Learning Object-Oriented Programming Concepts Through Visual Analogies

Victor Lian, Elliot Varoy, and Nasser Giacaman

***Abstract*—Object-oriented programming (OOP) is a widely used programming paradigm in modern software industry. This makes it an essential skill for students in many disciplines to learn. However, OOP is known to be challenging to learn and teach due to its abstract nature. Studies have shown that students often face difficulties and develop misconceptions in multiple aspects when learning OOP. This paper presents a systematic way of developing a teaching tool that uses a combination of visualization and analogies to help students overcome these barriers and understand the OOP concepts better. To ensure the tool would have pedagogical value and novelty, we first reviewed the educational theories on using analogies and visualization, as well as numerous existing tools. A set of activities in the form of code snippets were then designed to target these misconceptions and difficulties, along with a set of analogies and their interaction mechanisms that mapped to the OOP concepts. A teaching tool was then developed based on those designs and evaluated with engineering students in a CS2 course (*n* = 253). The results and our analysis show that a statistically significant improvement was achieved in student understanding and confidence through interaction with VOOPA exercises. Similar gains were also observed using more traditional methods. No statistically significant positive difference in understanding and confidence can be attributed to use of VOOPA through the current study.**

I. Introduction

OBJECT-ORIENTED programming (OOP) is a widely used programming paradigm in modern software industry [1]–[3]. Because of its popularity, it has become a fundamental part of software development, and is listed as a core topic to be covered in any computing curriculum [4]. However, OOP was found to be difficult to learn and teach. Standard programming languages are textual based and have a lot of syntactical complexities that distract students from understanding the underlying OOP concepts [2], [5]. The abstract nature of the concepts also means that students easily develop misconceptions around OOP [2], [5]–[8]. Given that students continue to face common difficulties, this indicates a need to investigate further how OOP learning can be facilitated.

The integration of technology with teaching provides more opportunities for the development of interactive and engaging learning methods. For example, some existing software tools utilize the power of visualization to help students [9]–[12], with successful teaching been carried out using those tools [1], [13], [14]. Their success is closely related to the difficulties students faced. For example, students often find visualizing the states of an OOP program difficult [1], so many tools try to visually present state transitions of an OOP program. Some tools work like an advanced IDE which allows students to write a program visually through a graphical user interface [15]. This helps students manage the syntactical difficulties of general purpose programming languages like Java and C++.

The authors are with the Department of Electrical, Computer, and Software Engineering, The University of Auckland, Auckland 1010, New Zealand (e-mail: vlia679@aucklanduni.ac.nz; evar872@aucklanduni.ac.nz; n.giacaman@auckland.ac.nz).

Studies have shown that visualization is great for learning abstract concepts [16], however, there are still some threats to its pedagogical value. First of all, the learner needs be actively engaged in the visualization in order to gain its educational value [9], [16]. Another problem is that, based on constructive alignment theory [17], activities around the visualization should be closely aligned with the learning outcomes (in this case the OOP concepts) for them to be most effective. Many tools focus on visualization of a specific program or the objects in it instead of directly aligning the activities with the underlying OOP concepts, which would be more effective in terms of teaching OOP concepts.

Analogies have a long history of being used as a learning aid, they have been proven to be useful in many science teaching programs [18]–[21], and also in computing education [22], [23]. Studies have shown that analogies are particularly effective in learning abstract concepts because they make abstract information more sensible and concrete [24]. OOP concepts are abstract, and because of this abstract nature, they can be represented through analogies. Analogies hide away the technicalities of a program and extract the concepts behind it. By using analogies, things that students are already familiar with can be related to a new unknown domain, thus allowing a transfer of knowledge to happen more easily [25]. However, analogies need to be designed carefully, because students can easily misinterpret them and develop misconceptions [22].

Given the combined successes and shortcomings of existing solutions, this research proposes combining both visualization and analogies to work together in helping students understand OOP concepts better. Analogies make abstract concepts easier to understand but there is the risk of misinterpretation; visualization can help mitigate this risk. If we present the analogy visually, and carefully model it to match the intention, then the chance of it being misinterpreted (or at least “mis-visualized”) is hopefully reduced. Based on those ideas, we present *Visualization of OOP concepts with Analogies* (*VOOPA*) [26]. The evaluation was carried out with 253 students in a CS2 course.

II. Literature Review

This section summarizes literature related to issues around OOP learning, analogies, visualization, and existing tools.

A. OOP Concepts, Misconceptions, and Difficulties

Armstrong [27] suggested that there is a lack of consensus on what constitutes OOP. Different studies have used different sets of concepts to describe it. Despite that, the Computer Science Curricula 2013 [4] provides good guidance on what students should understand about OOP. It identifies *objects with state and behavior, classes with fields, methods and constructors, inheritance, and dynamic dispatch* as knowledge that students "must have" in any computer science degree, while *subtyping, encapsulation, and collections* are listed as "should have" knowledge [4].

OOP is often recognized for being difficult to learn and teach, resulting in it being extensively studied by educators. After exploring a wide range of studies incorporating students and experts opinions, the difficulties and misconceptions were identified and partitioned into seven categories shown in Table I. It is worth noting that some studies have conflicting opinions on which misconceptions and difficulties students have. For example, the misconception "Conflation between class and object" was found by Holland *et al.* [8], Ragnois and Ben-Ari [28], Sanders and Thomas [29] and Sanders *et al.* [30], but Thomasson *et al.* [31] and Xinogalos [13] explicitly stated that this misconception was not found in their study. Such opposing results might have been caused by how studies were conducted, how the OOP courses were taught, or just general differences from cohort to cohort. In this research, the tool will aim to cater for as many categories as possible to make it valuable in different teaching settings.

Variation theory [32] suggested that in order to discern a critical aspect of a phenomenon, one must first experience variation in dimensions that correspond to that aspect. Holland [8] suggested that one reason students might develop misconceptions is due to lack of variation in teaching examples. Variation theory also suggests that when particular aspects of a phenomenon vary, and other aspects are kept constant, the varying aspects are discerned [33]. These ideas encourage VOOPA to use code snippets that incrementally build on the same example to cover a range of different topics.

B. Analogies in Education

Various theories have been developed as to why analogies help students understand concepts better [25], [40]. Application of analogies in teaching have been proven to be successful in various domains [18], [21], [24], as well as in computer science [22]. Analogies are useful in education because they are fundamental cognitive tools [21], [41], [42]. Furthermore, Simons [24] suggested that they are useful because they have three functions: *concretizing*, *structurize new information*, and *active assimilation*. Their theory was supported quantitatively through a series of experiments on students reading text with and without analogies. Amongst the three functions, the concretizing function is particularly interesting for this research. Since OOP concepts are inherently abstract (therefore contributing to the difficulty in learning them), analogies could help students concretize them, making them easier to grasp. Brown and Aradalan [41], [43] also suggested analogies can help students overcome misconceptions by restructuring incorrect information and forming a new explanatory model.

Gentner's Structure–Mapping Theory [44] provided some insights on how to develop effective analogies. It suggests when comparing a source analog to a target, we tend to map relations between objects in each analog, especially higher order relations. This means instead of using multiple independent analogies to represent different aspects of a system, using another system as an analogy is preferable. Thagard [40] provided similar theory on using analogies in education, suggesting that the source and target analogs should have *semantic similarity* and *structural correspondence*. Various studies have found that analogies, if not used carefully, can lead to misconceptions. This is because a source analogy, no matter how carefully chosen, can never match the target perfectly [22], [40] and disanalogous features will exist inevitably. Orgill and Bodner's [45] suggested a potential solution with visualization. It helps students overcome misconceptions by demonstrating exactly which parts of the analogy are alike.

C. Visualization

Visualization has been used extensively in computing education. The most common forms include visualization of algorithms, data flow and program execution [46]. Visualization was used by Sorva [47] in programming education in the form of visual program simulation. Sorva suggested that this form of visualization is theoretically sound and is helpful to students demonstrated by an empirical evaluation. Sorva's findings gave us confidence in using visualization and program tracing as part of our research. In a meta-study of existing visualization tools, Hundhausen *et al.* [48] suggested that when working on a visualization tool, one should not only focus on its *expressiveness*, but also consider its *effectiveness*. They suggest that the way students use visualization is actually more important than the content of the visualization itself, and that the amount of effort (or *engagement*) students have plays a vital role in their learning. This theory on engagement is later confirmed by many other studies and has enormous implications on visualization tools developed afterwards [9], [46], [49], [50]. This means that visual tools should ensure students are actively participating instead of passively watching.

