Visualizing the Conceptual Framework of Object Orientation for Novice Programmers

Jakob Staugaard,* Jens Bennedsen,† Christoph Seidl,* Sebastian Nicolajsen,* Mathias Fink,* & Claus Brabrand*
*Center for Computing Education Research (CCER), IT University of Copenhagen, DK-2300 Copenhagen S, Denmark
†Department of Electronics and Computer Engineering, Aarhus University, DK-8200 Aarhus N, Denmark

***Abstract*—This research-*to*-practice paper shows how to visualize the conceptual framework of object-oriented programming. *'Classes'* and *'objects'* (from the solution domain) are respectively visualized as *'phenomena'* and *'concepts'* (from the problem domain), thereby visualizing the connection between code and reality. In the spirit of bringing research-*to*-practice, we implemented a prototype programming environment for JAVA called 'SHOWMYCODE' based on this conceptual visualization.**

We report on a controlled experiment involving N=138 introductory programming (CS1) students. The experiment involves three tasks with progression (USE, MODIFY, and CREATE) and compares a treatment group *with* the reality-code visualization vs a control group *without* the visualization. The results show that, for USE tasks, students are *faster* (but not more accurate) when using the reality-code visualisation. In contrast, for CREATE tasks, learners are *more accurate* (but not faster) when using the reality visualisation. Finally, we report that students appear to value a visualization that shows the reality-code connection above a generic visualization without this connection.

I. INTRODUCTION

Software has become a driving force for innovation, and industry demand for qualified software developers is at an all-time high[1]. For future software development experts, a fundamental understanding of programming is an essential foundation. Also, software and digitisation has been adopted in a wide range of other domains not traditionally founded in computer science and programming is becoming increasingly relevant for many disciplines; e.g., mechanical engineering, architecture, or law [1].

Object-oriented programming (OOP) incorporates the relationship between real-world concepts & phenomena and programming elements by fostering the use of *classes* (and *objects*) to group and encapsulate program data and functionality along abstractions from reality; e.g., *classes* (and *objects*) for mechanical devices, buildings, or contracts. In addition to a general understanding of data manipulation and control structures, acquiring a working knowledge in OOP constitutes learning concepts specific to the paradigm, such as creating, modifying, and using of classes, constructors, methods, and attributes. Learning an OO language, such as JAVA or C# may prove difficult as programming is intertwined with software design, which ties strongly to the represented real-world phenomena [2, 3, 4]

For novice programmers, the learning process may be aided by educational programming environments, such as BLUEJ [5], which depict program state and effects of its manipulation visually as abstract shapes resembling computer memory. However, especially practitioners from non-computing domains typically have less knowledge of the peculiarities involved in devising or executing computer programs; i.e., while they may be experts in their (reality) domain, they are likely to be novices when it comes to code. Consequently, focusing on mere abstractions of computer memory may not enable learners to realise their full potential. Few educational programming environments have begun to incorporate the relation between reality and code as elementary visualisations of program design and behaviour connected to real-world phenomena; e.g., GREENFOOT [6].

We hypothesize that further emphasizing the connection between real-world phenomena and programming concepts in an educational programming environment bears as-of-yet untapped potential. In particular, we suggest exploiting novice learners' pre-knowledge of their (reality) domain to bridge to an understanding of OOP code to improve the learning performance for programming novices. In this paper, we explore this hypothesis by using a conceptual framework to devise a prototypical learning environment that embodies a strong visual connection between reality and code. We then perform a controlled experiment on novice programmers striving to acquire fundamental knowledge of OOP.

