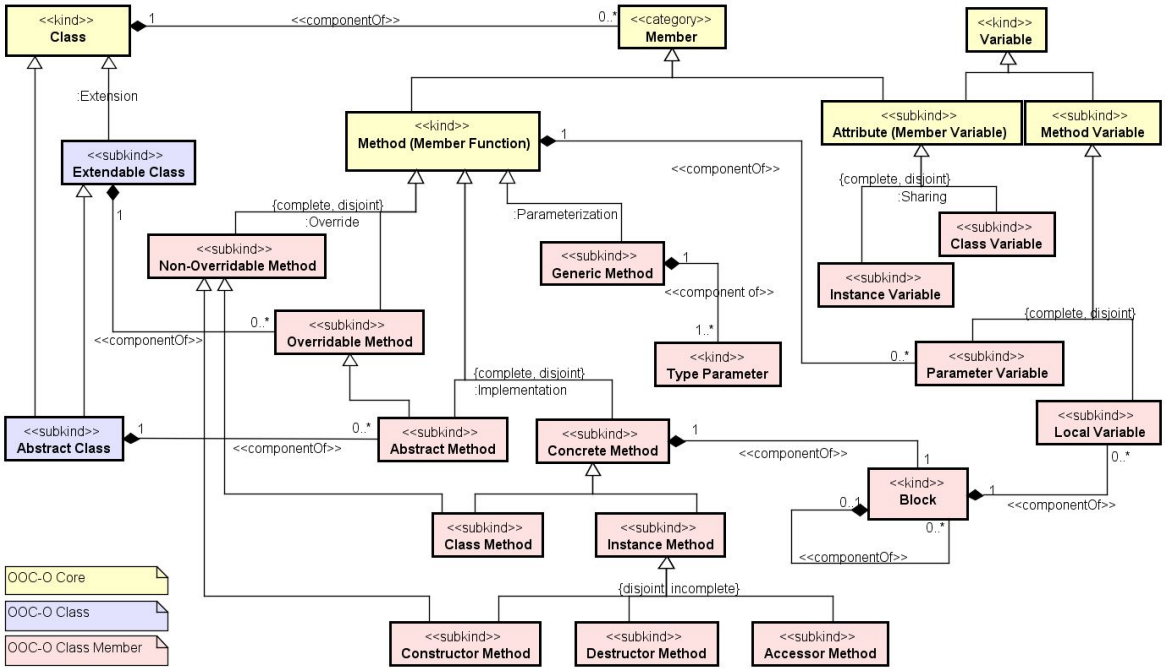

Figure 4.4.1 OntoUML Diagram of the OOC-O Class Member Module.

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 [14]. 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 [7], in Constructor Method, which species 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: ∀ rt : ReturnType;m : Method; characterization(rt;m) ! :ConstructorMethod(m) ^ :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 [13], 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: ∀ v : LocalVariable; b1; b2 : Block; componentOf(v; b1)^componentOf(b1; b2) ! componentOf(v; b2) (A8) and ∀ v : LocalV ariable; b : Block; m : ConcreteMethod; componentOf(v; b) ^ componentOf(b;m) ! 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.

4.4.3. Terms Dictionary

The definition of the terms present in OOC-O Class Member Module is presented in Table 4.4.2.

Table 4.4.2 Terms Dictionary of the OOC-O Class Member Module.

Term	Definition
Abstract Method	Method just composed by its signature [4], that is, it has no body and implementation [6] and should be implemented or "concreted" in all subclasses [14].
Accessor Method	Method providing an interface between the internal data of the object and the external world [7], that is, it allows access to private instance variables of an object.
Block	Instruction group handled by the compiler as a single instruction.
Class Method	Method independent of any instance of the class, being invoked by means of message sent directly to its class rather than its instance [11].
Class Variable	Variable independent of any instance of the class [13], whose single copy is shared by all objects of class [6].
Constructor Method	Special method that specifies how to create and initialize an object, ensuring that all its instance variables are properly initialized [4].
Concrete Method	Method that, in contrast to abstract method, must have body and implementation.
Destructor Method	Special method that cleans up unused objects, being explicitly invoked by a operator or automatically when an object is deallocated [4].
Generic Method	Parametrized method used to create instances or specializations of this model at compile time. For this it declares its own Type Parameter.
Instance Method	Method defined in the class scope and invoked by means of an object [4], operating only on objects of its class [13].
Instance Variable	Variable defined in the class scope to represent the state of an object [5], defining its particular state by means of attributes [6] and values.
Local Variable	Variable with small scope and lifetime [11] that exists in memory just during a method or block, not being visible outside of it.
Overridable Method	Method that can be overwritten in descendant classes [13].

Non-Overridable Method	Method that can not be overwritten in descendant classes [13].
Parameter Variable	Variable declared as formal parameter of methods, constructors, and exception handlers [6].
Return Type	Type of the return value declared for a method. Some methods can only act on the object state and do not send a return value to the program [5]
Type Parameter	Identifier that specifies a generic type name whose instance must set a recognized type to replate the identifier at runtime.