D. Existing Tools

Modern visual programming environments include the likes of Scratch [51] and Blockly [52]. These block-based languages have shown great success and popularity with younger learners. Alice is one of the prominent learning tools that has been used extensively in teaching OOP [1], [10]. Like this research, Alice is a three-dimensional (3-D) visualization tool. Cooper [9] revealed that one of the key pedagogical design feature in Alice is to make state visible to students as much as possible. Cooper also argued since the students created their own animation, a higher level of engagement was achieved. BlueJ is also a well known teaching tool in the OOP domain. Kölling *et al.* [14] suggested that the three key goals for BlueJ were to provide a truly OOP environment, to allow interactions with objects for experimentation, and as a

TABLE I
LIST OF OOP DIFFICULTIES AND MISCONCEPTIONS FROM THE LITERATURE

Category	Misconception (M) and Difficulty (D)
1. Class and Object	a. Understanding the nature of a class as a template for creating objects (D)
[3], [8], [13], [28]–[31], [33]–[35]	b. Understanding the relationship between class and object (D)
	c. Conflation between class and object (M)
	d. Conflation between objects and variables (M)
	f. A class is a collection of objects (M)
	g. An object is just a piece of code (M)
2. Identity and Attribute	a. Understanding the relationship between an object, its identifier and its attribute (D)
[3], [8], [28], [29], [31], [36]	b. Two variables cannot hold the same reference to an object (M)
	c. Two objects of the same attribute must be the same object (M)
	d. An attribute can be used as an identifier for an object (M)
	e. An object's textual (string) representation can be used as its identifier (M)
3. Methods and Invocation	a. Understanding a method can be invoked with any object from that class (D)
[6], [28], [29], [37]	b. Method invocation using the dot operator (D)
	c. Understanding method overloading (D)
	d. Methods can add attributes (M)
	e. Methods can only perform assignments instead of message passing (M)
	f. Methods cannot be the same name even if they are in different classes (M)
	g. Methods can only be invoked once (M)
4. Constructor and Instantiation	a. Understanding the need to invoke a constructor with the "new" keyword and the memory allocation that comes with the instantiation (D)
[6], [28], [29], [36], [37]	b. Understanding the default constructor (D)
	c. Understanding the copy constructor (D)
	d. Constructors are automatically called (M)
	e. Constructors can only be called once (M)
5. Inheritance and Polymorphism	a. Understanding the purpose for inheritance (D)
[34], [37], [38]	b. Understanding virtual functions and method overriding (D)
	c. Understanding how methods are called by dynamic dispatch (D)
	e. A downcast is always possible on the inheritance hierachy (M) (D)
	f. The variable type determines method dispatch (M)
	g. Overriding eliminates the overriden method (M)
6. Composed Class and Encapsulation	a. Manipulating a class storing a reference to another (D)
[8], [13], [28]–[31], [36]	b. Understanding the need for information hiding (D)
	c. Simple class objects will be created when the composed class object is created (M)
	d. An object cannot be the value of an attribute for another object (M)
	e. Methods that were declared in the simple class must be declared again in the composed class (M)
	f. Attributes defined in the simple class automatically becomes attributes in the composed class (M)
7. Program Execution	a. Understanding how objects communicate and interact during program execution (D)
[28]–[30], [35], [39]	b. Understanding how methods operate on attributes (D)
	c. Understanding the relationship between methods and objects (D)
	d. Understanding the role of the "main" method with respect to program state (D)
	e. Methods execute in the order that they were defined in the class (M)

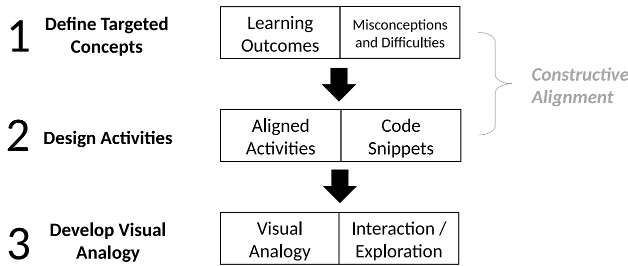


Fig. 1. The overall design process. The first two stages follow traditional constructive alignment. We propose the addition of the third stage—interaction with a visual analogy.

simplified IDE interface. Greenfoot [53] was developed as an extension to BlueJ to improve interactions on objects. Similar to BlueJ and Greenfoot, AGUIA/J [54] acts like an IDE. The difference is they present textual information visually and allow interactions such as giving instructions to objects. They are more general and flexible in the sense that teachers decide what activities are designed. For example, one activity could explain methods using a robot, while another activity could use houses to illustrate the concept of inheritance. This research takes a different approach, in that we try to demonstrate a specific pre-designed set of OOP concepts that are known to be difficult for students. The activities designed will directly map to these concepts through the use of an analogy.

III. Design

Fig. 1 illustrates the design process underpinning the VOOPA development. It includes three stages: defining targeted concepts, designing activities, and developing the visual analogy. The first two stages are inspired by constructive alignment [55], which suggests learning is more effective when activities are closely aligned with the learning outcomes defined. This well-established theory was followed to maximize value of the VOOPA activities, by explicitly targeting misconceptions and difficulties identified in the literature. Once the learning outcomes and the activities were set, the visual analogy was developed to be used alongside the activities.

A. Define Targeted Concepts

The Computer Science Curricula 2013 (CS2013) [4] specifies a set of learning outcomes to be covered under the *PL/Object-Oriented Programming* knowledge unit. None of the topics within this unit are considered *Elective* hours, and are listed as either *Core-Tier1* or *Core-Tier2* hours. With that said, some topics such as interfaces and abstract classes were not selected; these topics rely on the understanding of more general concepts such as inheritance and polymorphism, which were the primary focus of this work. More general topics like collections and usage of libraries are ignored in this work. The sub-topics selected from this knowledge unit include:

[Core-Tier1]

- *Object-oriented design*
  - *Decomposition into objects carrying state and having behavior*
  - *Class-hierarchy design for modeling*
- *Definition of classes: fields, methods, and constructors*
- *Subclasses, inheritance, and method overriding*
- *Dynamic dispatch: definition of method-call*

[Core-Tier2]

- *Subtyping (cross-reference PL/Type Systems)*
  - *Subtype polymorphism; implicit upcasts in typed languages*
  - *Notion of behavioral replacement: subtypes acting like supertypes*
  - *Relationship between subtyping and inheritance*
- *Object-oriented idioms for encapsulation*
  - *Privacy and visibility of class members*
  - ~~*Interfaces revealing only method signatures*~~
  - ~~*Abstract base classes*~~

Also from the CS2013 guidelines, the following relevant learning outcomes were selected:

[Core-Tier1]

LO1. Design and implement a class. [Usage]
LO2. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Usage]
LO3. Correctly reason about control flow in a program using dynamic dispatch. [Usage]

[Core-Tier2]

LO5. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Familiarity]
LO6. Use object-oriented encapsulation mechanisms such as ~~interfaces and~~ private members. [Usage]

B. Design Activities

The learning outcomes within CS2013 are high-level overviews, and not as detailed as the misconceptions and difficulties identified in the literature. The activities were therefore designed to target the misconceptions and difficulties, which would inherently cover the CS2013 learning outcomes. Table II presents two example activities; each is composed of the learning outcome and misconception or difficulty it covers along with the code snippet for the activity. The design of the activities followed the *variation theory* introduced in Section II-A, which suggests to keep most aspects constant while varying only some particular aspects as this helps those varying aspects be discerned [33]. Therefore, the code snippets chosen had a running theme of a `Person` class. The class was introduced in the first activity and gradually expanded by adding more fields, attributes, and eventually extended with subclasses. This ensures that code snippets between two consecutive activities are very similar, and the difference is often used to demonstrate a particular new learning point. This design not only follows variation theory, but it also reduces the time needed for students to comprehend the code snippets.

C. Develop Visual Analogy

The visual analogy development involves both the static aspect of having an analogy, and the dynamic aspect of how the analogy could be interacted with. Since the aim of the analogy was to present OOP concepts, they were designed to be language independent. The overall design follows *Gentner's Theory* mentioned in Section II-B, which led us to develop a

TABLE II
EXAMPLE ACTIVITIES WITH THEIR TARGETED LEARNING OUTCOMES AND MISCONCEPTIONS/DIFFICULTIES

Activity	Learning Outcomes	Misconceptions/Difficulties (Based on Table I)	C++ Code Snippet (`Student` extends `Person`)

Constructor with parameter	LO1. Design and implement a class
	
	Understand how an object is instantiated through its constructor
		Constructor and Instantiation (4a, b)	`Person *p = new Person("Sam");`
			`...`
			
			`class Person {`
			`    private string name;`
			`    public Person(string name) {`
			`        this->name = name;`
			`    }`
			`}`

Calling an inherited method	LO2. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses.
	
	Understand attributes are inherited when subclassing
	
	Understand how to call an inherited method
		Inheritance and Polymorphism (5a, b, c)	`Student *s = new Student("Alex", "Engineering");`
			`cout << s->getName() << endl;`
			`...`
			`class Person {`
			`    protected string name;`
			`...`
			`    public string getName() const {`
			`        return this->name;`
			`    }`
			`}`

cohesive set of analogies that revolve around a main theme. Fig. 2 will be used to help illustrate the analogy and how it maps to core OOP concepts. In this simple `Example` class, there are two overloaded constructors (one has no parameters, the other has a parameter `p1`). There are two **private** fields `f1` and `f2`, and two instance methods: `foo` (**private** with a local variable `v1`) and `bar` (**public** with a parameter `p2`).

1) Instance and Classes: As we were trying to map object-oriented concepts, the concept of an object (instance) naturally becomes the main focus point. Tanielu *et al.* [56] introduced the notion of a *house* as an analogy for an **instance**. We determined that this is an appropriate analogy to use for a few reasons. First, the concept of a house is easy to understand, making it pragmatic. Second, instances and houses have semantic similarity. Instances have states that are usually stored internally and are not visible to the rest of the program according to encapsulation. Similarly, houses can store things that are not visible to the outside world. Since we have decided an instance is a house, the notion of a **class** naturally becomes the *blueprint (or architectural drawing plans)* of the house.

Instance **methods** are mapped to *rooms* of the house, which are located within the house next to each other in a somewhat circular orientation. With this orientation, each rooms provide for *internal* and *external* doors, which map to **private** and **public** accessibility. In this regard, constructors are considered as special methods—but methods nonetheless. If a method is declared as **public** (e.g., `Example()` and `bar()`), it will have both an internal and external door. Otherwise (e.g., `foo()`), it will only have an internal door (i.e., the method is only accessible from inside the instance).