II. BACKGROUND: CONCEPTUAL FRAMEWORK FOR OOP

The object-oriented paradigm dates back to the 1960s. It is unclear who coined the term *"object-oriented programming,"* but according to an email from Alan Kay, he did.[2] Another often mentioned source is the programming language SIMULA that Nygaard & Dahl created in the mid-sixties [7]. In 1962, Nygaard initiated the creation of a generic "simulation system;" the project subsequently welcomed Dahl and eventually led to the development of the SIMULA 1 programming language, which was later refined into SIMULA 67 (later renamed to simply: SIMULA):

> *It [*SIMULA 67*] is a generalization and refinement of the former [*SIMULA 1*], fairly ambitious, intended mainly as a general purpose programming language, but with simulation capabilities.* [7, p. 15]

[1] https://money.usnews.com/careers/\\best-jobs/rankings/the-100-best-jobs

[2] http://www.purl.org/stefan_ram/pub/doc_kay_oop_de

Coming from a simulation background, Dahl & Nygaard viewed programming as making a (computer) *model* of the real world. They found that the power of SIMULA was in the way it was possible to *abstract* a real-life (concrete) *phenomenon* into a *concept* and subsequently *express* (aka, *realise* or create) it using the notion of a *class* in the (object-oriented) programming language. The focus on perceiving system development as abstracting over real-life phenomena and *modelling* the real world in your programming language, rather than obsessing over operational functionality, is further described and argued for in [8] (Ch. 18, co-authored by Nygaard) who argue that:

> *The programming process involves identification of relevant concepts and phenomena in the referent system and representation of these concepts and phenomena in the model system. This process consists of three sub-processes: abstraction in the referent system; abstraction in the model system; and modeling.* [8, Ch. 18, p. 286]

Based on this understanding of the programming process, program execution is defined as:

> *A program execution is regarded as a physical model* simulating *the behavior of either a real or imaginary part of the world.* [8, Ch. 2, p. 16]

Note that *reality* is not limited to physically tangible phenomena; imaginary phenomena (e.g., an *idea* or a *risk*) might also be included. Designing a new, innovative system may require inventing new phenomena, not previously (physically) present.

The term *Referent System* is used for the part of reality under scrutiny and the term *Model System* for the realized computer model [8]. Figure 1 depicts the object-oriented development process. We adopt the more colloquial terms *Reality* (to the left) & *Code* (right) and use the notation 'R ⇌ C' abbreviating the *correspondence* between Reality and Code.[3] According to the process, we start with a real-world *phenomenon* (1); e.g., a specific real-life blue *car* with a semi-flat tyre (bottom left of Figure 1). This specific car phenomenon is subsequently *abstracted* (2) into the *concept* of a `car` (3). Then, the `car` concept is *modelled* (4) as a class: `Car` (5); e.g., incorporating relevant attributes such as *speed* and *color*, but omitting (abstracting away) irrelevant details such as the semi-flat tyre in the abstraction (2). The modelling step takes us from the *object-oriented analysis* (OOA) development phase focusing on analysing the *problem domain* (left-hand side) to the *object-oriented design* (OOD) development phase focusing on designing the computer model in the *solution domain* (right-hand side). At runtime, a `Car` class can then be *instantiated* (6) into a *car object* (7), which, ultimately, will be a computerized (simulated) model, *representing* the car phenomenon (8).

Our work is inherently based on this object-oriented development process and perspective; in particular, on the *connection* between *reality* and *code* (R ⇌ C); in other words, on the *relationship* between the *left-hand* side and *right-hand* side of Figure 1 (or, the *problem domain* and the *solution domain*).

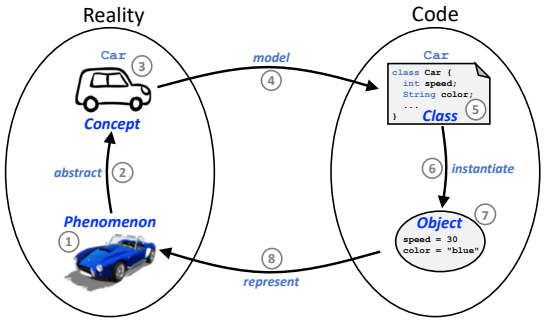

Fig. 1: Overview of the conceptual framework emphasizing the *connection* between Reality and Code: R ⇌ C.

