UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VINÍCIUS ABNER PEREIRA DE SOUZA

THE CORRESPONDENCE BETWEEN THE MEDIEVAL TRIVIUM AND OBJECT-ORIENTED PROGRAMMING

CAMPINA GRANDE - PB

2023

VINÍCIUS ABNER PEREIRA DE SOUZA

THE CORRESPONDENCE BETWEEN THE MEDIEVAL TRIVIUM AND OBJECT-ORIENTED PROGRAMMING

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Professora Eliane Cristina de Araújo.

CAMPINA GRANDE - PB

2023

VINÍCIUS ABNER PEREIRA DE SOUZA

THE CORRESPONDENCE BETWEEN THE MEDIEVAL TRIVIUM AND OBJECT-ORIENTED PROGRAMMING

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professora Eliane Cristina de Araújo

Orientadora – UASC/CEEI/UFCG

Professora Francilene

Examinadora – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em 14 de fevereiro de 2023.

CAMPINA GRANDE - PB

RESUMO

Ensinar o paradigma de programação orientada a objetos costuma ser um desafio para os professores. A principal dificuldade é muitas vezes atribuída à mentalidade que o paradigma exige. Essa mentalidade envolve raciocinar sobre elementos da realidade em termos de classes, objetos, atributos, polimorfismo etc. Em suma, é uma mentalidade que requer boas habilidades de abstração. Várias metodologias, abordagens e ferramentas já foram propostas para ajudar os alunos a alcançar a mentalidade necessária para aplicar esse paradigma, mas o aprendizado continua difícil. Diante disso, uma ferramenta que até então nunca havia sido considerada para o ensino de programação é o Trivium medieval. O Trivium consiste nas três artes liberais de Gramática, Lógica e Retórica. O syllabus e a estrutura das aulas do Trivium podem ser um modelo interessante para ser aplicado em cursos de programação orientada a objetos, pois abordam de forma bastante didática conceitos fundamentais idênticos ao do paradigma orientado a objetos. A demonstração da correlação entre os dois assuntos é um dos objetivos deste trabalho. Além disso, conjecturamos que ensinar os conceitos fundamentais da Gramática antes ou paralelamente ao ensino do paradigma orientado a objetos parece ser mais eficiente do que começar logo pela prática de programação, como costuma ser feito em cursos de programação. Este artigo propõe duas abordagens para o ensino do paradigma orientado a objetos. Eles consistem na estruturação do curso de Programação Orientada a Objetos com base na filosofia e metodologia educacional clássica, a fim de facilitar a compreensão do paradigma.

The Correspondence Between The Medieval Trivium And Object-Oriented Programming

Vinícius Abner Pereira de Souza
Federal University of Campina Grande
ORCID: 0000-0002-6048-4364
vinicius.souza@ccc.ufcg.edu.br

ABSTRACT

Teaching the object-oriented programming paradigm is often a challenge for teachers. The main difficulty is often attributed to the mindset that the paradigm requires. This mindset involves reasoning about elements of reality in terms of classes, objects, attributes, polymorphism, etc. In short, it is a mindset that requires good abstraction skills. Several methodologies, approaches, and tools have already been proposed to help students achieve the mindset necessary to apply this paradigm, but it remains difficult. In view of this, a tool that until then had never been considered for the teaching of programming is the medieval Trivium. The Trivium consists of the three liberal arts of Grammar, Logic, and Rhetoric. The syllabus and lesson structure of the Trivium can be an interesting model to be applied in object-oriented programming courses because it addresses in a very didactic way fundamental concepts that are identical to that of the object-oriented paradigm. A demonstration of the correlation between the two subjects is one of the goals of this paper. Moreover, we conjecture that teaching the fundamental concepts of Grammar before or alongside the teaching of the object-oriented paradigm seems to be more efficient than starting right away with programming practice, as it is usually done in programming courses. This article proposes two approaches to teaching the object-oriented paradigm. They consist of structuring the Object-Oriented programming course based on classical educational philosophy and methodology in order to facilitate the understanding of the paradigm.

Keywords

Object-Orientation, education, Trivium, paradigm, programming, abstract thinking, object-oriented programming, education in Computer Science

1. INTRODUCTION

The object-oriented paradigm has been consolidating itself as one of the most influential programming paradigms in recent years, which makes teaching this paradigm in programming courses indispensable. However, teaching is often a challenge for teachers, who recurrently report difficulties students face when trying to understand the fundamental concepts of the paradigm. The mindset necessary for the application of this paradigm involves reasoning about elements of objective reality in terms of classes, objects, attributes, interfaces, etc. In short, having good abstraction skills.

Several methodologies, approaches, and tools have already been proposed to help students achieve this mindset. However, there is still no consensus. In view of this, a tool that has not been considered for the teaching of programming is the medieval Trivium. The Trivium consists of the three liberal arts of Grammar, Logic, and Rhetoric. The concepts and theoretical frameworks presented by these liberal arts are the base for many modern knowledge fields, such as Computer Science. If those arts are so correlated to a field, it is natural to wonder whether aspects of the Trivium can be used as a model to Computer Science courses. More specifically, to Object-Oriented (OO) programming courses, since those liberal arts address in a very didactic manner fundamental concepts of object-orientation itself. Those concepts and their correlation to the paradigm will be explained and demonstrated later in section 4.

As already suggested in the literature [1][2][3][4], teaching the fundamental concepts of the OO paradigm before the introduction of programming seems to be more efficient than starting right away with practice and waiting for students to learn the fundamental concepts through exhaustive exercises and examples. Given these points, different approaches to teaching the object-oriented paradigm are welcomed. Two of them will be suggested by the end of this article.

2. METHODOLOGY

This work is a descriptive research based on bibliographic references regarding approaches, difficulties, and good practices for teaching OOP, as well as classic education. This research aims to demonstrate the correlation between classic Trivium and OOP. The resulting demonstration will then be used to base two teaching methodology proposals at the end of this paper.

3. PROBLEMS IN THE TEACHING OF OBJECT-ORIENTED PROGRAMMING

When it comes to learning a new subject, a common approach is to first give the student a general introduction to it before moving on to the details. This approach is used in many Information Technology courses, including Object-Oriented Paradigm (OOP) courses. However, the main difficulty students reportedly have when learning OOP is understanding the fundamentals [1][2]. Anecdotal evidence points out that teachers are not managing to introduce the subject effectively, and the discussion in the community often revolves around whether the OOP itself is complex or whether the tools, methodologies, and approaches used to teach it are inappropriate [5]. It seems to be a combination of both things.