A **variable** is mapped to a *box* with a label on it specifying its name and type. This analogy is commonly used to denote how values may be stored inside a variable. This idea is consistently used for instance fields, parameters, and local variables; what differs, is where that box is located. In the case of an instance field, the box would be placed at the centre of the house to reinforce the idea that fields are accessible by all methods of the instance (e.g., `f1` and `f2`). **Parameters** are mapped as boxes located on the *window sill* of the room

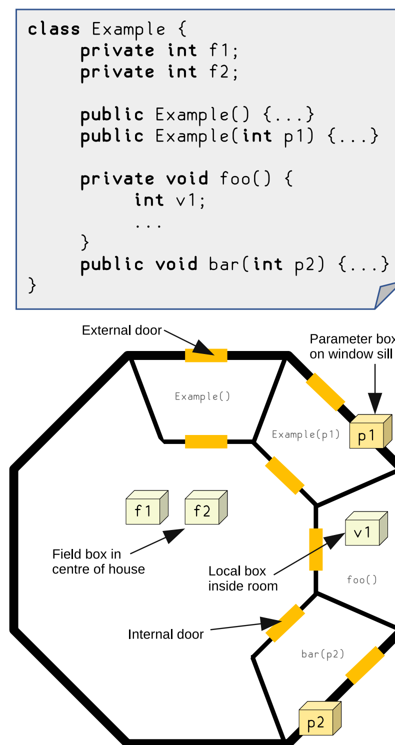


Fig. 2. Mapping of core OOP concepts to the house analogy, illustrated as the plan view of a house. The small rooms correspond to methods, accessible via the respective doors. Variables are represented as boxes, located based on whether they are fields, parameters, or local variables.

(e.g., `p1` and `p2`), allowing **method arguments** to be placed in them before entering the room, but still be accessible once inside the room. For a method-specific **local variable** (e.g., `v1`), the box is located inside the room and is only accessible to that room (unlike the field boxes in the center of the house).

Calling a public method was mapped to the interaction of *entering a room through its external door*. This mapping is analogous to doors having semantic similarity of "entry points" of methods. The room represents the method body, and once inside a method, the internal door opens to reveal

other methods and fields that are accessible. Structural correspondence is also achieved as the process of walking through the room to access items inside the house matches the process of calling a method to access instance fields.

2) Inheritance and Polymorphism: In order to cater for **inheritance**, the analogy required an extension that seemed natural to the core analogy described above. The notion of a *multi-level house* seemed to naturally support this. A **child class** extends a **parent class**, in very much the same way an *additional floor* would extend and be built on top of an existing *(base) floor*. In fact, the terms "parent class" and "base class" are used synonymously in OOP terminology. A child class may access information from both the parent class and the child class itself, and a multi-level house allows each level to represent information from the respective class while easily navigating to the parent class level.

Fig. 3 illustrates the core ideas that allow the analogy to support inheritance and polymorphism. Since inheritance allows information from multiple classes to be accessed from one instance, there needs to be a mechanism to move between the levels of the house. *Lifts* were used to represent the idea that **inherited attributes are accessible** between the parent and child classes. A lift is appropriate not only because it creates transportation between the two levels, but also because it allows user initiated actions just like a developer attempting to access fields and methods from both classes at compile time. To ensure both **public** and **private** attributes can be accessed, our design had an *external lift* and an *internal lift*. The external lift allows for public operations on an instance (e.g., accessing `m3()` from outside the class), while the internal lift allows for private operations within an instance (e.g., accessing `f1` while on the `Child` level). These operations needed to be treated separately as they have different rules in OOP.

Polymorphism is the ability for an object to take multiple forms, and its most common use is when a child class instance is stored in a parent class type variable. The main thing we needed to map for polymorphism is the dynamic dispatch mechanism: determining which method to call at runtime rather than at compile time. Since this also involves accessing an attribute from a different class, movement between the floors is needed once again. We decided to map **runtime polymorphism** to *moving platforms* placed in front of method doors. Unlike inheritance where the developer knowingly accesses specific members at compile time, polymorphism is the runtime behavior that is automatically actioned and requires no explicit coding action initiated by the programmer. Therefore, the lift was deemed unsuitable here as it was used to represent user-initiated action. The moving platform therefore moves automatically when dynamic dispatch takes place. An example would include calling the `m2()` method for an instance of `Child` on a variable declared of type `Parent`. Programmatically (and also within the analogy), it appears as one is about to enter the room for the `Parent` level, but instead the platform moves up to the `Child` level (thereby invoking the implementation of the `Child` class).

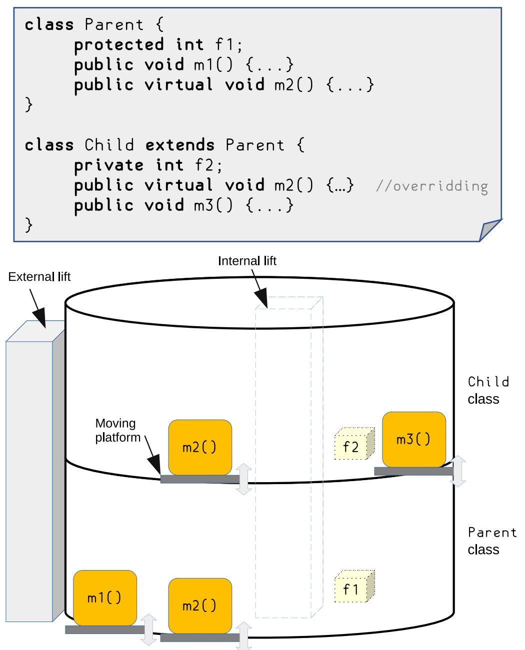


Fig. 3. Mapping the core concepts of inheritance and polymorphism by extending the house analogy. Multiple floors represent classes extending a base class, with external and internal lifts used to represent developer-initiated access to inherited components. Polymorphism (dynamic dispatch) is represented using moving platforms in front of method doors.

D. VOOPA Application

VOOPA [26] was implemented as a tool to present the core analogy elements to students visually, and also allowing them to interact with those elements. We decided to take a first-person virtual world approach. To ensure students understand the concepts presented in each activity, we set the goal for each activity as tracing through the code snippets provided step-by-step. It was therefore similar to a game, where the main quest is to perform actions defined by the code. The actions students can take matches with the designed interactions with the analogies. For example, if the code snippet shows that a method is to be called on an object, then the student must walk to the "method room" and open its external door. Students would also have the chance to explore the virtual world as they find where they need to go. There are, however, restrictions on where they can go and what they can see. For example, students would not be able to visit or see any internal fields before entering a method room due to encapsulation.

Fig. 4 shows the general user interface when a student uses VOOPA. Apart from the main virtual world, several panels were used to indicate different information to the student. The *code panel* on the right displays the code that the student is supposed to trace. To help students understand their progress, the next line of code to be executed is highlighted in green. The *console* at the bottom right corner displays any content from the print statements. The *minimap* at the bottom left shows a map of the house to help students orient in the virtual world. It also shows the current position of the student with a yellow arrow, and the current floor the student is on (which is helpful for multi-level houses). The *action indicator* at the top of the screen shows that an automatic action is in progress (e.g.,

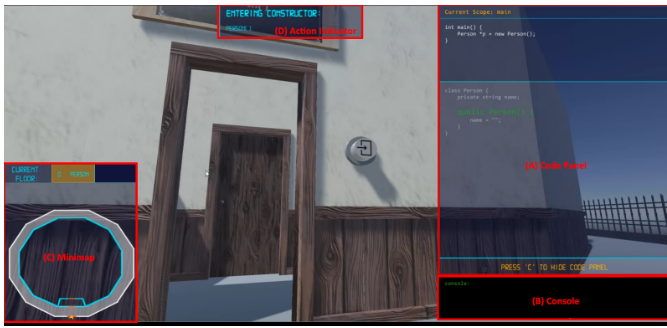


Fig. 4. Screenshot showing the VOOPA virtual world, as well as (A) the code panel, (B) the console, (C) the minimap, and (D) the action indicator.

when a method is called, the student would be taken into the room automatically).

Fig. 5 shows the steps involved in one of the introductory activities, in this case the interactions demonstrating how an instance is created in VOOPA. This involves the student declaring a variable, creating an instance, and entering the constructor to initialize the instance field. Fig. 6 shows the steps for a more advanced activity, by calling a polymorphic method. In this case, the variable is declared as the base-class type (hence only the house's bottom level is accessible in the analogy). The method being called is actually overridden (by the child class) and so polymorphism is "taking over."

IV. Evaluation

The primary goal of the evaluation is to assess the pedagogical effectiveness of the VOOPA learning tool. To achieve this, *learning gain* and *confidence gain* were used as measures. The learning gain was determined by assessing students' conceptual understanding of core OOP concepts, while confidence gain was determined by measuring students' self-efficacy (self-perceived competency). The evaluation also looked at other aspects related to software usability and student engagement. These goals are summarized in the following research questions:

RQ1. How effective is a visual analogy tool in helping students understand OOP concepts and overcome misconceptions?

RQ2. How effective is a visual analogy tool in helping students gain confidence in visualizing OOP concepts?

RQ3. What are the usability considerations in developing a visual analogy tool?

A. Methodology

The participants all came from a CS2 course. It was a compulsory programming course for all second-year engineering students across three different majors: computer systems engineering, electrical and electronics engineering, and mechatronics engineering. Using C++, the 12 week course covered OOP in the first six weeks followed by data structures and algorithms in the second six weeks. The evaluation discussed here was carried out in weeks 5–6 of the course. It is important to note that these students are not in a software major, meaning that programming might not be strength or interest for many (the software engineering major takes separate courses). The application was given as a supplementary tool for students to reinforce their understanding. Almost all of the OOP concepts covered in the application were first presented to students during lectures, before the application was delivered. As such, students were not expected to be learning the OOP material from the application itself.