III. Related Work: Systems for Learning OOP

Learning to program is, by many, reported to be a difficult task [9, 10, 11]. Many different approaches and tools to support learning to program have been proposed and evaluated. For an overview, see [12, 13].

In the following, we provide an overview of visualisation tools that support learning OO programming using visualizations similar to the one above. The overview of related systems is based on a paper from Sorva et al. [13], who have carried out an extensive review on program visualisation systems for introductory programming education, but we also include systems created after the publication of the paper from Sorva et al. [13].

Visualisation has been studied for many years, and a range of tools employing visualisation to aid learning processes was developed, especially within computer science, e.g., as visual programming languages, visualisation of algorithms, or visualisation of the notional machine. Prior research found that *virtual manipulatives* (based on visualisations of reality) are as effective as real physical ones in promoting student understanding of basic structural engineering concepts [14]. Also, research has found that despite having a larger cognitive load, realistic visualisations are beneficial in terms of retention performance [15]. In a meta-study of algorithm visualisation, [16] concluded that a crucial element is the possibility to interactively engage with a visualisation rather than merely passively perceiving it. Naps et al. developed a taxonomy of different forms of learner engagement with visualisation technology [17, p. 142]. They hypothesise that using visualisation actively (e.g., by discussing among students, called *presenting*), the beneficial effects of visualisation are most significant. For the specific endeavour of teaching fundamental OO programming to novices, a variety of systems that incorporate visualisation have been developed as a means of fostering understanding of the underlying mechanisms; e.g., GREENFOOT [6], BLUEJ [5], a metaphor-based animation tool (MBA) by [18], and block-based environments such as SCRATCH [19].

[3] For more experienced students learning to program (i.e., beyond novices), it would make sense to use the term *model* rather than *code* (as in R ⇌ M).

A. Criteria for Comparison

For our overview, we select systems supporting OO and JAVA that visualise the state of the objects when executing the code. Our focus is on visualising the connection between reality and code ($R \rightleftharpoons C$) and not systems that directly visualise the notional machine [20]; e.g., memory layout. We use two core aspects to compare related systems:

Type of visualisation distinguishes whether the system uses an $R \rightleftharpoons C$ visualisation or a memory-based visualisation. Systems that use reality-based visualisation utilise the depiction of real-world concepts or phenomena to visualise code; e.g., by visualising a `Car`-object using an image of a car. Systems that use a memory-based visualisation do not use the depiction of real-world concepts or phenomena; e.g., by visualising a `Car`-object as a coloured rectangle with the name of its respective class as a label.

Mode of instantiation captures whether the system has interactive object support and/or programmatic object instantiation. Systems with interactive object support let users instantiate classes and invoke methods upon the resulting objects in a reactive environment. Systems with programmatic object instantiation allow users to execute the main method of a program step-by-step. Some systems provide both interactive object support and programmatic object instantiation.

Most of the systems in [13] are not relevant to our overview as they do not provide the learner with the possibility to interact with the system or do not focus on OO.

B. Related Systems

GREENFOOT is a system with interactive object support *and* programmatic object instantiation that actively engages users in its visualisation (akin to $R \rightleftharpoons C$). In GREENFOOT, it is possible to create many so-called *scenarios* constituting actors that act in a micro-world; e.g., a car (actor) moving (acting) in a car park (micro-world). Here, the code and visualisation are *not* set up side-by-side, even though this could be configured manually. In this case, the code is *not* highlighted as it is executed. GREENFOOT supports JAVA as well as Stride, which is a JAVA-like programming language that combines the advantages of block-based and text-based programming languages [21]. [6] reports that extensive feedback on the system has been collected over several years. GREENFOOT is a successor to BLUEJ (presented shortly), both of which have been widely adopted by many educational institutions [22].