OOP is complex because it fundamentally imitates the high level of abstraction that human cognition works with when interpreting reality [6]. Apparently, this essential aspect of the paradigm as an emulation of the mind's process of abstracting information is not taken into consideration as it should by teachers, textbooks, and courses. That might be one of the causes of ineffective teaching methodologies. This conjecture is supported by [7], who argue that a weak foundation on basic concepts leads to difficulties in transforming textual problems into mathematical formulae and programming solutions. A reason why these fundamental concepts are not taken into consideration as much as they should be is that, as stated by [5], in general, teachers don't really understand these concepts themselves. Hence, it is predictable that they will struggle to explain the theory clearly. Consequently, teachers rely on coding from lesson one in hope that students will learn through tedious trial-and-error practice [2][8]. This approach might work well for teaching simpler and straightforward paradigms such as the procedural one [1], but is ineffective and arguably even counterproductive for teaching the Object-Orientation mindset [3][8][9]. This is due to the fact that when it comes to this paradigm, designing the solution first before starting coding is crucial. Starting the OOP course with coding right away might accustom to (or reinforce) a bad habit of focusing immediately on the code rather than on the design [3][8].

Despite all of that, this code-since-day-one approach works in the long run – that is, students will be able to write and understand Object-Oriented programs. Due to exhaustive practice, examples, and persistence, students will eventually abstract patterns of Object-Oriented design well enough to apply them satisfactorily to get a passing grade [3][8][9]. However, in the end, after so much effort to reach the desired mindset, students won't really understand why object orientation works so well as a paradigm. One might argue that this know-how but not know-why course outcome is not necessarily a problem for forming an industry programmer. Indeed. Nonetheless, when it comes to Computer Science courses, which are expected to form computer scientists, this outcome seems to be insufficient. Besides, for both future scientists and industry programmers, developing the necessary level of abstract thinking for the OOP remains difficult [2][3][8]. Abstract thinking can be defined as "a mechanism that allows us to represent a complex reality in terms of a simplified model so that irrelevant details can be suppressed in order to enhance understanding" [5]. A more technical definition is "the act of creating classes to simplify aspects of reality using distinctions inherent to the problem" [3]. This surely requires a certain

intellectual caliber to be put into practice. However, as accurately stated by [2], despite the need for helping students reach that high caliber, “most of the introductory programming teaching material only emphasizes lower-level knowledge such as declarative and procedural aspects. [...] Hence, it only emphasizes ‘what’ and ‘how’ about programming concepts and syntax. This is only suitable for learning lower-level knowledge but not for solving complex programming problems in the real world. Therefore, in the long-term, students are unable to solve a programming problem as they are only equipped with low-level knowledge”. This might be due to the nature of the Information Technology field, which is extremely technical and practical. In addition to the fact that procedural languages ruled the market for such a long time, this may be the reason why programming courses still focus on teaching the technique – that is, focus on the ‘what’ and ‘how’ of what they are teaching, but not on the ‘why’. This limited view of the discipline given to students by the traditional “programming-oriented” approach has been concerning the computer science education community for a long time, and some call for a breadth-first approach for teaching programming instead [8]. A breadth-first approach considers a much broader range of topics in introductory courses in order to provide a more holistic, deep understanding of the discipline. However, developing successful breadth-first approaches has shown to be a challenge [8]. This way, there is a demand for breadth-first teaching methodologies that give equal importance to the ‘whys’ of the Object-Oriented Paradigm, and a hint for such an approach can be found in ancient knowledge long forgotten by modernity’s thirst for innovation.

4. THE MEDIEVAL TRIVIUM

The Latin word *trivium* means "three roads." Ancient and medieval education was structured on the three “roads” of learning, which consisted of Grammar, the skill of comprehending the facts; Logic, the skill of reasoning out relationships between these facts; and Rhetoric, the skill in wise, effective expression and application of the facts and their relationships [10]. The Trivium is part of the Seven Liberal Arts of western medieval education and it includes the arts that are pertinent to the mind. The other part, the Quadrivium (Music, Geometry, Astronomy, and Arithmetic), includes the arts pertinent to matter and nature [11]. These subjects were taught in European universities aiming to form high-level intellectuals.

The Trivium was taught first because it promotes the organization of the mind of the students and gives them the intellectual tools necessary to comprehend the complex reasonings and abstractions presented further in the Quadrivium. This pedagogical organization is due to the philosophy of medieval education, which is: firstly, one must learn how to think and express themselves correctly before trying to understand things outside of their mind [11][12]. This philosophy substantiates the whole structure of the medieval syllabus, which determined that the three ways, Grammar, Logic, and Rhetoric, were to be taught separately in this strict order.

Grammar in the Trivium does not have the same meaning as it has today, which is the study of a specific language and how to write it correctly. That would be a specific Grammar. Instead, Grammar in the Trivium is the general Grammar, which studies language itself; simply as the medium in which thought is expressed in any spoken language. General Grammar relates words to objective reality and applies to all subjects as the first set of building blocks to integrated or fully mindful, objective knowledge. Special Grammar, on the other hand, is concerned with the relation of words to words within a specified language, such as English [11]. General Grammar points out that roughly every existing language classifies words morphologically. These classes are adjectives, adverbs, substantives, verbs, and so on. That means that there are structural aspects of reality that are commonly perceived by humans and are abstracted by them into common words in common categories in order to systematically organize knowledge and, finally, allow thoughts to be communicated.

Logic was taught next, and it aimed to teach students how to use the language; how to define terms, construct arguments, and detect fallacies [11][12]. This might be the only one among the classical three roads that remain recurrent in modern syllabuses since it consists of basic formal propositional logic. Finally, Rhetoric was taught so that students could use the language to express themselves logically,

elegantly, and persuasively [11][12]. After being approved in the studies of the Trivium, the student was ready to dig into the more abstract and complex subjects of the Quadrivium.

Despite its effectiveness, this classical education has become obsolete as the social, economic, and cultural changes that took place after the Renaissance. It promoted a different educational philosophy focused on social demands. This new philosophy states that education should primarily socially distribute knowledge and form good professionals with certain competencies rather than wise erudites [12][13]. However, due to its huge educational value, in recent years some experts are calling for a revival of the medieval Trivium in 21st-century education. They stress that it is not obsolete, but, on the contrary, the Trivium is a great tool to develop the competencies expected from modern citizens [14]. That includes programming competencies.