TABLE III
Types of Evaluation Questions

Name	Question type	Pretest	Posttest	Follow-up
Knowledge questions	Multi-choice	✓	✓	
Confidence questions	Likert scale	✓	✓	✓
Usability questions	Likert scale		✓	✓
Open questions	Open text			✓

Of the 296 students enrolled in the course, 284 students (96%) at least accessed the application. However, for the purposes of this evaluation, only the 253 students (85%) that completed all the tasks are included in the analysis. The evaluation setup shown in Fig. 7 incorporated a *pretest–posttest design*, a *randomized controlled design*, and a *counterbalanced measures design*. This involved randomly allocating students into one of two groups: the *Treatment group* (VOOPA) and the *Control group* (slides). The pretest–posttest allowed measuring the immediate before-and-after knowledge of both groups. The additional counterbalanced measures design ensured neither of the Control or Treatment groups were disadvantaged. For the Control group, explanatory slides were developed to mimic the same activities presented in VOOPA.

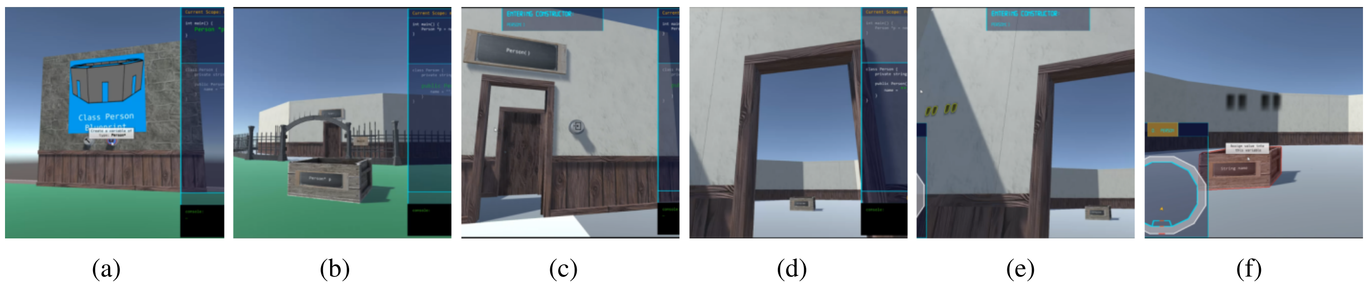
Table III shows the types of questions asked in each stage of the evaluation. The *knowledge test questions* were composed of 18 multiple choice questions on OOP concepts, as might be expected in a mid-term test. These were asked in the pretest and posttest to determine the impact on learning. The questions can be generally divided into conceptual and code analysis ones. The full set of questions are listed in the Appendix.

The *confidence questions* were composed of 14 Likert-scale (five-point) questions on students' confidence in visualizing various OOP constructs. These were asked in the pretest, posttest, and follow-up questionnaire. All questions were in the form of "*I am confident with visualizing <topic>*." The specific topics targeted by these confidence questions are listed in Table VIII. These questions were selected by the authors, based on the target concepts listed in CS2013.

The *usability questions* were asked every time after students engaged with either VOOPA or the slides, while *open questions* were asked when students were exposed to both options. All the pretest, posttest and follow-up questionnaire were asked independent of VOOPA and slides, but were incorporated into the respective method to ensure students answered them in the correct order of the evaluation.

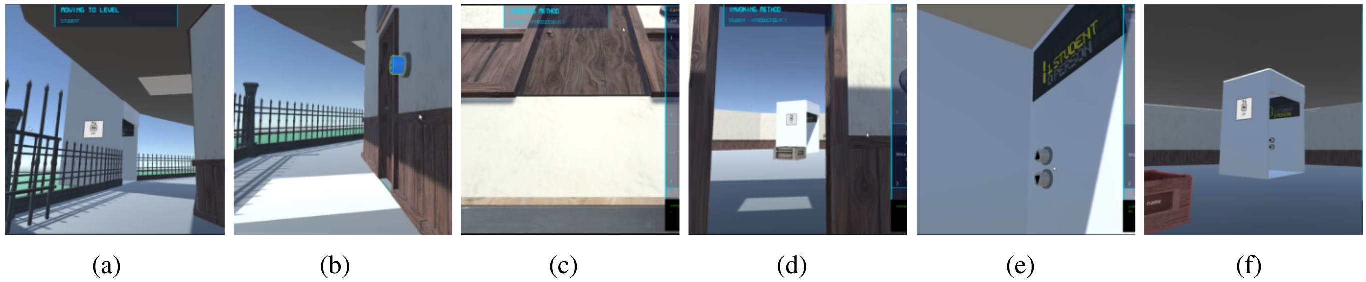
B. Results

1) Overall knowledge test scores: The first null hypothesis is that students do not learn anything using VOOPA. That is, their knowledge performance in the posttest (T_{Po}) is the same



(a) (b) (c) (d) (e) (f)

Fig. 5. An example interaction showing how an object is instantiated. (a) Declaring a variable and creating an instance by pressing the construct buttons underneath the class blueprint. (b) Revealing the instance created. (c) The constructor is invoked. (d) Entering the room representing the constructor. (e) Initializing a field by picking up the empty string literal. (f) Placing the literal in the box located at the centre of the room that represents the instance's field.



(a) (b) (c) (d) (e) (f)

Fig. 6. An example interaction for calling a polymorphic method. (a) An external lift when entering the house, which cannot be accessed due to the instance variable type being declared as a parent type. (b) When attempting to enter the polymorphic method room, the white platform in front of the room will raise. (c) The platform goes to the floor above automatically to enter the method room above. (d) Once inside the child's overridden method, the internal door opens to allow exploration of the fields declared in the child class. (e) An internal lift, which leads down to the base floor to access fields in the base class. (f) Returning back to the child class using the same internal lift once done.

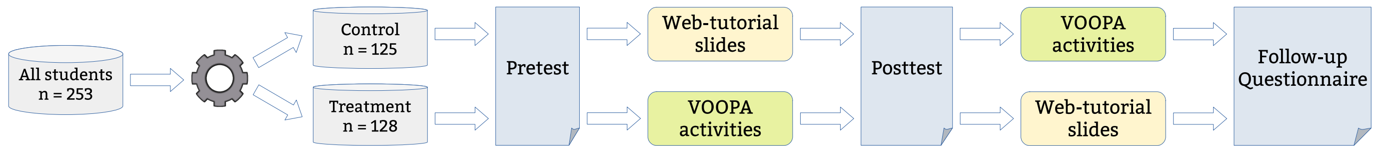


Fig. 7. The evaluation design involved randomly allocating students into two groups: treatment and control. Only the n = 253 students that fully completed the activities are included in the study. After a pretest, the treatment group will use VOOPA whereas the control group will use slides to learn. The pretest-posttest design evaluated differences in learning and confidence across the two groups, while the counterbalanced measures design ensured fairness in learning opportunity.

as their performance in the pretest (T_{Pr}). The before and after score results are presented in Table IV. Shorthand notations are used to represent the data source of the comparison (e.g., T_{Pr} stands for **T**reatment group at the **Pr**etest). Both groups had a p-value of $< .0001$, indicating there is statistically significant evidence that the students' understanding of OOP concepts improved after using the application (i.e., there is a learning gain). We can therefore reject the null hypothesis, as there is statistically significant improvements.

The next null hypothesis is that there is no difference in learning gain between students using VOOPA versus the slides. That is, knowledge performance of the VOOPA group's posttest (T_{Po}) is the same as knowledge performance of the control group's posttest (C_{Po}). The results presented in Table V confirm that both groups showed no statistically significant differences before the intervention, which is expected as the groups were randomly allocated. It is also shown that the differences of the two groups' test scores were still not statistically significant after the intervention. We therefore cannot reject the second null hypothesis of there being no difference between the two methods.

2) Breakdown of individual test questions: In order to understand which questions led to the improvement of the overall score, analysis was carried out on each question in the knowledge test individually. Again, both paired and unpaired comparisons were made. The McNemar's test was used as the paired test for the C_{Pr}–C_{Po} and T_{Pr}–T_{Po} comparisons, with these results shown in Table VII. All the questions that had a statistically significant difference had an increase in accuracy, thus there is no question that suffered a statistically significant decrease in correct response for either group. The Control group had six questions that were improved with statistical significance after learning, while the Treatment group had four. For the unpaired tests, the Chi-Squared Test was used. Only Question 11 in the Posttest had a statistically significant difference between the two groups. 90% of students in the Control group answered this question correctly in the Posttest whereas only 77% in the Treatment group. It is interesting to note that both groups had statistically significant improvement on this question after engaging with the exercises as per Table

TABLE IV
ONE-TAILED PAIRED t-TEST RESULTS FOR TEST SCORES

Name	Pretest Mean	Posttest Mean	Standard Error of Difference	95% Lower Bound	t-value	df	p-value
C_{Pr}–C_{Po}	12.44	13.50	0.195	0.68	5.4467	125	$< .0001$
T_{Pr}–T_{Po}	12.34	13.10	0.174	0.42	4.3966	128	$< .0001$

TABLE V
TWO-TAILED UNPAIRED t-TEST RESULTS FOR TEST SCORES

Name	Control Group Mean	Treatment Group Mean	Standard Error of Difference	Lower Bound	Upper Bound	t-value	df	p-value
C_{Pr}–T_{Pr}	12.44	12.34	0.361	-0.61	0.82	0.2882	253	.7734
C_{Po}–T_{Po}	13.50	13.10	0.386	-0.36	1.16	1.0435	253	.2977

TABLE VI
USABILITY AND ENGAGEMENT RESULTS COMPARING CONTROL AND TREATMENT GROUPS USING THE WILCOXON SIGNED-RANK TEST, SHOWING THE C_{PO}–T_{PO}, AND THE C_F–T_F COMPARISONS

	Posttest C_{Po}–T_{Po}		Follow-up Questionnaire C_F–T_F	
	W	p-value	W	p-value
Q1. app was easy to use	6290	$< .0001$***	6762.5	$< .0001$***
Q2. app was fun and engaging	3959	.9568	4561	.4555

*** Significant at .001 probability level

VII, with the Control group starting at only 69% and the Treatment group starting at 63%.

