Interactive Learning of Abstract Programming Concepts with InteractiveOOP

Asheer Ahmad, Harsh Chokshi, Giuseppe De Ruvo, Nasser Giacaman
Parallel and Reconfigurable Computing lab
Department of Electrical and Computer Engineering
Auckland, New Zealand
{aahm849, hcho928}@aucklanduni.ac.nz, {g.deruvo, n.giacaman}@auckland.ac.nz

***Abstract*—Not only is understanding object-oriented programming fundamental for software engineering and computer science students, but it is also important for non-major programming courses. Combined with the burden of navigating a new programming language, students can struggle to understand the abstract concepts that underpin the paradigm. Existing learning solutions typically concentrate on a particular programming language. This paper proposes The Realization Framework for teaching abstract programming concepts without code. It employs analogies and visualization to translate the abstract concepts into forms students can understand. Theories on education and student interaction are incorporated to guide the learning process. INTERACTIVEOOP, a practical implementation of this framework, was evaluated by novice programmers in a non-major CS2 programming course. Students responded positively to the tool, noting that the interactive, visual presentation of abstract concepts helped them deepen their understanding.**

I. INTRODUCTION

The object-oriented programming (OOP) paradigm underpins many of today's most popular programming languages. In this paradigm, objects and classes are intended to correlate with real-life objects [1]. Its benefit stems from concepts dictating the relationship between code elements. Applying OOP is a fundamental skill of graduates in Computer Science related fields [2]. It is also typically a requirement in non-major programming courses offered to Electrical and Electronic, Mechatronics and Computer Systems Engineering students.

Due to their abstract nature, novice students can struggle to correlate OOP concepts to their real-world equivalents [3]. This prevents students from fully realizing the powerful features of OOP-based languages. Inheritance and polymorphism are among the most abstract and complex OOP concepts [4], [5]. Students often develop theoretical misconceptions, especially when left to create analogies and models to explain OOP on their own. It is acknowledged more research is needed into how to help students learn these abstract concepts [4].

A possible solution to this problem is to use interactive software applications. Incorporating these into the learning process increases student performance more than traditional, passive teaching methods [6]. However, misconceptions can persist even when such tools are used [7]. Presently, students often struggle to apply abstract OOP concepts outside of the interactive application they were learnt in [8], [9], [10].

The contemporary approach is for students to learn abstract concepts simultaneously with learning the syntax and semantics of a language. While this may appear efficient, it can adversely impact a student's grasp of both the concepts and programming language [8]. As there is no consensus on what language novices should learn first, existing tools focus on a particular language, thus limiting their audience.

In this paper, a framework is proposed for creating engaging applications for learning abstract programming concepts in a language-independent format. Analogy and visualization are used to represent abstractions, rather than raw code. The framework is grounded in educational theories such as constructive alignment and timely feedback.

Section II provides some background into educational theories, while Section III discusses related work. The Realization Framework is proposed in Section IV, with a concrete implementation of the framework (INTERACTIVEOOP) presented in Section V. An evaluation of INTERACTIVEOOP is provided in Section VI, before concluding in Section VII.

II. BACKGROUND

Constructive alignment recognizes that a student's actions contribute to how effectively they learn [11]. It provides a way for instructors to identify what learning activities should be chosen to improve student engagement [12]. Learning outcomes guide what to teach, how to teach it, and what kind of understanding the students should have about the topic. Activities and assessments need to be aligned with the learning outcomes. Chosen activities should reflect the way students are expected to use the concepts, rather than imparting knowledge without any cohesive relation to their pragmatic relevance [12].

Feedback is as much a part of the learning process as it is a way to reflect on learning. Using feedback has been shown to have a profound impact on educational achievement [13]. Hattie and Timperley [13] propose a model for feedback that centers on three questions: 1) "Where am I going?"; 2) "How am I going?"; and 3) "Where to next?". Feedback based on these questions help students understand where shortcomings in their knowledge are, and how to improve. In turn, this motivates students to persist in achieving learning outcomes. Timely feedback given closer to the task's completion is most beneficial on academic performance [13].

Visualization is commonly used by educators to represent general abstract concepts in a visual manner [14], [15], including abstract computer science principles [16]. While useful

in aiding student comprehension, studies have identified risks that come with them. The suitability of a visualization suffers when the connection between an abstract concept and its visual representation is poorly thought out [17]. However, no matter how strong the connection is, visualizations remain ineffective unless students actively engage with them [16]. The Unified Modeling Language (UML) is an international standard for visualizing software concepts. In particular, class diagrams represent objects and their relationships as boxes connected with various kinds of arrows. Class diagrams are well suited to showing inheritance, though their detail may be challenging to a novice programmer.