5. THE RELATION BETWEEN TRIVIUM AND OOP

What have these ancient pearls of wisdom and methodologies to do with such a modern concept as Object-Oriented programming? As surprising as it might sound, the answer is everything. One of the advantages of the OOP is that it gives us tools for organizing our programs and our thoughts about them [15], and, as seen, organizing our thoughts about reality is the main goal of studying the Trivium. Moreover, as argued by Dorothy Sayers, every subject in every field of knowledge can be broken into three aspects analogous to Grammar, Logic, and Rhetoric [12]. Sayers states that every subject has its Grammar, that is, the study of the basic facts and concepts and the fundamental rules. Furthermore, every subject has its Logic; the theoretical understanding of the relationships between these facts and the rules — in other words, how all the parts fit together. Finally, every subject has its Rhetoric, which is the wisdom to verbally express and practically apply what one knows and understands. To fully understand a subject, one must study all of these aspects of it. The next sections will focus on demonstrating how Object-Oriented programming can also be broken down into these three aspects and how it can be useful for understanding the paradigm.

5.1 Grammar in Object-Orientation

One of the first concepts that are taught in Grammar is the essence. Essence is defined as “what makes a being what it is”[11]. This is such a crucial concept because the essence of an object is the result of the mind’s abstraction process of reality. Miriam Joseph explains it: “The intellect through abstraction produces the concept. The imagination is the meeting ground between the senses and the intellect. From the phantasms [mental image of the individual object perceived] in the imagination, the intellect abstracts that which is common and necessary to all the phantasms of similar objects [...]; this is the essence [...]. The intellectual apprehension of this essence is the general or universal concept.”[11]. This mental process is so important for computational thinking and modeling Object-Oriented software that it is also mentioned by many Computer Science publications [3][6][14][16]. What makes a bench a "bench" and not a table is its essence. The essence is sometimes hard to define in words, but the mind can easily tell the difference between a bench and a table. After abstracting the essence of objects, the next step the intellect takes is to classify them into groups of objects that share the same specific essence [11]. In Grammar, these groups are called species; in the Object-Oriented paradigm, they are called classes. For comparison, this is one definition of class extracted from a Computer Science article: “A description of the organization and actions shared by one or more similar objects.” [5].

Another core concept in Grammar that is taught concurrently with that of species is that of an individual. An individual is a physical being that instantiates the essential characteristics of a species [11]. A species cannot exist on its own because it is not a substance, rather it is an abstraction: “[...] for one cannot photograph the species horse or dog; one can photograph only an individual horse or dog since every horse or dog that exists is an individual”[11]. In the same way, a class in Java is not useful unless it is instantiated as an object that can act and be acted on. A class definition is merely the description of the essential characteristics of a potential object. Alongside species, Grammar teaches the concept of genus, which is a wider class made up of two or more different species that have in common the same

generic essence or nature [11]. For example, a real horse is an individual of the abstract concept of 'horse', which is a specific kind of the even more abstract concept of 'animal'. A genus in Grammar would be equivalent to an abstract class in Java. Therefore, it can be said that the species that compose the genus inherits some common characteristics from an upper class. As can be seen, Grammar teaches that there are layers of abstraction in our perception of reality, although the concept of inheritance is not explicitly explained as in OOP.

5.1.1 Aristotle's Categories in Object-Orientation

Aristotle's Categories are also taught in Grammar as an important framework of thinking to guide one's investigations and understanding of reality. Those ten categories are Substance, Quantity, Quality, Relation, Action, Passion, When, Where, Possession, and Posture. In this Categories theory, also known as categorialism [17], words are categorized by their relationship to being and to each other. Aristotle's categories enable us to translate the linguistic symbol into a logical entity ready to take its place in a logical proposition [11].

Substance (or subject) is the most important category since it denotes an individual being that exists in themselves, whereas the other nine categories denote accidents (or predicates). Accidents are concepts that cannot exist on their own, but only in others. As an example, a chair is a substance while its attributes are accidents. Those attributes might be the chair's color (Quality), the chair's location (Where), whether the chair is being used (Passion), whether the chair has four legs (Quantity), whether the chair in question is in the present or in the past (When), and so forth. In summary, the usage of the Categories as a framework evolves breaking down facts of a being into categories so our knowledge about it is clearer.

An analogous breaking-down process happens when we design classes in OO programs. For example, if we design a class that represents a client, one of the first questions to be asked in the brainstorming step is: what is a client in a specific context of the program? When this question is answered, programmers have defined the substance. The next questions revolve around the substance's attributes. Does the client have a name? (Quality). Does the client's class inherit from another upper class or exchange information with another class? (Relation). Are there different types or tiers of clients? (Quality, once again). Which methods should clients have? (Action). How can other classes interact with the client? (Passion). Therefore, roughly speaking, a class name in an OO program defines a substance whereas everything inside the class is a description of their accidents.

Here, the concept of information hiddenness is subtly present. Since an accident cannot exist on its own, it needs to be associated with a substance in order to exist. For instance, the color red doesn't exist, unless it is part of another being. In other words, we can only talk about an attribute if there is something concrete that it is attributing. This need for an association is analogous to information hiddenness. In pure OO programming, there are no global variables that can be defined outside a class and used anywhere by all the classes that compose the program. A variable must be defined inside a class before it can ever be referred to in a program.

5.2 Logic in Object-Orientation

In the Logic stage, medieval students sort the facts they learned in the Grammar stage and learn how to discern the truth from the falsity. This discernment ability is developed by training students to establish valid relationships among facts, which includes the ability to identify fallacies, structure a valid argument, detect and avoid contradictions, and so on. In other words, by studying Logic the student will learn to translate the words in natural language into logical entities in order to find the truth in them. The logic studied in the Trivium is simple propositional logic.

In the study of OOP, the logical step refers to how classes can relate to each other and how to associate them correctly. The relationship between classes and the way of handling information access is the main differential of OOP and it is one of the reasons for it being so efficient. From an object-oriented perspective, it would make no sense to design a program that allows every class to access and manipulate data held by every other class. That would go against the very reason for grouping information into classes in the first place, which is to hide information. In other words, this solution design would violate the core OO concept of encapsulation. Despite still running if it has no lexical or syntactic

errors, the program has serious logical inconsistencies from an object-oriented point of view.