BLUEJ is, to the best of our knowledge, the de-facto system used in CS1 introductory programming courses. It is used by millions of users worldwide and can be used throughout an entire introductory programming course as it is able to visualise both small and medium-size programs due to its relatively generic program visualisation [5, 22]. BLUEJ visualises each object as generic red-coloured rectangle where the user can inspect the each field's name and value. Additionally, BLUEJ shows the static structure of the user's program with a UML class diagram and, similarly to GREENFOOT, does not show code and visualisation side-by-side.

MBA *Metaphor-Based Animation of OO Programs* is a system developed by Sajaniemi et al. that uses metaphor-based animation of OO programs as a reality-based visualisation [23]. It employs metaphors to visualise classes and objects; e.g., their predefined example class `BankAccount` is visualised as a blueprint from which `BankAccount`-objects can be instantiated. Application of MBA is limited to a few predefined examples with predefined visualisations [18]. As opposed to GREENFOOT, MBA does not have interactive object support, but allows the user to step through the execution of a main method line-by-line. However, it does show code and visualisation side-by-side, and highlights the code as it is executed to further support novice programmers. [23] have used MBA as one of several visualisation systems to evaluate student visualisation of programming state.

PLANANI: Before MBA, Sajaniemi & Kuittinen created a system called PLANANI that supported their concept of *roles of variables* [24]. They found that ten *roles* could cover 99% of variables in introductory programs [25]. As an example, they visualised a variable holding a constant value like a stone. They analysed whether role images and animation had an effect of resulting mental models concerning program and programming knowledge [26]. They concluded that *"what a student does plays a more central role in the usefulness of a visualisation than representation used by the tool"* (p. 395).

Research Gap: Considering this state of the art in visualization tools for novices learning OO programming, we conclude that there is a lack of an interactive tool that *clearly* connects reality and code; in particular, where the user (learner) can connect an easy-to-understand reality-based visualisation of objects and their state to the corresponding code. GREENFOOT employs a visualisation of objects, but does not show the code side-by-side with the visualisation. MBA has visualisation and code side-by-side and highlights the executed code, but does not have interactive object support that allows the user to actively engage with the visualisation.

IV. VISUALISING THE REALITY $\rightleftharpoons$ CODE CONNECTION

We hypothesize that better visualisation support for $R \rightleftharpoons C$ helps novice programmers in learning OO programming. To explore this, we developed a prototype environment called SHOWMYCODE, which connects reality and code by allowing users to instantiate (visualised) objects and invoke methods on them. We use this to test our hypothesis. In particular, we utilise the connection of how a *concept* from reality is *modelled* into a *class* in the code domain ($R \rightharpoonup C$, step (4) in Figure 1) and how an *object* in the code domain *represents* a *phenomenon* from reality ($R \leftharpoondown C$, step (8) in Figure 1).

SHOWMYCODE presents code and a reality-based visualisation side-by-side to strengthen the $R \rightleftharpoons C$ connection: Figure 2 shows a simplified version of SHOWMYCODE with the source code to the left (compile-time), and reality-based visualised objects to the right (runtime). In the object-visualisation area, novice programmers can instantiate new objects and invoke methods on existing visualised objects. Objects are visualised based on the phenomenon they represent. In the presented

case, the `Car`-object is depicted using a picture of an actual car. Furthermore, each field is visualised using a reality-like depiction. In Figure 2, the field `speed` is visualised as a well-known speed sign, and field `color` is visualised by the actual colour of the car depicted (in this case: blue). We use the term *scenario* when referring to an example in the prototype that is a combination of source code and object visualisation (e.g., a car-scenario).