An *analogy* is the use of a familiar source concept as a metaphor to explain an unfamiliar target one. Analogies can suffer from the same issues as visualization; when there is a severe disconnect between the source and target, analogies could jeopardize student understanding [18]. Analogies are made effective when the source concepts used are something already familiar and understood by the students [18]. Misconceptions can be avoided by making it clear to students how the source and target are associated.

III. Related Work

There are several interactive environments and tools for teaching OOP concepts. BlueJ [19] extends the traditional OOP programming environment by visualizing class structures as a modifiable UML class diagram. Olsson et al. [14] also utilized UML, but instead they used it as a foundation for an animation showing the relationships between objects at execution time. Students struggled to comprehend the visualization, most likely due to the multiple focus points it demands. The Greenfoot Programming Environment [20] has a more palatable visualization and uses a class hierarchy, reminiscent of UML, but simplified to remove unnecessary features.

Greenfoot further differs from BlueJ by its application of analogies to OOP concepts. It displays a world in which users can place object instances and interact with them through method calls. ObjectKarel [8] visualizes objects in a similar way, but features guided lessons in OOP concepts. JavaTown [21] personifies objects as a town of people to help students understand abstract concepts in a context they are already familiar with.

An issue common to existing works is their reliance on specific programming languages. Students can sometimes find it difficult to learn programming concepts at the same time as picking up the syntax and semantics of a language [8]. ObjectKarel and JavaTown try to circumvent this by using simplified subsets of the programming languages. Albeit the availability of such tools, students can have difficulty with applying concepts outside of the environment they were learnt in [8], [9], [10]. This project focuses on learning concepts independently of any programming language.

IV. The Realization Framework

This paper proposes the Realization Framework to support the creation of interactive learning technologies that allow students to engage with and explore an abstract domain. In this context, realization refers to giving substance to abstract concepts. Theories on analogies, education, and visualization (as outlined in Section II) are brought together to support learning. The framework is presented as a process which instructors can use to develop their own interactive learning technologies. Though intended for aiding students to learn abstract programming concepts, the principles are general enough to be applied in other contexts. The Realization Framework is shown in Figure 1 and described as follows.

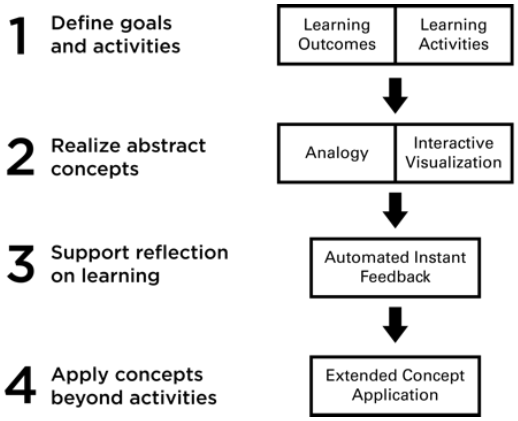

Fig. 1. The Realization Framework

A. Define Goals and Activities

The framework incorporates constructive alignment to drive focus toward the identified learning outcomes. Based on this, the first step is to identify the intended learning outcomes for the concepts to be taught. For best effect, these should be grounded in practical considerations that students will need to apply, which will form the foundation of the learning activities that will be developed for teaching. The activities should be developed through a process of identifying the practical uses of the learning outcomes, while considering aspects of the concept to be emphasized. Any prerequisite knowledge that students have should be considered when devising activities and determining the order to present them in.

B. Realize Abstract Concepts

An analogy that can be effectively visualized in the frame of the proposed activities should then be identified. The analogy should be relevant to students and ideally appeal to all target students, rather than a subset. The choice of analogy is crucial in relation to the planned activities; limitations in the analogy may make the activities more difficult than intended, or even cause misconceptions for the students if they apply the analogy where it breaks down.

The visualization of the analogy should be flexible, to support an extended range of interactions. The ability to interact by creating, modifying, and reasoning about the visualization better supports learning than simply viewing the visualization. The most important factor is that students need to interact with

the analogy and visualization through the learning activity. This is what provides the greatest benefit to learning [16]. Having real-world objects as source analogs can provide a natural basis for both the visualization and interaction method.

C. Support Reflection on Learning

Feedback is most useful when it is timely [13]. In a software based application, feedback can be given to users immediately. The feedback should describe what the student did well, what they could improve on, and relate back to the learning outcomes. This reinforces what the student needs to have learnt from completing the activity. Identify where in the activities success or failure states can occur. For each of these, develop feedback which can help the user understand the outcomes of their actions. Feedback should be designed to provide the student with a sense of progress, while being transparent and encouraging.

D. Apply Concepts Beyond Activities

Students can sometimes struggle to apply concepts outside of the context they are taught in [8], [9], [10]. To overcome this, interactive learning technologies should provide a way to bridge a student's understanding between what they did in the learning activity and how they can apply that knowledge further. This could be as simple as asking the student to answer questions about the concept which make them think about why, rather than how, they apply them.