The other two main logical concepts of OOP, polymorphism, and inheritance, were already mentioned in the subsection regarding the Grammar of OOP. As demonstrated before, the concepts of species and genus and how they relate have the notion of polymorphism embedded into them. Polymorphism is "the ability of different classes to respond to the same message and each implement the method appropriately"[5], and this ability is also seen in the physical realm. For example, the abstract concept of an animal manifests itself in reality in many different specific forms, like a horse or a dog. Each form of animal shares behaviors that are characteristics of animals, such as reproducing, eating, sleeping, moving, etc.; They all do those things roughly differently from one another. This differentiation is what drives the abstraction and classification of entities into species and genera. Likewise, an abstract class (or interface) User can be instantiated as (or implemented by) concrete classes such as one for a Client or an Administrator. Every species of user has its own methods implementation of the methods defined for a User. Similarly, inheritance is also embedded in the relationship between levels of abstraction, although it is slightly different from the one of polymorphism. Inheritance is defined as "a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class"[5]. In other words, the main difference between polymorphism and inheritance is that, whereas in polymorphism the implementation of shared methods varies, in inheritance, those implementations remain the same as the upper classes. In the end, both core logical concepts of OOP are somewhat derived from levels of abstraction.

Finally, Logic is intrinsically connected to Computer Science itself, not only OOP [18]. In general, Computer Science graduation includes at least one subject related to formal logic and its application of it in computer systems. However, it is important to note that the logic used in these courses is more modern than the one taught in the classical Trivium. For instance, the classic trivial art focused on classical propositional logic and, therefore, didn't include modern contributions to the field such as Boole's laws, which are an essential part of the logic for Computer Science. Although it is surely not sufficient for the needs of a modern computer scientist, the Logic as it is presented in the Trivium is the base of all the other further developments of logic and its study stands as necessary even in modern times.

5.3 Rhetoric in Object-Orientation

The relation between object-orientation programming and Rhetoric is more subtle. That is because Rhetoric in the classical Trivium aims towards appropriate verbal communication, whether by written or spoken speech [11]. In more detail, classical Rhetoric teaches students how to use the facts they learned in Grammar alongside the rules of Logic in order to make a persuasive, coherent and appropriate speech. Sayers adds that the rhetorical aspect of a subject is the practical application of all the knowledge associated with it and not only verbal communication [12].

Given this, it is possible to notice a rhetorical aspect in the practice of programming regardless of the paradigm. When programmers code an abstracted model from reality, they are applying their knowledge to something practical; the program. However, the programmer should not write the code only aiming towards making it work. That is because, in the end, a program is also a form of communication. Thus, they need to write it clearly and neatly because other programmers will eventually need to work on that program, and they must understand it. Choosing good variable names, keeping a decent indentation, extracting blocks of code into well-named methods and many other good practices are exactly forms of appropriate speech; the Rhetoric of programming.

Moreover, it is possible to identify a rhetorical aspect in the practice of OOP specifically. As mentioned before, classical Rhetoric aims towards teaching students how to use the body of information they learned in Grammar in combination with the logic rules for associating them correctly to communicate knowledge and thoughts effectively. Because of this, modeling languages such as UML and OMT can be considered to be rhetorical devices used for OO programming. That is because one of their main goals is to communicate the design of an OO system effectively. Furthermore, in order to build a model for a OO system, one needs to have an understanding of

the body of information associated with the system (classes, attributes, methods, etc) and an understanding of how these pieces of information should be associated for the system to work correctly (inheritance, polymorphism, and other classes associations). All this knowledge is then graphically communicated through UML, which has become a standard in the industry due to its rhetorical effectiveness [19].

6. APPLYING THE TRIVIUM TO PROGRAMMING TEACHING

There is, indeed, a correlation between the Trivium and Object-Oriented Programming – and, more generally, programming itself. In face of that, the question that emerges is: *is it possible to apply the Trivium to the teaching of Object-Oriented Programming somehow?*

6.1 Some possibilities of using the Trivium for teaching OOP

As demonstrated, OOP has aspects analogous to Grammar, Logic, and Rhetoric. However, among the three classical subjects, Grammar seems to be the most useful for the teaching of the paradigm in most programming courses. More precisely, it can be useful to introduce the paradigm and provide an easier mindset shift. Next, there are two proposed approaches that use Grammar as a teaching tool for the Object-Oriented Paradigm.

6.1.1 The Introduction to Object Orientation Module

In this approach, the OOP course includes an introductory module that is divided into two parts. The first part is used for introducing how the human brain interprets and abstracts reality. In other words, the first part is a Grammar module. Since the proposition is not to teach the Trivium itself, this part doesn't need to be long. It could be the first couple of lessons, depending on each course's chronogram and structure. The second part of the introductory module is used for introducing OO programming itself by writing programs based on what was discussed in the first part.

In more detail, in the first part, the lecturer will explain how human communication works based on the concepts of substance, genus, individuals, and classes (as in the Trivium, not in OOP yet). Moreover, it will be explained how Aristotle's Categories can be a useful tool to organize our understanding of reality. By the end of this part, the students are expected to understand how the human intellect abstract and classify words and concepts in order to compose the human language.

The programming paradigm itself will be introduced in the following part, which will introduce the basic Object-Oriented concepts, such as objects, classes, abstract classes (or interfaces), attributes, methods, polymorphism, and inheritance. This introduction will be based on writing programs based on the examples given in the previous part. With this methodology, students should note how the same cognitive process they learned in the introductory module is emulated by the Object-Oriented Paradigm. This way, their mindset shift to Object-Orientation would be more natural and quick.

It would be interesting to use examples related to entities of the natural realm, such as animals, rather than entities of the 'corporate' or digital realm. Those corporate or digital realm entities include common roles in computer programs, such as Users, Clients, Administrators, or even companies and organizations. These entities are usually used as examples in programming textbooks and classrooms because they are common in real world programs. However, they are not to be used in this proposed approach. That is because the teachers of the Trivium would teach Grammar by using examples related to what their students would see concretely in their daily lives. Those concrete entities would include animals, objects, and other natural elements that students are likely to be used to from their personal experiences. By starting with these more concrete examples, we would probably take the most out of Grammar because it would be closer to how it was originally taught. An application of this approach will be detailed below:

6.1.1.1 Introductory Module Part 1: Introduction to Grammar

The lecturer should start the module by explaining that, in order to understand the Object-Oriented Paradigm, it is important to understand human communication first. Below, the concepts to be tackled in this module are listed in order, followed by a summary of what the lecturer should explain about them. Both order and explanation of these concepts

were based on the Grammar section of Miriam Joseph's Trivium textbook [11].

Language. Only human beings can emit sounds that are tied into phrases to express thoughts. This is a language, and it is formed by symbols that have arbitrary meanings. Those symbols are called "words" and they represent concepts of reality. Some examples of words can then be given, such as "horse" (substantive), "good" (adjective), "eat" (verb), "fast" (adverb), etc. It would be interesting to give examples of words from different morphological classes to prevent students from sticking too much on substantives when they think of "words". According to Aristotle, language and reality have the same structure.