3) Self-efficacy: Table VIII and Table IX show the results of the one-tailed Wilcoxon signed-rank tests for the Control group and Treatment group respectively. From the results, we can make the following observations:

- For the Control group, after using the slides as their first study method, student confidence increased for all questions except Question 2.
- For the Control group, student confidence continued to increase (for 11 out of the 14 questions) when VOOPA was used after the explanatory slides.
- For the Treatment group, after using VOOPA as their first study method, student confidence increased in 8 out of the 14 questions.
- For the Treatment group, student confidence continued to increase (for 12 out of the 14 questions) when explanatory slides was used after VOOPA.
- For *both* Control and Treatment groups, student confidence increased for all questions after being exposed to both study methods.

4) Usability and Engagement: As usability and engagement were not core items to be evaluated, there were only two five-point Likert scale questions. Those are shown in Table VI. They indicated that there was a statistically significant difference for ease of use. Students generally found the slides easier to use. The results also showed no significant difference in self-reported engagement levels. Usability issues are discussed in more details in the qualitative analysis.

5) Qualitative analysis: The good: Many students generally felt VOOPA helped them understand OOP concepts better:

S1: *The visualization application actually helped me understand most of the things that I was struggling with in class.*

S2: *It was a very good summary of what we went over in lectures. Definitions of the new terminologies became very clear after this exercise.*

Many students were able to appreciate the analogies and liked them, with some of them commenting that the analogies were cohesive, with every analogy revolving around the house:

S3: *Having the same theme throughout (the house) and having each method as its own room helped to understand and retain the knowledge.*

S4: *I liked the way every OOP concept (e.g., pointers, methods) had their own visual analogy.*

S5: *The visualizations and analogies for each concept let me easily understand how the code worked.*

Many students also found VOOPA very helpful for understanding the concepts of inheritance and polymorphism. Some also mentioned it helped clear some of the misconceptions:

S6: *The last few exercises were good at helping me visualize inheritance and polymorphism.*

S7: *I realized that a base class and derived class are more "intimate" than I had previously thought. ... It especially helped to improve my understanding of "permissions" of a pointer of type [base] pointing to a [derived] object.*

S8: *The visualization analogy helped clear up some confusion surrounding pointers and polymorphism.*

Confidence boost was also apparent:

S8: *I have always found programming quite difficult because it is so abstract. It was hard to imagine what is actually going on. By using this visual analogy, it has helped me clear up what variables can be accessed by which instances/class types.*

TABLE VII
McNEMAR'S TEST RESULTS FOR CONTROL GROUP AND TREATMENT GROUP COMPARING PERFORMANCE FOR EACH QUESTION, SHOWING THE C_{PR}–C_{PO} AND T_{PR}–T_{PO} COMPARISONS

Question no.	Topic	Control Group (C_{Pr}–C_{Po})		Treatment Group (T_{Pr}–T_{Po})	
		χ^2	*p*-value	χ^2	*p*-value
1	Class and object—purpose of class	0.08	.7815	2.78	.0956
2	Constructor and Instantiation—purpose of a constructor	1.80	.1797	1.60	.2059
3	Constructor and Instantiation—time of object instantiation	0.47	.4913	0.00	1.0000
4	Program Execution—purpose of methods	0.33	.5637	0.07	.7963
5	Methods and Invocation—purpose of methods	0.17	.6831	0.18	.6698
6	Program Execution—execution order of methods	1.29	.2568	0.53	.4669
7	Methods and Invocation—calling instance methods	0.39	.5316	2.00	.1573
8	Identity and Attribute—purpose of "this"	0.00	1.0000	0.18	.6698
9	Inheritance and Polymorphism—multiple constructor calls	7.53	.0061**	4.00	.0455*
10	Inheritance and Polymorphism—multiple constructor calls	5.00	.0253*	7.11	.0077**
11	Inheritance and Polymorphism—available method calls	19.70	$<$ .0001**	9.76	.0018**
12	Inheritance and Polymorphism—available method calls	6.55	.0105*	0.00	1.0000
13	Inheritance and Polymorphism—available method calls	0.60	.4386	4.50	.0339*
14	Inheritance and Polymorphism—available method calls	9.38	.0022**	0.86	.3545
15	Inheritance and Polymorphism—dynamic dispatch of methods	12.60	.0004**	3.10	.0782
16	Inheritance and Polymorphism—fields accessibility	3.60	.0578	2.95	.0858
17	Composed Class and Encapsulation—method call on composed object	1.65	.1985	3.27	.0704
18	Composed Class and Encapsulation—access field of composed object	0.43	.5127	0.67	.4142

[*] Significant at .05 probability level
[**] Significant at .01 probability level

S9: *It was very useful and make me more confident in my understanding of OOP.*

6) Qualitative analysis: The bad: On the contrary, there was also some negative feedback. It seems that some students did not appreciate VOOPA as a learning tool and prefer the more traditional explanatory slides method:

S10: *Slides with diagrams were good, visual game was bad.*
S11: *I've already done an OOP course and so this was just painful. I wish I hadn't done them, I just feel confused about something I already understood.*
S12: *I found it hard to relate playing the game to programming concepts.*

Usability was a very big issue reported by a lot of students, including those who found the tool useful. This was consistent with the usability comparison presented previously. The cursor not being locked and the frame rate being too low were the two most frequently reported issues:

S13: *The cursor makes it hard to look around sometimes.*
S14: *I got sick of moving floors within the game cos the game was laggy.*

Some students also reported feeling motion sickness after using the tool, and it impaired their ability to learn from it:

S15: *I also got a bad headache from the graphics.*
S16: *I got crazy motion sickness playing this game, having to take a few minutes break after the levels.*

V. DISCUSSION

A. Learning Gain

RQ1: *How effective is a visual analogy tool in helping students understand OOP concepts and overcome misconceptions?*

Student scores on the knowledge test questions had a statistically significant improvement for both the "traditional" learning style (using explanatory slides) and VOOPA. This means a learning gain was achieved by both approaches. This gives some degree of confidence in the value of VOOPA in the learning of OOP concepts. The traditional use of slides also had a statistically significant improvement in test scores. This was not surprising, as slides tend to be more familiar to both teachers and students. The comparisons between the control group and the treatment group revealed that student test scores had no statistically significant differences. This result shows neither form of learning is superior than the other, and VOOPA cannot be used as a replacement for traditional teaching methods. Rather than viewing VOOPA as a contender against traditional modes of learning, we speculate that using *both* forms of learning together would maximize learning. Unfortunately this study cannot verify this, as the test questions were not repeated in the follow-up questionnaire.

Analysis of individual questions showed that questions with significant improvement were all in the *inheritance* and *polymorphism* categories. This is also evident from the qualitative analysis as the majority of students reported inheritance and polymorphism being the most helpful content from VOOPA. The likely reason these topics witnessed most learning benefit is that they are the more complex topics that students struggle

TABLE VIII
CONFIDENCE RESULTS FOR THE CONTROL GROUP USING WILCOXON SIGNED-RANK TEST, SHOWING THE C_{PR}–C_{PO}, C_{PO}–C_F AND THE C_{PR}–C_F COMPARISONS

Ques. no.	Topic	Pretest vs Posttest C_{Pr}–C_{Po}		Posttest vs Follow-up Questionnaire C_{Po}–C_F		Pretest vs Follow-up Questionnaire C_{Pr}–C_F	
		W	*p*-value	W	*p*-value	W	*p*-value
1	Methods and Invocation—methods in general	6191	< .0001***	4163	.2888	7060	< .0001***
2	Class and Object—class versus object	4361	.1481	4085	.3577	5708	< .0001***
3	Class and Object—identify instance with variable	5627	< .0001***	4782	.0187*	6744	< .0001***
4	Constructor and Instantiation—role of constructor	5242	.0007***	4062	.3790	6261	< .0001***
5	Class and Object—instance variables	5205	.0008***	4786	.0182*	6646	< .0001***
6	Class and Object—instance methods	5537	< .0001***	5258	.0006***	7034	< .0001***
7	Methods and Invocation—invoking instance methods	5050	.0031**	5278	.0005***	6844	< .0001***
8	Methods and Invocation—methods modify variables	5366	.0002***	5206	.0009***	6976	< .0001***
9	Program Execution—program flow	6171	< .0001***	5697	< .0001***	7326	< .0001***
10	Inheritance and Polymorphism—inherited members	7083	< .0001***	5298	.0004***	7642	< .0001***
11	Inheritance and Polymorphism—overridden methods	6788	< .0001***	5285	.0004***	7581	< .0001***
12	Inheritance and Polymorphism—dynamic dispatch	6623	< .0001***	5559	< .0001***	7583	< .0001***
13	Composed Class and Encapsulation—member reference	5627	< .0001***	6144	< .0001***	7139	< .0001***
14	Composed Class and Encapsulation—methods	6083	< .0001***	5858	< .0001***	7342	< .0001***

* Significant at .05 probability level
** Significant at .01 probability level
*** Significant at .001 probability level

TABLE IX
CONFIDENCE RESULTS FOR THE TREATMENT GROUP USING WILCOXON SIGNED-RANK TEST, SHOWING THE T_{PR}–T_{PO}, T_{PO}–T_F AND THE T_{PR}–T_F COMPARISONS