V. INTERACTIVEOOP

INTERACTIVEOOP[1] is an application developed by applying the principles of the Realization Framework, to teach the OOP topics of *inheritance* and *information hiding*. INTERACTIVEOOP supports the learning process by allowing students to revise concepts independently of any specific programming language. It is intended to be used alongside other teaching methods as a way of providing additional support. As such, it assumes that students have some theoretical knowledge of OOP before they use the app. Students can apply this knowledge by performing the activities within the app.

A. Learning Outcomes

After an analysis of existing research, it was noted that abstract concepts like inheritance and information hiding were particularly difficult for students to comprehend [4], [5], [7]. Learning outcomes guide the way a topic is presented in INTERACTIVEOOP, and are used to derive appropriate content to support a student's learning. This is driven by alignment to the following Core-Tier1 learning outcome from the CS2013 Curriculum Guideline [2]:

> **PL/Object-Oriented Programming**
> *"Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses".*

This decision was motivated by the importance of inheritance as a core topic. It also enables for future work to explore further related concepts, such as polymorphism. For more effective constructive alignment of the application's activities with the learning outcome, the above CS2013 learning outcome was decomposed into more intricate learning outcomes:

1) Design and reason about a class.
2) Design and reason about simple class hierarchies.
3) Use class hierarchies to promote reuse.
4) Use the protected and private visibility modifiers to hide information between classes.

B. App Flow Overview

Individual topics form separate modules in INTERACTIVEOOP. Each module contains a Background section, multiple Class Structure Activities, and a Quiz. Once students select a topic from the menu, they are presented with its overarching learning outcome.

1) Contextual Background: Although INTERACTIVEOOP assumes students have some prior OOP knowledge, it still provides some background information. The content is not intended to be in-depth enough to teach new concepts to a student, but should serve as reminder to refresh their memory. The main aim is to ensure students are not struggling to use the app simply because of a disconnect between what the student expects and what the application expects. Background information first reiterates the learning outcome for a topic, then provides a motivating example, and finally describes the content of the topic. It draws a student's attention to terms used throughout INTERACTIVEOOP.

2) Class Structure Activities : Class Structure Activities are the main manifestation of the Realization Framework. Analogy and visualization are applied to help make abstract concepts palatable. Students solve a randomized problem by creating a class structure that meets a description. Each activity has its own sub-learning outcome that works toward the main learning outcome of the respective topic. Activities are organized such that later ones build on the concepts of earlier ones, either by adding a new idea or by extending the application of previous ones. Additionally, the activities provide pseudocode, such that students understand how the activities translate to code.

3) Quizzes: Each topic includes a quiz containing 10 multi-choice questions that provide additional assessment. Instead of being used to practice concepts, the quizzes test if a student has a grasp of the abstract programming concepts outside of the Class Structure Activities. They were included to be an intermediary step to connect the concepts used in the activities to their purpose in programming. Quizzes can be used to test a student's knowledge in ways that the activities cannot, particularly in thinking about the importance of concepts.

C. Activities Covered

A brief summary of the activities covered for each of the three topics is provided below.

Classes Topic:

- *Single:* create a single class based on a description.
- *Multiple:* create multiple classes based on a description.

[1]Available for Android, Linux, Mac and Windows from `http://parallel.auckland.ac.nz/education/interactiveoop`

These activities are aimed at familiarizing the user with INTERACTIVEOOP, and introducing basic class concepts.

Inheritance Topic:

- *Simple:* create a simple class hierarchy based on a description.
- *Reuse:* understand the relationship between inheritance and code reuse.
- *Advanced:* create an inheritance hierarchy based on a description.

Users are introduced to the principle of inheritance. The Reuse activity is slightly different from all others, as it has two parts. A problem is first solved without inheritance enabled, and then with it. It is designed to show how inheritance reduces the number of classes. Feedback for the activity includes highlighting the reduction in code duplication.

Information Hiding Topic:

- *Simple:* use the private visibility modifier to hide methods and fields.
- *Advanced:* use the private and protected visibility modifiers to hide methods and fields.

The Simple Information Hiding activity involves solving similar problems to the Multiple Classes activity, but with visibility modifiers incorporated. Likewise, the Advanced Information Hiding activity extends on the Advanced Inheritance activity. This final activity requires a user to put together all of the concepts covered in the application.

INTERACTIVEOOP was designed to support the addition of new activities and quizzes easily. In code, the Template Method software design pattern [22] is used for Class Structure Activities. The main functionality common to all activities is contained in an abstract class, including the overall processes of generating problems and providing feedback. By using this pattern, the process of adding new activities is streamlined. Data used to generate problems for the quizzes and activities is specified in human-readable files.