Individuals. Words can mean individuals or entities. An individual is a unique physical being. Examples of individuals that can be given are a woman, a tree, a horse, a snake, a rock, and so forth. For this demonstration, three individual animals, a horse, a snake, and a human will be used as examples from now on. For better understanding, the lecturer should show an illustration of them when needed (whether by drawing on the board or by showing a picture). In this example, the chosen animal for representing the concept of an individual was a horse (figure 1).

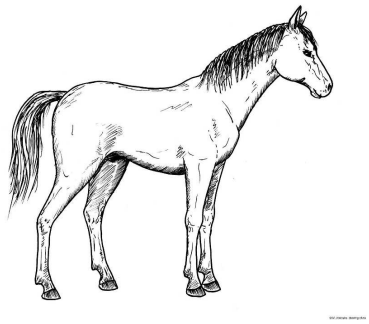

Figure 1 - an individual.

Essence. Essence is what makes an entity what it is, and without which it can't be what it is. At this point, a discussion can be proposed by the lecturer by asking the class what makes a horse a horse. What do all horses have in common? What is it that something needs to have in order to be a horse? This reflection would be a good initial abstraction exercise, which will happen more often during the course when they need to define classes.

Species. A species is a word that represents a collection of all the individuals that share the same specific essence. The species of the individual illustrated in figure 1 is *horse* (figure 2).

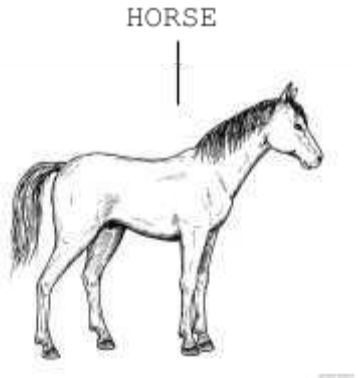

Figure 2 - The word *horse* is a species.

Now, the lecturer should make clear that a species in this context is a word that represents an essence, not the biological classification, as students might think at first. In order to make this clear for the students, it can be given the example of the word umbrella as being a species of objects. Once the concept of species in Grammar is understood, a link can be made between the two previous concepts: the essence is what makes an individual similar to the other individuals of their species, whereas his individuality is what makes him different from the other individuals of his species. To exemplify this concept in a sentence, all horses are similar but each one of them is unique.

Genus. Genus is a word that represents a collection of all the species that share the same generic essence. In other words, a genus is an even more abstract species. The species "horse" is part of the genus "animal", as well as the species "snake". Another discussion can be proposed by the lecturer by asking students questions such as "What makes a horse an animal? What does a horse have in common with a snake?" The level of abstraction of a genus can be explained as the following: When one thinks of a "horse", they will think of individuals which have the specific characteristics of what would be considered a horse. However, when one thinks of an "animal", they

can think of a wider variety of individuals that have the specific characteristics of an animal, like a horse, a snake, a human, a fish, etc. The concept of genus can be illustrated as in figure 3.

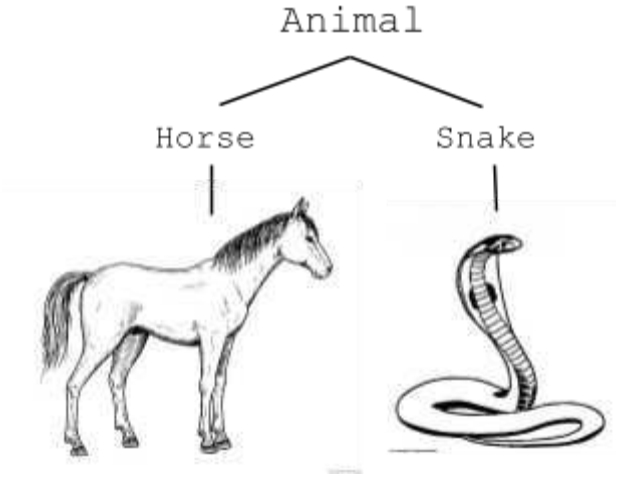

Figure 3 - Another level of abstraction.

Specific Essence and Generic Essence. These concepts can be exemplified like this: Only animals can be born, grow, eat, reproduce and die – this is part of their generic essence. Every specific animal, however, has its own way of doing some of those things. The same action can have many forms. How the animal sleeps, what it eats, and how it reproduces are part of its specific essence. Note how the concept of polymorphism can be subtly introduced by stating that different species act the same in different ways.

Ten Categories of Aristotle. Words are created by our intellect in order to abstract reality, and they can be manipulated and cataloged to increment our understanding of reality. The Ten Categories of Aristotle classify words and phrases according to our knowledge of the being. In other words, Aristotle's Categories give us a framework of reasoning to understand what we know and what we don't know about a being. Almost every word in human language can be used to describe an aspect of a being. Those aspects that can be described are exactly what the categories try to classify. The principle of the categories is: every being exists on itself or in another.

Substance. If it exists in itself, it is a substance (or a subject). That's the first category.

Accident. If it exists in another, it is an accident (or a predicate). Those are all the other nine categories. The accidents are Quantity, Quality, Relation, Action, Passion, When, Where, State and Posture.

The lecturer will then show how to classify what is known of an arbitrary individual based on the Categories. The individual chosen for this demonstration is the horse used as an example so far. Even though there are ten categories, only six of them (namely Substance, Quantity, Quality, Relation, Action, and Passion) will be taken into consideration for this example. This is because they are the ones that will be more useful in a later programming exercise. However, how many categories and which of them will be used for demonstration is up to the teacher. In order to demonstrate the Categories, let's consider that the horse used as an example so far is a male white Arabian horse, with four legs, 185.5 cm in height, and is trotting while being mounted on by his owner. Table 1 shows what the classification with those six categories could look like.

Facts: A male white Arabian horse, with four legs and 185.5 cm in height is trotting while being mounted on by his owner.

Facts classification:

Category	**Knowledge**
Substance	Horse
Quantity	Four leg; 185.5cm in height
Quality	White; Arabian; Male
Relation	Ownership (relation with human)
Action	Trotting
Passion	Being mounted on

Table 1 - A classification of the known fact using Aristotle's Categories.