Question no.	Topic	Pretest vs Posttest T_{Pr}–T_{Po}		Posttest vs Follow-up Questionnaire T_{Po}–T_F		Pretest vs Follow-up Questionnaire T_{Pr}–T_F	
		W	*p*-value	W	*p*-value	W	*p*-value
1	Methods and Invocation	5202	.0053**	4625	.1186	6952	< .0001***
2	Class and Object	5208	.0051**	4665	.1006	5230	< .0001***
3	Class and Object	4784	.0592	5138	.0081**	6901	< .0001***
4	Constructor and Instantiation	4204	.4278	5074	.0122*	6386	< .0001***
5	Class and Object	4246	.3890	4949	.0254*	6404	< .0001***
6	Class and Object	4764	.0651	5285	.0030**	6953	< .0001***
7	Methods and Invocation	4507	.1834	5695	.0001***	7041	< .0001***
8	Methods and Invocation	4199	.4330	5790	< .0001***	7029	< .0001***
9	Program Execution	5086	.0114*	5562	.0003***	7415	< .0001***
10	Inheritance and Polymorphism	5831	< .0001***	6430	< .0001***	7754	< .0001***
11	Inheritance and Polymorphism	6171	< .0001***	5674	.0001***	7601	< .0001***
12	Inheritance and Polymorphism	6548	< .0001***	5210	.0050**	7376	< .0001***
13	Composed Class and Encapsulation	5055	.0137*	5320	.0023**	7114	< .0001***
14	Composed Class and Encapsulation	5771	< .0001***	5532	.0004***	7401	< .0001***

* Significant at .05 probability level
** Significant at .01 probability level
*** Significant at .001 probability level

to understand; there was therefore more scope for improvement compared to the more basic OOP topics (such as classes and instances). Related to this, another possible explanation is that students were new to those concepts compared to the basic OOP topics. The VOOPA exercises were released to students only days after inheritance and polymorphism were taught in lectures, so most students would not have yet fully understood these concepts. Contrastingly, the basic OOP concepts were taught about 2–3 weeks earlier in the course. It is therefore likely that most students already had a good grasp of these basic concepts through course in general.

Naturally, not all elements of OOP (or misconceptions surrounding OOP) were targeted in the activities and the knowledge tests. However, due to many OOP concepts being inherently tied together, targeting some aspect(s) of OOP will often have carry-on effects to other OOP concepts. For example, the identity and attribute topic was not covered in the evaluation questions. However, concepts around identity and attribute were inevitably incorporated in VOOPA and the slides. It is therefore possible that improvements are seen in topics that were not discovered by the evaluation.

Overall, the evaluation seems to suggest positive learning gains in using a visual analogy tool for OOP concepts. The VOOPA tool was especially strong in demonstrating more

complex concepts such as those around inheritance and polymorphism. It is also effective at helping students overcome misconceptions, as many students reported their understanding of certain OOP concepts becaming clearer. In terms of the effectiveness, newer concepts received the biggest impact; slightly older topics (but still new in terms of learning OOP) received less benefits. This implies that the timing of such tools plays an important role in effectiveness. It should be used very soon after an abstract concept is introduced, and continue to be used as more topics are covered to allow the analogy to expand. Further research could be carried out to investigate the impact of timing on effectiveness.

B. Confidence Gain (Self-Efficacy)

RQ2: *How effective is a visual analogy tool in helping students gain confidence in visualizing OOP concepts?*

The analysis showed notable improvements on students' confidence, both in the Control and Treatment groups. For both groups, the results suggest that student confidence increased after the first learning method was provided, and continued to increase after the alternative method was also introduced. The overall results showed that student confidence was higher in every aspect after they used both methods. We can therefore conclude that both VOOPA and slides have their own merit in contributing towards student confidence, and using both together is most beneficial. This is not surprising as it was never intended for VOOPA to be a replacement for traditional teaching methods, but rather be a supplementary tool.

With regard to which aspects of OOP had more confidence boost after using VOOPA, it appears that once again the more complex topics had a higher impact. In addition to inheritance and polymorphism, a confidence boost was also apparent in visualizing program flow and how associated objects work. This improvement exists regardless whether VOOPA was presented before or after slides. It shows that VOOPA alone is a powerful tool to help students construct their own visualization of these complex and abstract concepts. When VOOPA is used after slides, areas of improvement extended to more basic concepts such as visualizing instance variables and instance methods. We speculate that this is because with the more basic concepts, stronger students would already have grasped them and they do not receive as much value from using VOOPA. With the weaker students however, directly exposing them with the analogies is not as helpful as explaining to them how the concepts work and then reinforcing it with the analogies.

C. Usability and Engagement

RQ3: *What are the usability considerations in developing a visual analogy tool?*

The quantitative evaluation on usability and engagement was simple, but results clearly showed that the slides had much better usability compared to VOOPA. One must concede that games are going to demand higher interaction complexity, particularly in comparison to slides (only requiring clicking "Next" to progress through them). This disparity is especially going to be prominent when the learner is not "a gamer" [57]. In terms of engagement, however, the results showed no statistical significance in differences between VOOPA and slides. This was somewhat surprising, as other studies frequently report visualization tools achieving good engagement [46], [50]. However, the reason behind this became clear after we conducted the qualitative analysis, which is discussed next.

There was a large number of responses regarding usability issues, mostly technical issues. Students reported that the control was difficult with the mouse not being locked. Unfortunately, this was an issue to do with a browser-based virtual world application that could not be fixed. Students also reported that the application ran too slow or lagged during the activities. This issue is mostly caused by hardware specification. As a first-person tool, VOOPA requires a good amount of graphics processing power which many students did not have on their personal computers. Some students also reported motion sickness, which is common with simulator applications [58]. However, this issue was dependent on individual differences and could not be resolved easily. The existence of such technical issues likely contributed to the relatively poor-perceived usability for VOOPA. The poor usability had a negative impact on student motivation and engagement, as a number of students complained to the teaching team that the application was tediously long due to game slowness. Reasons such as these help cater for the 31 students that accessed the activities but never completed them.

Despite the relatively poor usability, VOOPA still demonstrated distinct benefits towards students' understanding of OOP confidence and their confidence on visualizing them. It is possible that VOOPA will become even more powerful if these technical issues around usability could be fixed. Once these technical issues are fixed, we are confident that VOOPA will not only teach students OOP concepts effectively but also give students an engaging experience while learning. For future developments, we recommend developers to carefully consider how the control of the interaction is designed and the performance of their application. It is important to keep in mind that many students have never played a first-person game, and not every student has a high performance computer.

D. Threats to Validity

Although careful thought was put into the evaluation design, we must also consider the potential weaknesses that might compromize the validity of the conclusions we draw. The experiment was given to the students as a mini-assignment to be completed any time over five days, meaning external learning might happen during this time that would impact the results. Repeated testing using the same set of questions may also have led to students recalling their previous answer. Threats to validity may also come from the measurements taken. For example, the set of questions we designed may not fully demonstrate the "OOP concepts" as the students understand. There could also be noise and unreliable data as students might have guessed some responses or provide answers informed by other sources.

E. Lessons Learned

There are many lessons we could learn from in the process of designing VOOPA. First, we must recognize that the analogies and activities presented in Section III are the final product. An iterative software development was used, and essential, during both the design and implementation phases of VOOPA. The core ideas underlying the analogy evolved over two years of trialling it in earlier iterations of the CS2 course. The analogy was expanded and presented to students and other instructors in order to refine it. This was an important process to follow, and ensured the quality of the analogy design. In fact, the analogy details were finalized well before implementation of the application commenced. This proved valuable, as it helped guide the expected visualization. A key design decision for the analogy was that we always wanted it to be suitable regardless the OOP language of choice. In fact, since the analogy and its components were programming language agnostic, this allowed us to easily extend VOOPA to support Java alongside C++ [26].

Second, a learning tool will always have its limitations, so selecting the scope of the tool was crucial. This was more apparent with the use of analogy—as the concepts we tried to teach became more complex, so did the analogy. To avoid the analogy became overwhelmingly convoluted, we decided to leave out aspects of OOP like interfaces and abstract classes. We felt it would be more helpful to focus on a set of key learning outcomes that the analogy would be capable of teaching well, rather than trying to extend it to cover everything. We were aware that more complicated analogies not only lead to more difficulty in memorization, but are also more likely to result in disanalogous features—which is dangerous in analogy-based teaching [22], [40]. Although the evaluation for the inheritance and polymorphism elements was positive, mapping these as lifts and moving platforms may have already exceeded the definition of a "house" in most peoples' understanding, and required some (stretched) imagination. More work could be carried out to determine whether the current set of analogies are comprehensible and memorable, and thus successful in their mission.

Third, we found the use of a software framework or library can make the development process much easier. VOOPA was developed using Unity and presented as a browser-based web application thanks to Unity's support for WebGL. This made the application highly accessible to students and also greatly reduced development time as the web-based deployment simplified testing and trialling across the team. Finally, we found it was important to consider the usability aspect of the learning tool. We tried to cater for different types of learners with different levels of technical confidence when implementing VOOPA. We also carefully considered things like choice of color, fonts, and screen real-estate. For example, the code panel was hideable to reach a balance of visualizing the virtual world versus mapping to learning points in the code. However, as we have previously discussed, many students still faced usability issues which adversely impacted learning with the tool. It would be a pity if a tool with great content was not made use of by students because it has poor usability.