D. Selected Analogy and Visualization

INTERACTIVEOOP uses the analogy of people in a school. Other work, like JavaTown [23], saw that people was a successful analogy. Further, the idea of a school would be something very familiar to students. This format lends itself well to the OOP paradigm, as it is simple to define hierarchies of people who fit into increasingly specialized roles within a school. To be consistent with the idea of classes having a single responsibility, students and staff in the fictitious school are described as having a single talent. The analogy runs through all Class Structure Activities to build a unified model that users can reflect on. To make the analogy feel more practical, the user was given the role as the only student at the school whose talent was programming. In this context, they are a novice programmer tasked with learning OOP concepts to help the school build a program that simulates school life.

Acknowledging the risk of creating a visualization that could confuse students, a search was conducted for existing options to represent abstract programming concepts. UML is the ideal choice, given its status as an industry standard for visualizing classes and their relationships. However, this would require students to know how UML worked before they could make use of INTERACTIVEOOP. Thus, a decision was made to create a simplified, tree-like class hierarchy instead. In place of boxes, classes are represented with a person-like icon, to support the analogy. As illustrated in Figure 2, the visualization follows the same basic pattern as UML, while being simple enough for novice students to understand without additional training.

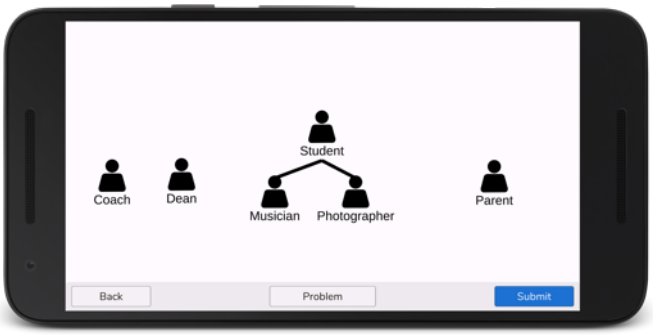

Fig. 2. Example of a Class Structure Activity canvas in INTERACTIVEOOP. The analogy and visualizations are unified, mimicking a simplified UML.

E. Completing an Activity

For each activity, users are presented with a *problem statement* constructed of four parts that describe:

1) The learning outcome,
2) The problem context,
3) A description of classes that students need to create a class structure for, and
4) A question statement that tells them what they need to do to solve the problem.

The third part is randomized, enabling students to repeat the same activity multiple times without feeling repetitive. It also prevents a user from simply memorizing solutions. A problem is generated by randomly selecting classes from a predefined problem file. A textual description of the selected classes is built from statements contained in the problem file. For each class, there is a statement of its purpose, followed by descriptions of each of its methods and fields. Users need to use this to work out what classes to make, and if inheritance can be used to minimize the number of classes they use. Though the contents of a problem is randomized, each possibility is isomorphic. The skills required to complete an activity, and its difficulty, are equal within a given activity.

To solve the activity, students are given a blank canvas to create their visualization on, which becomes populated as shown in Figure 2. They can tap anywhere in the canvas to create a class. Once they do, a window similar to Figure 3 pops up allowing the user to modify the class. Along with renaming the class, users can select methods and fields to include in it from a list of possibilities. Adding a child to the class is done via the Add Child button.

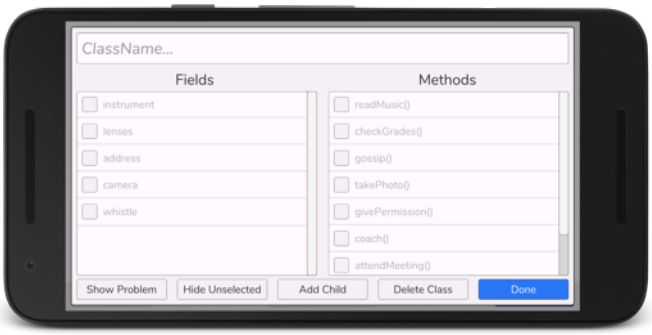

Fig. 3. A dialog for setting up a class. The user specifies which attributes should belong to the respective class they are defining.

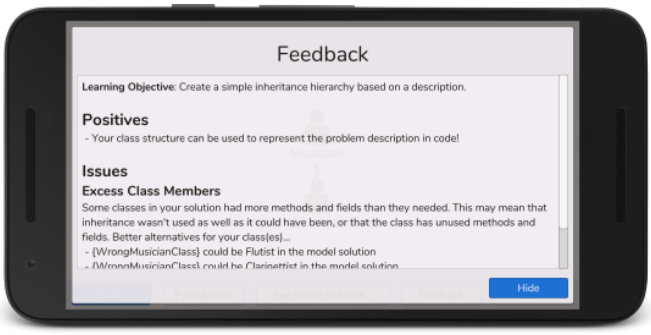

Fig. 4. Class structure activities give textual feedback answering Hattie's questions for effective feedback.