The elaboration of this table can be done through a discussion with the students led by the lecturer. In

this discussion, the lecturer can guide the student's classification of the given fact by asking questions about each category. Those questions could be: "What are the horse's qualities?", "What are its quantities? How many things can we count on it?", "What relation does it have with anything in this context?", "What is it doing? What can it do?", "Is it suffering from some action? ", and so forth. Those questions are very similar to the ones that are made during the process of designing OO programs. Therefore, by doing this exercise, the students are being trained to abstract certain information about an entity and classify them, which is a crucial ability for OO programming.

Conclusion. After exercising Aristotle's categories, the introductory module should head to its conclusion. The conclusion is that the human language is structured following grammatical concepts (species, genera, individuals, substances, accidents, etc). Those concepts are rules that allow us to systematically organize raw factual data gathered from objective reality in such a way that communication is possible. This is useful knowledge because the Object-Oriented Paradigm, in practice, emulates human language. The demonstration of this correspondence is the goal of the next module.

6.1.1.2 Introductory Module Part 2: Introduction to the Object-Oriented Paradigm

The second module should introduce the Object-Oriented Paradigm itself. In this module, the teacher will write OO programs based on the familiar examples given in the previous module. The idea behind this approach is that the two modules should follow somewhat the same structure and use the same examples. This way, it will be easier for students to note the similarities between human language and OOP. Once they understand this, their mindset shift would happen more naturally. Moreover, this approach considers the programming language used in the course as a tool for conceptual modeling, which is one of the roles of programming languages [20]. In other words, in this part, the programming language is used for expressing the concepts and structure discussed in the previous part [20] and not as the main focus. To demonstrate how this module could be conducted, a pseudo code based on Java will be used for the coding examples for simplicity reasons. In this pseudo code, access modifiers are going to be ignored and the standard Java printing statement "System.out.println" will be referred to as "print". Irrelevant code for the specific example will be suppressed and represented by three dots ("..."). However, any OO programming language other than Java can be used in class for this module.

Classes. The lecturer should start by explaining that an Object-Oriented program is mostly made of classes and their relationship with one another. A class is a structure that describes something. More technically, a class is a data structure that holds data related to some entity. These data can be called the attributes or fields of a class. A class that describes a horse can be defined as shown in figure 4.

```
class Horse {

}
```

Figure 4 - A class which describes a horse.

This piece of code defines the class Horse. It can also be said that it defines the species horse. It would be the programming equivalent of the species illustration in figure 2.

Attributes. The body of the class is where we are supposed to write a species' characteristics (or attributes). Attributes are what is known (and relevant) about a class. The process of defining a class' attributes and methods is analogous to the process of categorizing with the Ten Categories. The lecturer can either conduct a discussion similar to the one that happened for the construction of table 1 or use table 1 already in this step. In a class, the substance is the class name itself whereas everything inside the class body can be considered to be its accidents. To show how attributes are defined in a OO program, firstly let's try to define the familiar example of a white male Arabian horse, with four legs and 185.5 cm in height as a class. Let's start with its qualities, which were already listed in table 1 (figure 5).

```
class Horse {
  String color = "white";
  String gender = "male";
  String breed = "Arabian";
}
```

Figure 5 - The horse's qualities translated to code.

This is a good opportunity to explain to students what variables and data types are if they are not aware of them yet. The horse's quantities can be used to introduce numeric data types (figure 6).

```
class Horse {
  String color = "white";
  String gender = "male";
  String breed = "Arabian";
  int legs = 4;
  float height = 185.5;
}
```

Figure 6 - The horse's qualities translated to code as well.

Methods. The Action category can be used to introduce the concept of methods, arguments, and return types (figure 7). As discussed in the previous module, since a horse is an animal, it can eat, move, and sleep. Moreover, a "toString" method will be defined for later use in this module. For simplicity, attribute definitions will be suppressed for the following figure.

```
class Horse {
  ...

  void eat(String food){
    print("Eating "+ food);
  }
  void move(float distance){
    print("Walking for "+distance+   " miles");
  }
  void sleep(int hours){
    print("Sleeping for "+hours+"hours");
  }
  String toString(){
    return "I'm a "+gender+" "+color+
    " " + breed + " horse with "+legs+"
     legs and "+ height + "cm in height!"
  }
}
```

Figure 7 - A horse's potential actions translated to code as methods.

As for the other categories (namely, Where, When, and Posture), they won't be considered for this demonstration. However, lecturers can exercise them as much as they wish.

Object. All that was done until now was to describe which characteristics a horse has and what it can do. However, as a class is simply a description of something, it can't do anything or have characteristics itself. That is because a species (which is what a class defines) is simply an abstraction; a collection of potential individuals. Only individuals can act and have attributes such as color, gender, number of legs, and height. For a class to be useful in a program, it is necessary to instantiate an individual out of it. In Object-Oriented Programming, an individual is called an object. To create an object, a constructor method must be defined (figure 8).

```
class Horse {

  ...

  Horse (){

  }
}
```

Figure 8 - A constructor method added to the Horse class.

Once again, for simplicity, previously defined methods and attributes were suppressed in the last figure.

Running the program. After the first class is properly defined, students should see it working. For that, a Main class will be created and, for this demonstration, two horses will be instantiated and then printed (figure 9). It is a good chance to explain how to access data and methods inside an object.

```
class Main {

   void main(){
    Horse horse1 = new Horse();
    Horse horse2 = new Horse();

    print(horse1.toString());
    print(horse2.ToString());
   }
}
```

Figure 9 - A program which instantiates and uses the Horse class.

The expected output for this program is shown in figure 10.

```
I'm a male white Arabian horse, I have 4 legs and
my height is 185.5cm
I'm a male white Arabian horse, I have 4 legs and
my height is 185.5cm
```

Figure 10 - The desired outcome of the program.

However, this is very limited. Every object instantiated from this class will always have the same attributes (accidents). As seen, a class (species) represents a collection of potential objects (individuals) who share the same specific characteristics (specific essence). Every object has its individuality which differentiates it from the other objects instantiated from its class. For example, every horse has a color, a gender, a breed, and a height. But not every horse is white, male, or Arabian, nor has 185.5cm in height. Horses can have different attribute values. To enable individuality for classes' objects, both attribute definitions and constructor need to be changed as in figure 11. In figure 11 irrelevant methods for this step were suppressed, along with some arguments and declarations on the body of the constructor.

```
class Horse {
  String color;
  String gender;
  String breed;
  int legs;
  float height;

  Horse(String color, String gender,...){
    this.color = color;
    this.gender = gender;
    ...
    }
}
```

Figure 11 - A horse's attributes are now defined in run-time.