4.4.4. Ontology Verification

The Table 4.4.3 presents the verification of competency questions listing necessary elements of the ontology to answer each competency question (CQ).

Table 4.4.3 Verification of the OOC-O Class Member Module.

CQ1	**What types of methods are present in an class?** Generic Method *subtype of* Method; Concrete Method *subtype of* Method; Abstract Method subtype of Method; Non-Overridable Method *subtype of* Method; Overridable Method *subtype of* Method.
CQ2	**What variables are present into a class?** Member *component of* Class; Attribute (Member Variable) *subtype of* Member; Instance Variable and Class Variable *subtype of* Attribute (Member Variable).
CQ3	**What are type parameter of a generic method?** Type Parameter *component of* Generic Method.
CQ4	**What is a class method?** Concrete Method *subtype of* Method; Class Method *subtype of* Concrete Method and Non-Overridable Method.
CQ5	**What is a class instance?** Concrete Method *subtype of* Method; Instance Method *subtype of* Concrete Method.
CQ6	**What types of instance methods are present in an class?** Concrete Method *subtype of* Method; Instance Method *subtype of* Concrete Method; Constructor Method and Destructor Method and Accessor Method *subtype of* Instance Method.
CQ7	**What is a constructor method?** Concrete Method *subtype of* Method; Instance Method *subtype of* Concrete Method; Constructor Method Method *subtype of* Instance Method and Non-Overridable Method.

CQ8	**What blocks are present into a method?** Block *component of* Concrete Method; Block *component of* Block.
CQ9	**What variables are present into a method?** Local Variable *component of* Block; Block *component of* Concrete Method; Parameter Variable *component of* Method (Member Variable).

4.4.5. Ontology Validation

The Table 4.4.4 presents the validation of the ontology instantiating their concepts. For this we use instances of the Java programming language.

Table 4.4.4 Validation of the OOC-O Class Member Module.

Term	Instantiation

Abstract Method	**containCoordinate** in:
	public abstract class Shape{
	  public abstract boolean containCoordinate(double x, double y);
	}

Accessor Method	No Realted in Java

Block	**{int p= this.width * this.height; return p; };** in:
	public class Rectangle{
	  public int perimeter(){
	  int p = this.width * this.height;
	  return p;
	  };
	}

Class Variable	**initialCoordinateX** in:
	public class Rectangle{
	  public statict double initialCoordinateX = 0;
	}

Constructor Method	**Rectangle(int width, int height){};** in
	public class Rectangle{
	  Rectangle(int width, int height){};
	}

Concrete Method	**perimeter** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Destructor Method	**finalize** in:
	public class Rectangle{
	  protected void finalize() {};
	}

Generic Method	**printShapes** in:

	public class GenericShapes<T>{
	  public void printShapes(T type){};
	}

Instance Method	**perimeter** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Instance Variable	**width** in:
	public class Rectangle{
	  private int width;
	}

Local Variable	**p** in:
	public class Rectangle{
	  public int perimeter(){
	  int p = this.width * this.height;
	  return p;
	  };
	}

Overridable Method	**perimeter** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Non-Overridable Method	**perimeter** in:
	public class Object{
	  public final Class getClass(){};
	}

Parameter Variable	**width and height** in:
	public class Rectangle{
	  Rectangle(int width, int height){};
	}

Return Type	**int** in:
	public class Rectangle{
	  public int perimeter(){};
	}

Type Parameter	**<T>** in: public class Polygon<T>{}

5. Referências

[1] Turner, R.: Programming languages as technical artifacts. Philosophy & technology 27(3), 377397 (2014)

[2] Turner, R., Eden, A.H.: Towards a programming language ontology. Citeseer (2007)

[3] LaLonde, W.R., Pugh, J.R.: Inside smalltalk, vol. 2. Prentice Hall (1990)

[4] Tucker, A.B.: Programming languages> Principles and Paradigmas. Tata McGraw-Hill Education (2007)

[5] De Melo, A. C. V., & Da Silva, F. S. C. (2003). Princípios de linguagens de programação. Edgard Blucher.
[6] Pinheiro, F. A. (2006). Fundamentos de computação e orientação a objetos usando Java. LTC.
[7] Hunt, J.: Java and object orientation: an introduction. Springer Science & Business Media (2002)
[8] Phillips, D.: Python 3 object-oriented programming. Packt Publishing Ltd (2015)
[9] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D.: The java language specication: Java se 10 edition, 20 february 2018 (2018)
[10] Lutz, M. (2013). Learning python: Powerful object-oriented programming. " O'Reilly Media, Inc.".
[11] Lewis, S.: The art and science of Smalltalk, vol. 1. Prentice Hall (1995)
[12] Eiffel, E.: Eiel: Analysis, design and programming language. ECMA Standard ECMA-367, ECMA (2006)
[13] Sebesta, R.W.: Concepts of programming languages. Boston: Pearson, (2012)
[14] Brauer, J.: Programming Smalltalk-Object-Orientation from the Beginning. Springer (2015)