The available options for fields and methods are named based on keywords used to describe them in the problem statement. When designing the problems in INTERACTIVEOOP, there was a conscious effort to make these names as unambiguous as possible. None could belong to more than one of the possible classes needed for the solution. The visibility modifier of these can also be set. There are three possible visibility modifiers that can be used for a method or field - public, private, and protected. These were chosen because they are common to most OOP programming languages.

F. Feedback

A student's solution is compared to a model solution generated by INTERACTIVEOOP when it is creating the problem. Through this comparison, it is possible to advise students on how well they met the learning outcome. Feedback is provided immediately after a user submits a problem. INTERACTIVEOOP is very careful when providing feedback. Designing class structures is complex and open to subjectivity. It would not be appropriate for the app to tell a student that what they did was wrong when their solution could be equally valid. Another consideration is that feedback should not just tell the student if they are right or wrong; INTERACTIVEOOP focuses on the *process* of getting to a correct answer, and support students self-identifying where they are making mistakes. This gives them a better understanding of how they could improve for next time. Enforcing specific design choices would go against this. Taking these factors into account, a multi-part feedback system was created to provide high-quality feedback for a student to actively engage with.

1) Descriptive Feedback: Descriptive feedback guides a student on where they should look for problems. It was designed to answer the three questions in the model suggested by Hattie and Timperley [13].

- **Where am I going?** – The feedback begins by restating what the learning objective of the activity was, reminding the user of what they were aiming to achieve.
- **How am I going?** – First, the positives of the user's solution are listed. It is important to acknowledge where the user is correct, so they know they are making progress towards the learning outcome. After this, issues are detailed. Rather than telling a user they are wrong, the app describes how their solution differs from the model solution. After reading this description, students are given the opportunity to analyze the issues through visual feedback and/or pseudocode comparison.
- **Where to next?** – At the end of the descriptive feedback, a suggestion is made to the user that they should hone their skills by trying out another randomly generated problem. If the user made a mistake with the current problem, it also suggests that they use the Retry feature to go back and fix their solution.

With this information, students can use the next two feedback sources to fully grasp the nature of the problem.

2) Visual Feedback: The internal model solution is visualized and presented to the user in the same format of the solution that the user created (i.e. similar to Figure 2). At the press of a button, users can switch between the model solution and their own, which allows them to compare the differences between the two (if any). In this feedback mode, students are able to view (but not modify) the attributes of the classes by accessing the list in a similar manner to Figure 3.

3) Pseudocode: To assist students in transferring the visual interactions, INTERACTIVEOOP derives pseudocode for class structures. Figure 5 shows how users are able to compare the pseudocode of their solution side-by-side with that of the model solution. Only class, method, and field definitions are included with the pseudocode. This ensures the feedback is simple, with enough detail to address the learning outcomes. This also serves as a connection between what the student has done in the activity and how they are expected to apply that knowledge outside of INTERACTIVEOOP (for example how it would translate to code).

4) Post-Feedback Engagement: After reviewing feedback, the student is given the opportunity to retry the same activity. Their original solution remains intact, and they are free to modify it as they were before. This provides the student with the opportunity to apply the feedback they have just received. It serves to reinforce a student's new knowledge and allow them to try out the process of getting to a correct answer. Alternatively, students can exit the activity without retrying it.

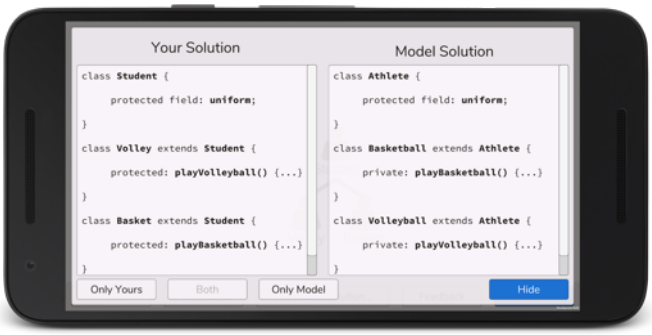

Fig. 5. A side-by-side view of pseudocode, with the user's solution on the left and the model solution on the right.

TABLE I
LOGGING DATA FOR COMPLETED ACTIVITY ATTEMPTS.

Topic	Activity	Completed Attempts	Completion Rate	Success Rate
Classes	Single	48	66%	79%
	Multiple	48	87%	48%
	Quiz	75	83%	60%
Inheritance	Simple	44	85%	61%
	Reuse	36	86%	75%
	Advanced	34	89%	56%
	Quiz	36	88%	61%
Information Hiding	Simple	17	81%	65%
	Advanced	27	93%	22%
	Quiz	32	94%	50%

VI. EVALUATION

INTERACTIVEOOP was offered to a CS2 course consisting of 223 students. The course is compulsory and includes Electrical and Electronic, Mechatronics, and Computer Systems Engineering students. For most of these students, this was their first exposure to OOP. Participation in the study was voluntary, with the tool being presented as an optional additional resource to complement lectures. INTERACTIVEOOP was installed 53 times, with 48 users completing at least one of the activities or quizzes. The evaluation consisted of an analysis of INTERACTIVEOOP's logging data and a qualitative questionnaire, both anonymized. The questionnaire focused on the user's enjoyment of the application, and how they felt it impacted their learning. The questionnaire was only available after users completed at least six different activities or quizzes, and had spent at least ten minutes using the application. This decision was made to ensure participants had used the app enough to provide meaningful responses. Although 24 users used the app enough to activate the questionnaire, only 19 of them submitted a response.