VI. Conclusions

The primary goal of this research was to help students understand OOP concepts through the use of visual analogies. After understanding student difficulties and misconceptions on OOP concepts, various activities and analogies were designed which aimed at helping students overcome those barriers. The analogies and their interactions were designed carefully by following guidelines provided by educational theories. A software tool, namely VOOPA, was developed to incorporate the activities and to allow students to visualize and interact with the analogies. The application was used by 253 students from a CS2 level course for evaluation. The results showed that students achieved both a learning gain and a confidence boost with statistical significance on various OOP concepts after using VOOPA. Student responses suggested that both the analogies and the activities used in VOOPA were well received. VOOPA offers an alternative form of learning support, achieving statistically significant learning gains in the study population pre to post-test, and thus is an excellent complement to existing (traditional) teaching methods.

In the future, many aspects of VOOPA could be extended. This includes creating more activities and analogies for other topics. Its usability could be improved, particularly its control, navigation and display. Evaluations could also be carried out on a larger scale and in more detail. With a more detailed evaluation, we could determine which parts of the analogy was useful and which parts require modification. Also, it would be helpful to evaluate the effectiveness of the analogy itself without the visualization. We could also evaluate how effective VOOPA is when used in conjunction with traditional teaching methods. Furthermore, we could track students' performance in terms of their programming ability in the long term.

References

[1] A. A. Al-Linjawi *et al.*, "Using Alice to teach novice programmers OOP concepts," *J. King Abdulaziz Univ.–Sci.*, vol. 148, no. 632, pp. 1–20, 2010, doi: 10.4197/Sci.22-1.4.

[2] M. Kölling, "The problem of teaching object-oriented programming, part 1: Languages," *J. Object-Oriented Program.*, vol. 11, no. 8, pp. 8–15, Apr. 1999.

[3] M. Teif *et al.*, "Partonomy and taxonomy in object-oriented thinking: Junior high school students' perceptions of object-oriented basic concepts," *ACM SIGCSE Bull.*, vol. 38, no. 4, pp. 55–60, Jun. 2006, doi: 10.1145/1189215.1189170.

[4] ACM/IEEE-CS Joint Task Force on Computing Curricula, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, New York, NY, USA, 2013, doi: 10.1145/2534860.

[5] D. Gupta, "What is a good first programming language?" *ACM Crossroads*, vol. 10, no. 4, pp. 7–7, Aug. 2004, doi: 10.1145/1027313.1027320.

[6] A. E. Fleury, "Programming in Java: student-constructed rules," in *ACM SIGCSE Bull.*, vol. 32, no. 1. ACM, Mar. 2000, pp. 197–201, doi: 10.1145/330908.331854.

[7] L. Grandell *et al.*, "Why complicate things? Introducing programming in high school using Python," in *Proc. 8th Australasian Conf. Computing Education (ACE'06)*, Hobart, Australia, Jan. 16–19 2006, pp. 71–80, doi: h10.5555/1151869.1151880.

[8] S. Holland *et al.*, "Avoiding object misconceptions," *ACM SIGCSE Bull.*, vol. 29, no. 1, pp. 131–134, Mar. 1997, doi: 10.1145/268084.268132.

[9] S. Cooper, "The design of Alice," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010, Art. no. 15, doi: 10.1145/1868358.1868362.

[10] W. Dann *et al.*, "Mediated transfer: Alice 3 to Java," in *Proc. 43rd ACM Tech. Symp. Computer Science Education (SIGCSE'12)*, vol. 12, Raleigh, NC, USA, Feb. 29–Mar. 3 2012, pp. 141–146, doi: 10.1145/2157136.2157180.

[11] F. I. Anfurrutia *et al.*, "Visual programming environments for object-oriented programming: Acceptance and effects student motivation," *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 12, no. 3, pp. 124–131, Aug. 2017, doi: 10.1109/RITA.2017.2735478.

[12] M. Kölling, "The Greenfoot programming environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010, Art. no. 14, doi: 10.1145/1868358.1868361.

[13] S. Xinogalos *et al.*, "Teaching OOP with BlueJ: A case study," in *6th IEEE Int. Conf. Advanced Learning Technologies (ICALT'06)*. Kerkrade, Netherlands: IEEE, Jul. 5–7 2006, pp. 944–946, doi: 10.1109/ICALT.2006.1652599.

[14] M. Kölling, "Using BlueJ to introduce programming," in *Reflections on the Teaching Programming*. Berlin, Germany: Springer, Berlin, Heidelberg, 2008, pp. 98–115, doi: 10.1007/978-3-540-77934-6.

[15] M. C. Carlisle, "Raptor: A visual programming environment for teaching object-oriented programming," *J. Comput. Sci. Colleges*, vol. 24, no. 4, pp. 275–281, Apr. 2009, doi: 10.1145/1047344.1047411.

[16] T. L. Naps *et al.*, "Exploring the role of visualization and engagement in computer science education," *ACM SIGCSE Bull.*, vol. 35, no. 2, pp. 131–152, Jun. 2002, doi: 10.1145/960568.782998.

[17] J. Biggs, "Enhancing teaching through constructive alignment," *Higher Educ.*, vol. 32, no. 3, pp. 347–364, Jun. 1996, doi: 10.1017/CBO9781139048224.009.

[18] J.-J. Dupin *et al.*, "Analogies and modeling analogies in teaching: Some examples in basic electricity," *Sci. Educ.*, vol. 73, no. 2, pp. 207–224, 1989, doi: 10.1002/sce.37307302073.

[19] S. M. Glynn, "Explaining science concepts: A teaching-with-analogies model," in *The Psychology of Learning Science*, 1991, pp. 219–240, doi: 10.4324/9780203052396-17.

[20] C. B. Hutchison *et al.*, "How to create and use analogies effectively in the teaching of science concepts," *Sci. Activities*, vol. 44, no. 2, pp. 69–72, Aug. 2007, doi: 10.3200/SATS.44.2.69-723.

[21] V. Diehl *et al.*, "Elaborated metaphors support viable inferences about difficult science concepts," *Educational Psychology*, vol. 30, no. 7, pp. 771–791, Dec. 2010, doi: 10.1080/01443410.2010.504996.

[22] M. Forišek *et al.*, "Metaphors and analogies for teaching algorithms," in *Proc. 43rd ACM Tech. Symp. Computer Science Education (SIGCSE'12)*, Raleigh, NC, USA, Feb. 29–Mar. 3 2012, pp. 15–20, doi: 10.1145/2157136.2157147.

[23] T. R. Colburn *et al.*, "Metaphor in computer science," *J. Applied Logic*, vol. 6, no. 4, pp. 526–533, Dec. 2008, doi: 10.1016/j.jal.2008.09.0057.

[24] P. Simons, "Instructing with analogies," *J. Educational Psychology*, vol. 76, no. 3, pp. 513–527, 1984, doi: 10.1037/0022-0663.76.3.513.

[25] D. N. Perkins *et al.*, "Transfer of learning," in *International Encyclopedia of Education*, Sep. 1992, pp. 6452–6457.

[26] V. Lian *et al.* (2021) Visualising object-oriented programming using analogies. [Online]. Available: https://digitaledu.ac.nz/voopa

[27] D. J. Armstrong, "The quarks of object-oriented development," *Commun. ACM*, vol. 49, no. 2, pp. 123–128, Feb. 2006, doi: 10.1145/1113034.1113040.

[28] N. Ragonis *et al.*, "A long-term investigation of the comprehension of OOP concepts by novices," *Comput. Sci. Educ.*, vol. 15, no. 3, pp. 203–221, Feb. 2005, doi: 10.1080/08993400500224310.

[29] K. Sanders *et al.*, "Checklists for grading object-oriented CS1 programs: Concepts and misconceptions," *ACM SIGCSE Bull.*, vol. 39, no. 3, pp. 166–170, Sep. 2007, doi: 10.1145/1269900.1268834.

[30] K. Sanders *et al.*, "Student understanding of object-oriented programming as expressed in concept maps," in *Proc. 39th SIGCSE Tech. Symp. Computer Science Education (SIGCSE'08)*, Portland, OR, USA, Mar. 2008, pp. 332–336, doi: 10.1145/1352135.1352251.

[31] B. Thomasson *et al.*, "Identifying novice difficulties in object oriented design," *ACM SIGCSE Bull.*, vol. 38, no. 3, pp. 28–32, Sep. 2006, doi: 10.1145/1140123.1140135.

[32] F. Marton *et al.*, *Learning and awareness*. New York, NY, USA: Routledge, 1997, doi: 10.4324/9780203053690.

[33] A. Eckerdal *et al.*, "Novice Java programmers' conceptions of object and class, and variation theory," *ACM SIGCSE Bull.*, vol. 37, no. 3, pp. 89–93, 2005, doi: 10.1145/1151954.1067473.

[34] K. Goldman *et al.*, "Identifying important and difficult concepts in introductory computing courses using a delphi process," *ACM SIGCSE Bull.*, vol. 40, no. 1, pp. 256–260, Mar. 2008, doi: 10.1145/1352135.1352226.

[35] S. Xinogalos, "Object-oriented design and programming: An investigation of novices' conceptions on objects and classes," *ACM Trans. Comput. Educ.*, vol. 15, no. 3, Sep. 2015, Art. no. 13, doi: 10.1145/2700519.

[36] R. Shmallo *et al.*, "Fuzzy OOP: Expanded and reduced term interpretations," in *Proc. 17th ACM Annu. Conf. Innovation and Technology Computer Science Education (ITICSE'12)*, Haifa, Israel, Jul. 3–5 2012, pp. 309–314, doi: 10.1145/2325296.2325368.

[37] I. Milne *et al.*, "Difficulties in learning and teaching programming—views of students and tutors," *Educ. & Inf. Technol.*, vol. 7, no. 1, pp. 55–66, Mar. 2002, doi: 10.1023/A:1015362608943.