By removing initial values, the attributes a horse can have were made generic. In other words, the "color" field now, since it is not initialized, is a species of attributes, whereas the color name, such as "white", is an individual color. In order to create an individual horse object, it is necessary to manually specify its attributes as in figure 12. The expected output is shown in figure 13.

```
class Main {

   void main(){
    Horse horse1 = new Horse ("white","male",
"Arabian", 4, 185.5);
    Horse horse2 = new Horse ("black", "female",
"Mustang", 3, 190.0);

    print(horse1.toString());
    print(horse2.ToString());
   }
}
```

Figure 12 - A program that instantiates and uses custom horses.

```
I'm a male white Arabian horse, I have 4 legs
and my height is 185.5cm
I'm a female black Mustang horse, I have 3 legs
and my height is 190.0cm
```

Figure 13 - The desired outcome of the program.

Inheritance. The concept of inheritance can be explained next as another level of abstraction in OO programs. Inheritance will be explained with an abstract class. An abstract class cannot be instantiated because it does not define a collection of

individuals, but a collection of classes instead. An abstract class would be equivalent to a generic word (genus) which can mean different specific words (species). The generic word used as an example in the previous module, *animal*, will be used again to explain how inheritance works in an OO program. However, before that, it is important to create a similar class to that of a horse but for a snake, another familiar example from the previous module (figure 14).

```
class Snake {
  String color;
  String gender;
  int legs;
  float height;
  boolean venomous;

  Snake(String color, String gender,...){
     this.color = color;
     this.gender = gender;
     ...
  }
  void eat(String food){
    print("The snake swallows "+ food);
  }
  void move(float distance){
    print("The snake crawls for "+ distance+
"miles");
  }
  void sleep(int hours){
    print("The snake sleeps for "+ hours);
  }
  String toString("I'm a "+ gender + color + "
snake, I have "+legs+" legs, "+ height +"in
height and it is "+venomous+" that I am
venomous";
}
```

Figure 14 - A similar class to that of a horse, but for a snake instead.

A class that describes a snake as illustrated in figure 14 will show students how similar the two classes, horse and snake, are. This is because they share common characteristics that can (and should) be abstracted as animalistic attributes. However, these classes are not completely the same. In this specific case, only horses have breeds and only snakes can be venomous. Therefore, these are specific characteristics of an animal that aren't necessarily true for other animals. The classes' common attributes can be defined in a generic, abstract class that describes an animal (figure 15).

```
abstract class Animal {
  String color;
  String gender;
  int legs;
  float height;

  void eat(String food);
  void move(float distance);
  void sleep(int hours);
}
```

Figure 15 - A class that abstracts the characteristics of animals.

This way, every class that shares the generic essence of an animal can inherit this essence from this abstract class. By adding the 'extends' notation to a class definition, we denote that the class is a specific type of another more general class. As illustrated in figure 16, both Horse and Snake are specific types of Animal. In Object-Oriented programming, this is called inheritance. Figure 16 suppresses the methods defined in both classes.

```
class Horse extends Animal{
  String breed;

  Horse(String breed){
    super();
    this.breed = breed;
  }
  ...
}

class Snake extends Animal{
  boolean venomous;

  Snake(boolean venomous){
    super();
    this.venomous = venomous;
  }
  ...
}
```

Figure 16 - The Horse and Snake classes inherit common characteristics from the Animal class.

The other methods defined in the Animal class (eat and sleep) can also be implemented in these classes to reinforce the concept of polymorphism (figure 17). The idea of general characteristics and specific

characteristics could continue to be exercised by the lecturer by implementing special methods for the classes used as examples. Those methods could include actions such as a kick method for the Horse class and a bite method for the Snake class, for instance.

```
class Horse extends Animal{
  ...
  @Override
  void move(int distance){
    print("Trotting for " + distance +
" miles");
  }
}

class Snake extends Animal{
  ...
  @Override
  void move(int distance){
    print("Crawling for " + distance +
" miles");
  }
}
```

Figure 17 - Classes might have different implementations for the same inherited methods.

Object manipulation. The Passion category can be useful to explain how objects can be manipulated by other objects. The phrase used as an example to be categorized in table 1 was: "A male white Arabian horse, with four legs and 185.5 cm in height is running while being mounted on by its owner". The substance in this example is *horse*, whereas its passion (action being suffered) is "being mounted on" (table 1). This can be codified as well. For that, a Human class will be created and it will have a method to mount a horse. (figure 18). The other methods pertinent to an animal were suppressed for this figure.

```
class Human extends Animal {
  String name;
  String profession;

  Human(String name, String profession){
    super();
    this.name = name;
    this.profession = profession;
  }
  void mount(Horse horse){
    print(name + ", the "+ profession + ", mounted a " +
horse.breed + " horse! ");
  }
    ...
}
```

Figure 18 - A Human class is defined as a type of Animal as well.

This piece of code defines that a species can suffer actions by another species. More accurately, a Human object can manipulate a Horse object – it can access the horse's breed. This possibility of information hiddenness shown in figure 18 is the core idea and the biggest advantage of the OOP. Finally, there should be a demonstration of the program running (figure 19). The expected output is illustrated in figure 20.

```
class Main {
  void main() {
    Horse horse = new Horse("white", "male",
4, 185.5, "Arabian");
    Human human = new Human("black", "male",
2, 190.0, "Bob", "Jockey");

    human.mount(horse);
  }
}
```

Figure 19 - A program in which a Human instance manipulates a horse instance.

```
Bob, the Jockey, mounted the arabian horse!
```

Figure 20 - The desired outcome of the program.

Conclusion. This is basically how the Object-Oriented Paradigm functions. We define classes of objects and try to abstract them into more generic classes. Classes can hold specific data and exchange them with other classes.

This should be the end of the second module and the introductory part of the course. In this module, the

basic concepts of OO programming were explained in light of what the students learned in the previous Grammar module. Hopefully, students will finish the two introductory modules with a more sound understanding of how OOP works and, more importantly, why it works. The methodology to be used from now on to teach what is left from OOP is up to the lecturer.

6.1.2 The Iterative Cycle Methodology

A good methodology for the teaching of OOP is the Iterative Cycle [3]. This methodology utilizes both abstract representation and detailed programming experience in order to provide abstract thinking skills to novice students learning OO. This is accomplished by iterating over two basic steps, which are:

1. Explaining a target OO concept through modeled examples built with the aid of UML – the abstract step.
2. Code the example modeled in the previous step – the concrete step.