Threats to validity

The ethics approval granted for this project was restrictive, therefore a number of factors affect internal and external validity. As a result, a true experimental design study (e.g. a two-group control design) could not be performed. Not only was using INTERACTIVEOOP optional for students in the course, it was also anonymous. By respecting privacy, it was hoped to encourage a higher student uptake. Furthermore, if the same student downloaded the app on different platforms (e.g. on an Android mobile device and on a Windows PC), they would appear as two different users (logs are based on device ID). The evaluation likely exhibits experimental mortality, where there is a loss of participants as only more-motivated students persisted in using the app and completing the questionnaire. Multiple treatment interference is also possible, as students are likely to be accessing other learning resources simultaneously.

A. Logging Results

Logging data was used to identify user behavior when interacting with INTERACTIVEOOP. A total of 533 events were recorded. Logging mainly focused on which activities and quizzes a user completed, and how well they completed them. Table I shows results relating to the number of attempts which users completed. An attempt was considered complete if the user started an activity and submitted a solution. The Completion Rate shows the proportion of attempts that were completed from the total number of attempts. The Success Rate shows the number of Completed Attempts that had no errors. A Class Structure Activity was deemed correct if a user completed it without any errors. For a quiz, correctness was recorded if a user got at least 7/10 questions correct.

Table I shows there was a gradual decrease in the number of completed attempts as activities appeared later in the application. The Completion Rate, however, gradually increases for those activities with less attempts. This is likely attributable to more-motivated students progressing through the application. Something immediately noticeable is the low Completion Rate for the first activity (66%) compared to other activities. At first glance, the low Completion Rate seems to indicate that users were put off by not understanding how the activity works. However, of the 18 users who had incomplete attempts for this first activity, only one did not go on to complete another activity. This activity also possessed the highest Success Rate, so it unlikely represents students "giving up". It is therefore assumed that users may have been in the process of acclimatizing to the app, and this was the first activity they encountered.

Although many attempts were incorrect, only six users made use of the option to retry a Class Structure Activity. This was despite mentioning the feature in feedback, as well as highlighting the Retry button when the user submitted an incorrect solution. It seems that users did not wish to go back and change their solution after seeing the model answer. Instead, they opted to repeat the same activity with a different random problem. There tended to be a high level of attempts for the quizzes. Quizzes are shorter, and much more familiar to students. This observation confirms the importance of incorporating Constructive Alignment considerations (i.e. assessment following learning activities), and providing students with timely feedback.

The Single Classes activity was more heavily constrained with less leeway for mistakes to be made, so it is no surprise that it had a high success rate. However, performance in

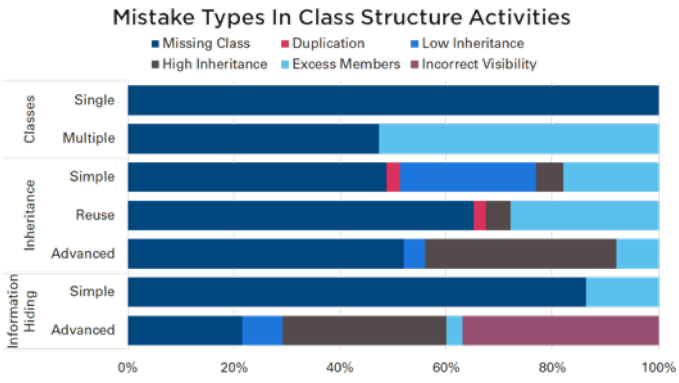

Fig. 6. Distribution of each mistake type in the Class Structure Activities.

other activities dropped notably, indicating students potentially faced difficulties completing them correctly. A breakdown of the kinds of mistakes that students made in Class Structure Activities is shown in Figure 6. There were six different categories of mistakes identified:

- **Missing Class**: the user's solution was missing at least one of the classes required by the problem description.
- **Duplication**: the user created multiple classes that could meet the same requirement.
- **Low Inheritance**: methods and fields grouped together into a single large class instead of using inheritance.
- **High Inheritance**: the user uses more levels of inheritance than is necessary to meet the description of the class hierarchy.
- **Excess Members**: a class contains a method or field that it does not need.
- **Incorrect Visibility**: the wrong visibility modifier was used for a class or activity.