[38] N. Liberman *et al.*, "Difficulties in learning inheritance and polymorphism," *ACM Trans. Comput. Educ.*, vol. 11, no. 1, Feb. 2011, Art. no. 4, doi: 10.1145/1921607.1921611.

[39] J. Sajaniemi *et al.*, "A study of the development of students' visualizations of program state during an elementary object-oriented programming course," *J. Educational Resources Comput.*, vol. 7, no. 4, pp. 1–31, Jan. 2008, doi: 10.1145/1316450.1316453.

[40] P. Thagard, "Analogy, explanation, and education," *J. Research Sci. Teaching*, vol. 29, no. 6, pp. 537–544, Aug. 1992, doi: 10.1002/tea.3660290603.

[41] K. Ardalan, "Using entertaining metaphors in the introduction of the case method in a case-based course," in *Exploring Learning & Teaching in Higher Education*. Berlin, Germany: Springer, Berlin, Heidelberg, Sep. 2014, pp. 69–96, doi: 10.1007/978-3-642-55352-3.

[42] J. L. Braasch *et al.*, "The role of prior knowledge in learning from analogies in science texts," *Discourse Processes*, vol. 47, no. 6, pp. 447–479, Sep. 2010, doi: 10.1080/016385309034209606.

[43] D. E. Brown *et al.*, "Overcoming misconceptions via analogical reasoning: Abstract transfer versus explanatory model construction," *Instructional Sci.*, vol. 18, no. 4, pp. 237–261, Dec. 1989, doi: 10.1007/BF00118013.

[44] D. Gentner, "Structure-mapping: A theoretical framework for analogy," *Cognitive Sci.*, vol. 7, no. 2, pp. 155–170, Apr. 1983, doi: 10.1016/S0364-0213(83)80009-3.

[45] M. Orgill *et al.*, "What research tells us about using analogies to teach chemistry," *Chemistry Educ. Research & Practice*, vol. 5, no. 1, pp. 15–32, 2004, doi: 10.1039/B3RP90028B6.

[46] J. Urquiza-Fuentes *et al.*, "A survey of successful evaluations of program visualization and algorithm animation systems," *ACM Trans. Comput. Educ.*, vol. 9, no. 2, Jun. 2009, Art. no. 9, doi: 10.1145/1538234.1538236.

[47] J. Sorva, "Visual program simulation in introductory programming education," Doctoral thesis, School of Science, Aalto Univ., 2012. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/3534

[48] C. D. Hundhausen *et al.*, "A meta-study of algorithm visualization effectiveness," *J. Visual Languages & Comput.*, vol. 13, no. 3, pp. 259–290, Jun. 2002, doi: 10.1006/jvlc.2002.0237.

[49] C. D. Hundhausen *et al.*, "What you see is what you code: A "live" algorithm development and visualization environment for novice learners," *J. Visual Languages & Comput.*, vol. 18, no. 1, pp. 22–47, Feb. 2007, doi: 10.1016/j.jvlc.2006.03.002.

[50] E. Fouh *et al.*, "The role of visualization in computer science education," *Comput. Schools*, vol. 29, no. 1-2, pp. 95–117, Apr. 2012, doi: 10.1080/07380569.2012.651422.

[51] J. Maloney *et al.*, "The Scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010, Art. no. 16, doi: 10.1145/1868358.1868363.

[52] Google. (2021) Blockly developers. [Online]. Available: https://developers.google.com/blockly

[53] P. Henriksen *et al.*, "Greenfoot: Combining object visualisation with interaction," in *Companion 19th Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, Vancouver, Canada, Oct. 24–28 2004, pp. 73–82, doi: 10.1145/1028664.10287019.

[54] A. Santos, "AGUIA/J: A tool for interactive experimentation of objects," in *Proc. 16th ACM Annu. Conf. Innovation and Technology Computer Science Education (ITICSE'11)*, Darmstadt, Germany, Jun. 27–29 2011, pp. 43–47, doi: 10.1145/1999747.1999762.

[55] J. Biggs, "What the student does: Teaching for enhanced learning," *Higher Educ. Research & Development*, vol. 18, no. 1, pp. 57–75, Nov. 1999, doi: 10.1080/0729436990180105.

[56] T. Tanielu *et al.*, "Combining analogies and virtual reality for active and visual object-oriented programming," in *Proc. ACM Conf. Global Computing Education (CompEd'19)*, Chengdu, China, May. 2019, pp. 92–98, doi: 10.1145/3300115.3309513.

[57] C. Heeter *et al.*, "Theories meet realities: Designing a learning game for girls," in *Proc. 2nd Conf. Designing for User eXperience (DUX'05)*, San Francisco, CA, USA, Nov. 3–5 2005.

[58] R. S. Kennedy *et al.*, "Research in visually induced motion sickness," *Applied Ergonomics*, vol. 41, no. 4, pp. 494–503, Jul. 2010, doi: 10.1016/j.apergo.2009.11.006.

APPENDIX
LIST OF KNOWLEDGE QUESTIONS

1. *A class is…*
   - *A vector/group of instances.*
   - *A vector/group of variables.*
   - ***A template for creating instances.***
   - *The same as an instance.*
2. *One of the purposes of a constructor is to initialize fields of a new instance:* ***True***
3. *When defining a constructor, an instance is created automatically:* ***False***
4. *By calling a method, you can change the value of the instance's fields:* ***True***
5. *By calling a method, you can add or remove fields of an instance:* ***False***
6. *Methods are executed according to their order in the class definition:* ***False***
7. *An instance method must always be called on an instance:* ***True***
8. *What is the meaning of the keyword* `this` *when you use it in a method?*
   - *It refers to the parameters of the method.*
   - *It refers to local variables that are defined in the method.*
   - ***It refers to the instance that the method was invoked on.***
   - *It refers to the method itself.*

For Questions 9 to 16, refer to the code below:

```
class Vehicle {
    protected:
        int fuel;
    public:
        Vehicle (int fuel);
        void fuelUp();
        virtual void move();
};
class Car : public Vehicle {
    private:
        int seats;
    public:
        Car(float fuel, int seats);
        virtual void move();
        void addPassenger();
};
// main.cpp
Car *car1 = new Car(30, 5);
Vehicle *car2 = new Car(20, 5);
```

9. *Which constructor(s) will be called when* `car1` *is created?*
   - *Only the Vehicle constructor.*
   - *Only the Car constructor.*
   - ***Both the Car and the Vehicle constructors.***
   - *No constructor will be called*
10. *Which constructor(s) will be called when* `car2` *is created?*
    - *Only the Vehicle constructor.*
    - *Only the Car constructor.*
    - ***Both the Car and the Vehicle constructors.***
    - *No constructor will be called.*
11. `car1->fuelUp()` *is a valid method call:* ***True***
12. `car2->fuelUp()` *is a valid method call:* ***True***
13. `car1->addPassenger()` *is a valid method call:* ***True***
14. `car2->addPassenger()` *is a valid method call:* ***False***
15. *Which method(s) will be executed when you call:*

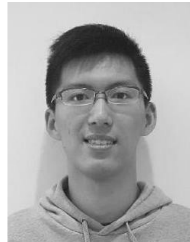
    ```
    car1->move();
    car2->move();
    ```

    - *The* `move()` *declared in the Vehicle class is executed in both cases.*
    - ***The*** **`move()`** ***declared in the Car class is executed in both cases.***
    - `car1->move()` *executes* `move()` *declared in the Car class, and* `car2->move()` *executes* `move()` *declared in the Vehicle class.*
    - `car1->move()` *executes move() declared in the Vehicle class, and* `car2->move()` *executes* `move()` *declared in the Car class.*
16. *Which field(s) can the* `move()` *method defined in the Car class access?*
    - *Only* `fuel`*.*
    - *Only* `seats`*.*
    - ***Both*** **`fuel`** ***and*** **`seats`*****.***
    - *Neither.*

For Questions 17 and 18, refer to the code below:

```
class Vehicle {
    private:
        Driver *driver;
    public:
        Vehicle(Driver *driver);
        void move();
        static void showSafetyHint();
};
class Driver {
    private:
        int id;
        void showLicense();
    public:
        Driver(int id);
        int licenseType;
        void printId();
};
```

17. *You are in the* `move()` *method, and you wish to call* `printID()`*:*
    - ***Call it by using*** **`driver->printID()`*****.***
    - *Simply use* `printID()` *— the method is transferred automatically.*
    - *You must define a* `printID()` *method in the Vehicle class first, then call it.*
    - *You cannot, there is no way to call a method defined in a different class.*
18. *The private* `id` *field declared in Driver can be accessed by:*
    - *The Driver's constructor only.*
    - *The Driver's constructor and the private method* `showLicense()` *only.*
    - ***All the methods and constructors declared in the Driver class.***
    - *The* `main()` *function, provided a Driver instance is created there first.*



Victor Lian received the B.Eng. (Hons) and M.Eng. degrees in software engineering from the University of Auckland, Auckland, New Zealand, in 2019 and 2020, respectively. He is currently working as a software engineer at MEA Mobile. His interests include exploring the application of software and machine learning in education.

Elliot John Varoy received the B.Eng. (Hons) and M.Eng. degrees in software engineering from the University of Auckland, Auckland, New Zealand, in 2016 and 2017, respectively. He is currently a doctoral student at the University of Auckland focusing on computing education. His interests include STEM and computational-thinking education.

Nasser Giacaman received the B.Eng. (Hons) and Ph.D. degrees from the University of Auckland, Auckland, New Zealand, in 2006 and 2011, respectively. He is currently a senior lecturer in the Department of Electrical, Computer, and Software Engineering at the University of Auckland. His disciplinary research includes parallel programming, with current research focusing on digital solutions across a number of different educational domains.