In practice, for teaching an OO concept, the lecturer will initially draw the relevant entities' shapes and their associations using UML and discuss their theoretical meaning (the abstract step). Immediately after, they will write the basic code which represents the diagram, and conclude with inner detailed code insertion (the concrete step). The students are expected to absorb the equivalence of the two views (model and code) and become familiar with the simplified abstract representation [3]. Once those steps are completed, the cycle starts over for the teaching of the next concept. It is important to note that the authors of the Iterative Cycle Methodology described a successful application of this approach as the student being able to absorb the equivalence of the model and the code implementation. This perception of equivalence is also the goal of the first methodology proposed in this paper.

The Grammar can be useful during the abstract step. The following proposed methodology is a combination of the first approach (the Introductory Module) and the original Iterative Cycle approach. Instead of dividing the introductory part of the course into two parts as proposed in the Introductory Module approach (one part for theory and the other, for practice), this alternative doesn't divide theory and practice so strictly. For each concept explained, the practice will be taught immediately after, as in the Iterative Cycle methodology. The course should start similarly to what was proposed before, that is, introducing how the human language works. What follows is an example of the application of this approach to the teaching of the concept of a class. The chosen class is, once again, that of a horse.

After introducing what is a language, an individual, and an essence, just like in the previous approach, the teacher should then introduce the concept of a species. As seen, a species in Grammar is equivalent to a class in a programming language. The lecturer will illustrate a species with a drawing, then abstract it into a UML model, and, finally, convert it into code. Figure 21 shows the flow of this cycle.

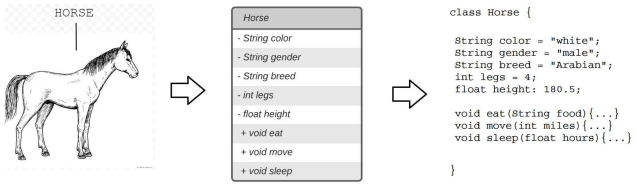

Figure 21 - The flow of this Iterative Methodology approach for teaching the concept of a class.

The illustrated species will be modeled as a UML class, which will end the abstract step of this cycle. The concrete step will follow immediately when the teacher writes the class represented in the UML. The cycle is then complete, and the next concept, that of an object, will follow the same steps. The Ten Categories of Aristotle can also be used in this approach similar to how it was used in the previous approach. It can be introduced before the first modeling with UML and applied to the construction of a table such as table 1. The table can then be used to support the design of the UML model.

After each cycle, the students will be able to grasp the equivalence between the concrete entity, its UML model, and the code.

7. CONCLUSION

This paper demonstrated the correlation between Object-Oriented Paradigm and the classic Trivium. This correlation hints towards Trivium being a promising teaching tool for introducing the OO

programming paradigm. In face of that, two approaches for teaching this paradigm based on Trivium classical teaching were proposed in this work. Although it is not possible to say whether those approaches are indeed efficient, because they weren't tested in real subjects, what goes beyond the scope of this research, this validation will be addressed as a future work.

8. ACKNOWLEDGMENTS

Our thanks to the artists of the drawings used in this paper.

9. REFERENCES

[1] KÖLLING, M (1999). The Problem of Teaching Object-Oriented Programming, Part 1: Languages. Journal of Object-Oriented Programming, 11 (8). pp. 8-15.

[2] CHEAH, C. S. (2020). Factors Contributing to the Difficulties in Teaching and Learning of Computer Programming: A Literature Review. Contemporary Educational Technology, 12(2), ep272.

[3] HADAR, I., HADAR, E. (2006) Iterative cycle for teaching object oriented concepts: from abstract thinking to specific language implementation.

[4] SIVASAKTHI, M., RAJENDRAN R. (2011) Learning difficulties of 'object-oriented programming paradigm using Java': students' perspective.

[5] ARMSTRONG, DEB. (2006). The quarks of object-oriented development. Commun. ACM. 49. 123-128. 10.1145/1113034.1113040.

[6] WIRFS-BROCK, R., WILKERSON, B., WIENER, L. (1990) Designing Object-Oriented Software.

[7] BYRNE, P., & LYONS, G. (2001). The effect of student attributes on success in programming. SIGCSE Bull., 33(3), 49-52. https://doi.org/10.1145/507758.377467.

[8] ROBERTS, ERIC & ENGEL, G. & CHANG, CARL & CROSS, J. & SHACKELFORD, R. & SLOAN, ROBERT & CARVER, D. & ECKHOUSE, RICHARD & KING, W. & LAU, FRANCIS & SRIMANI, PRADIP & AUSTING, R.. (2001). Computing curricula 2001: computer science. ACM Transactions on Computing Education / ACM Journal of Educational Resources in Computing - TOCE/JERIC.

[9] BENNEDSEN, J. (2008). Teaching and learning introductory programming : a model-based approach.

[10] BLUEDORN, H. (1993) What Is the Trivium?

[11] JOSEPH, M. (1948) The Trivium: The Liberal Arts of Logic, Grammar, and Rhetoric.

[12] SAYERS, Dorothy (1947) The Lost Tools of Learning.

[13] COMENIUS, J. A (1649) Magna Didactica.

[14]TEIRA-LAFUENTE, J., GIL-GONZÁLEZ, A.B, REBOREDO, A.(2021) From Trivium to Smart Education. In: Herrero, Á., Cambra, C., Urda, D., Sedano, J., Quintián, H., Corchado, E. (eds) The 11th International Conference on EUropean Transnational Educational (ICEUTE 2020). ICEUTE 2020. Advances in Intelligent Systems and Computing, vol 1266. Springer, Cham. https://doi.org/10.1007/978-3-030-57799-5_2.

[15] GRIES, D. (2002). Where is programming methodology these days? SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 34(4), 5-7.

[16] BUDD, T. (2000) Understanding Object Oriented Programming with Java.

[17] STUDTMANN, P ,(2012) Aristotle's Categorial Scheme, in Shields (ed.) 2012, pp. 63–80.

[18] HALPERN, J., HARPER, R., IMMERMAN, N., KOLAITIS, P., VARDI, M., & VIANU, V. (2001). On the Unusual Effectiveness of Logic in Computer Science. Bulletin of Symbolic Logic, 7(2), 213-236. doi:10.2307/2687775.

[19] KOÇ, H.; ERDOĞAN, A.M.; BARJAKLY, Y.; PEKER, S. UML Diagrams in Software Engineering Research: A Systematic Literature Review. Proceedings 2021, 74, 13. https://doi.org/10.3390/proceedings2021074013 .

[20] KNUDSEN, J. L., & MADSEN, O. L. (1988). Teaching object-oriented programming is more than teaching object-oriented programming languages. ECOOP '88 European Conference on Object-Oriented Programming, Oslo, Norway. 21-40.