Of the different categories, the one students had the most trouble with was *Missing Class*. For the Classes activities, this would come down to the student not being able to understand the problem statement. However, in later activities, the problem could arise due to a student misunderstanding how to use inheritance or information hiding constructs. Incorrectly applying these concepts could result in the user not generating the class they want to.

Looking at the other problem categories garners more interesting information. The *Excess Members* category shows users had difficulties with separating the responsibilities of classes. Instead, they would place methods and fields into the same class. When looking at inheritance, students were more likely to use too much inheritance, rather than not enough. Comparing the success rate in the Inheritance Reuse activities to the other Inheritance activities shows a difference. Users did much better in the Reuse activity, which required them to solve a problem without inheritance before solving it with inheritance. Potentially, the extra step made it clearer to the user how much sharing occurred between the classes, making it easier to create an inheritance hierarchy.

With regards to Information Hiding, users seemed to have no trouble with knowing when to use the private modifier. However, when inheritance and the protected modifier were added to the mix, students struggled. The Advanced Information Hiding activity showed the lowest success rate. The errors were not isolated to just a few users' mistakes – there were 15 different people that had trouble with this activity.

B. Questionnaire Results

Responses received from the questionnaire were largely positive. Table II summarizes the responses from the 19 respondents in the anonymous questionnaire. Overall, students tended to agree that INTERACTIVEOOP was an effective learning tool that helped in understanding OOP concepts, and how they relate to the real world (Q1-Q3). Despite the activities not requiring students to write code, which they appreciated (Q4), students felt that INTERACTIVEOOP still helps in applying the OOP concepts to writing code (Q5).

Interestingly, though 89% of respondents liked not needing to write code in order to solve a problem (Q4), 63% of them wanted the application to be more code-centric (Q6). In the open-ended section of the questionnaire, some suggested that the pseudocode could be included in activities – potentially in later exercises once users had become familiar with how the concepts worked outside of code. One student believed pseudocode was the most useful feature as it "*provides a link between the concepts in the app and code*". As a form of feedback, both pseudocode and the visualizations were responded to positively over the descriptive textual feedback; one respondent mentioned that descriptive feedback can be vague, though others did not mention this as an issue. However, based on the comments made by respondents, it seems that not all considered that they could use the pseudocode to model the problem in a programming language of their choice (although it was still seen as a valuable form of feedback).

In general, respondents felt that INTERACTIVEOOP was simple enough to use, with activities progressing in a logical manner and having strong alignment to learning outcomes (Q7-Q10). While students saw INTERACTIVEOOP as a valuable learning resource to complement lectures (Q11), and would like to use similar apps in other programming courses (Q12), it was not strongly viewed as sufficient on its own (Q13). Respondents also felt it could have been more intellectually stimulating (Q14), which was echoed in the open-ended section of the questionnaire with some students suggesting more activities and quizzes should be added.

When asked what was the most helpful aspect of INTERACTIVEOOP, eight students credited the interactive visualizations. This inclination arose from their enjoyment of the alternative presentation of abstract concepts which did not rely on code. Four students credited the quizzes rather than the Class Structure Activities, while other features mentioned were the pseudocode, the background information, and the use of an analogy to explain concepts. The most common criticism about INTERACTIVEOOP was centered around wanting more activities covering other abstract concepts, or wanting changes to the user interface of the application. This also included having

TABLE II
MULTI-CHOICE RESPONSES FROM 19 RESPONDENTS TO THE ANONYMOUS QUESTIONNAIRE.

	SD (1)	D (2)	N (3)	A (4)	SA (5)	$\bar{x}$	s
Perceived learning and transferring of OOP concepts							
Q1. InteractiveOOP provided me with a clearer understanding of OOP concepts.	0	0	2	11	6	4.2	0.63
Q2. InteractiveOOP helped me relate OOP concepts to the real world.	0	0	3	7	9	4.3	0.75
Q3. InteractiveOOP was an effective learning tool.	0	0	1	14	4	4.2	0.50
Q4. I found it useful to learn OOP concepts, without having to write code.	0	0	2	11	6	4.2	0.63
Q5. I feel confident I could apply the principles I learned through InteractiveOOP to write code.	0	0	4	9	6	4.1	0.74
Q6. I think I would learn better using InteractiveOOP if it was more code-centric.	0	3	4	9	3	3.6	0.96
INTERACTIVEOOP's relevance and supportive progression							
Q7. I found it easy to understand the purpose of each exercise.	0	1	3	9	6	4.1	0.85
Q8. The exercises in InteractiveOOP related well to their stated purpose.	0	0	0	11	8	4.4	0.51
Q9. The exercises in InteractiveOOP progressed in a logical order.	0	0	0	7	12	4.6	0.50
Q10. The activities in InteractiveOOP were well aligned with the learning outcomes for my course.	0	0	0	13	6	4.3	0.48
INTERACTIVEOOP's value as a complementary learning resource							
Q11. Using InteractiveOOP helped me study and apply what I learned in lectures.	0	1	2	11	5	4.1	0.78
Q12. I would use similar apps for learning other programming concepts in the future, if available.	0	0	2	5	11	4.6	0.77
Q13. InteractiveOOP provided enough support to practice OOP concepts without looking for more resources.	0	4	3	8	4	3.6	1.07
Q14. I found InteractiveOOP intellectually stimulating.	0	1	3	11	4	3.9	0.78

more questions for the quizzes. Overall, INTERACTIVEOOP appeared to have left a good impression on students.

VII. CONCLUSIONS

Novice programmers face difficulties when learning abstract concepts regarding object-oriented programming (OOP). Analogies and visualizations can be used to make abstract concepts more palatable to students. However, to ensure pedagogical value, these are best presented through engaging activities with clear learning objectives and supported by timely feedback. This paper presented the Realization Framework as a process for creating interactive visualizations to help students learn abstract concepts, grounded in appropriate educational theories. The framework was demonstrated in action through INTERACTIVEOOP – an application for revising OOP concepts without relying on a student's knowledge of any particular programming syntax.

INTERACTIVEOOP was evaluated by novice programmers in a CS2 course. As an optional additional learning resource, it was well received by those that engaged with it. The overwhelming majority of respondents agreed it was an effective learning tool that complemented lectures. The logs provided valuable insight as to the types of mistakes students encounter when dealing with inheritance and information hiding concepts. While evaluations showed self-reported learning potential, future work would include conducting a longer-term and controlled study integrating the Realization Framework, as well as including other OOP concepts like polymorphism.

REFERENCES

[1] Oracle Corporation, "What is an object?." https://docs.oracle.com/javase/tutorial/java/concepts/object.html, 2017.

[2] ACM and IEEE, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, 2013.

[3] L. Yan, "Teaching object-oriented programming with games," in *International Conference on Information Technology*, 2009.

[4] N. Liberman, C. Beeri, and Y. Ben-David Kolikant, "Difficulties in learning inheritance and polymorphism," *Transactions on Computing Education*, vol. 11, no. 1, 2011.

[5] N. Ragonis and M. Ben-Ari, "A long-term investigation of the comprehension of OOP concepts by novices," *Computer Science Education*, vol. 15, 2005.

[6] R. R. Hake, "Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses," *American Journal of Physics*, vol. 66, no. 1, 1998.

[7] S. Xinogalos, M. Sartatzemi, and V. Dagdilelis, "Studying students' difficulties in an OOP course based on BlueJ," in *International Conference on Computers and Advanced Technology in Education*, 2006.

[8] S. Xinogalos, M. Satratzemi, and V. Dagdilelis, "An introduction to object-oriented programming with a didactic microworld: objectKarel," *Computers & Education*, vol. 47, no. 2, 2006.

[9] S. Xinogalos, "A proposal for teaching object-oriented programming to undergraduate students," *International Journal of Teaching and Case Studies*, vol. 2, no. 1, 2009.

[10] B. Y. Alkazemi and G. M. Grami, "Utilizing BlueJ to teach polymorphism in an advanced object-oriented programming course," *Journal of Information Technology Education*, vol. 11, 2012.

[11] J. Biggs, "What the student does: Teaching for enhanced learning," *Higher education research & development*, vol. 18, no. 1, 1999.

[12] J. B. Biggs, *Teaching for quality learning at university: What the student does*. McGraw-Hill Education, 2011.

[13] J. Hattie and H. Timperley, "The power of feedback," *Review of Educational Research*, vol. 77, no. 1, 2007.

[14] M. Olsson, P. Mozelius, and J. Collin, "Visualisation and gamification of e-learning and programming education," *Electronic Journal of e-Learning*, vol. 13, no. 6, 2015.

[15] H. Rushmeier, J. Dykes, J. Dill, and P. Yoon, "Revisiting the need for formal education in visualization," *IEEE Computer Graphics and Applications*, vol. 27, no. 6, 2007.

[16] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. A. Velazquez-Iturbide, "Exploring the role of visualization and engagement in computer science education," in *ACM SIGCSE Bulletin*, vol. 35, 2002.

[17] G. Domik, "Do we need formal education in visualization?," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, 2000.

[18] P. Thagard, "Analogy, explanation, and education," *Journal of research in science teaching*, vol. 29, no. 6, 1992.

[19] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, 2003.

[20] M. Kölling, "The Greenfoot programming environment," *Transactions on Computing Education*, vol. 10, no. 4, 2010.

[21] I. Hadar and U. Leron, "How intuitive is object-oriented design?," *Communications of the ACM*, vol. 51, no. 5, 2008.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*, 1993.

[23] D. Feinberg, "A visual object-oriented programming environment," in *ACM SIGCSE Bulletin*, vol. 39, 2007.