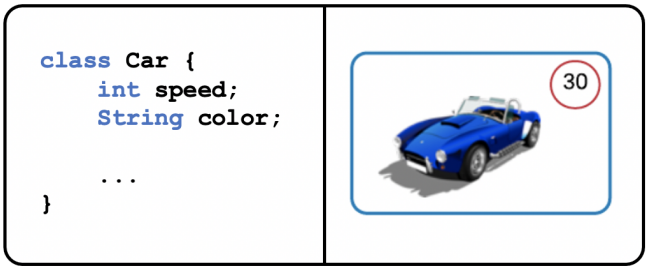

Fig. 2: Simplified version of the prototype

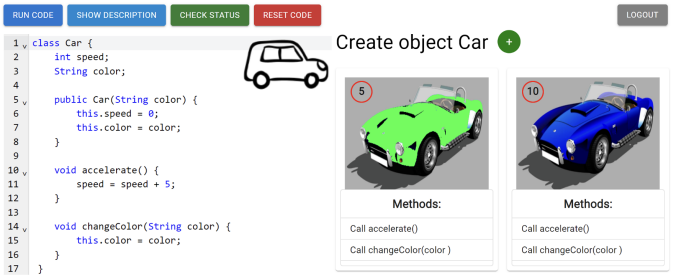

Fig. 3: Screenshot of the prototype in the `Car`-scenario.

Figure 3 shows a screenshot of SHOWMYCODE in a car-scenario with two `Car`-objects instantiated. The `Car` *class* (left) is modelled from the *concept* of a car and has two fields (`speed` and `color`), a constructor, and two methods (`accelerate()` and `changeColor()`). We consider the change of colour of a car to be easily relatable, even though it occurs infrequently compared to a car accelerating. To show that the class is modelled from the concept of a car, a stylized hand-drawing of the *concept* (of a car) is depicted next to the source code (in the upper right corner of the code window).

Two `Car`-*objects* are shown to the right (visualizing two car *phenomena*) with different states, affecting both speed and colour. Users can choose to instantiate a new object or to invoke defined methods on an existing object. Whenever users interact with an object, the corresponding code that is executed (left) is highlighted to emphasise the connection between reality and code. Users can instantiate an object by clicking the ⊕ button in the top right corner. Then, a popup window is shown with a prompt containing input fields corresponding to the actual parameters of the selected constructor while highlighting the code of the respective constructor (in this case: lines 6–9 in Figure 3). Additionally, users can invoke methods on an object by clicking on the corresponding buttons; e.g., "`Call accelerate()`". Similar to object instantiation, method invocation prompts users for the parameters of the method while highlighting the method in source code (see, for instance, the online demo[4]).

We developed SHOWMYCODE as a tool for our experiment to exclusively focus on two aspects: (1) the connection between reality and code; and (2) the difference between *class* vs *object* (corresponding to the difference between *concept* vs *phenomenon*). For the experiment, we have predefined selected scenarios where the source code (left) can be modified by users; e.g., by adding an additional method or modifying an existing one. (The particular visualisation images used for concepts and phenomena are, as of now, static and thus cannot be modified.) Users are asked to solve *tasks* where they can get instant feedback on the correctness of their solution as well as an appropriate error message if the source code does not compile. For practical use, we deem the functionality of SHOWMYCODE most relevant in the initial lectures of a CS1 introductory programming course at university level due to its specific focus on fundamental OO concepts.

V. EXPERIMENT

To evaluate the effect of visualising the connection between reality and code on teaching fundamentals of OOP to programming novices, we designed a controlled experiment in the context of an introductory programming course with approximately 150 first-semester university students enrolled. On the day of the experiment, N=138 students participated. We now describe the design and execution of the experiment.

A. Objective

The objective of our experiment is to investigate our hypothesis that strengthening the connection between reality and code (R ⇌ C) via visualisation is beneficial to the learning process of programming novices striving to gain fundamental OOP knowledge. We investigate this objective via the following three research questions:

- **RQ1 (Correctness):** To what extent do programming novice university students *more correctly* solve introductory object-oriented programming exercises with a visualisation of the connection between reality and code (in comparison to a control group without this visualisation)?
- **RQ2 (Speed):** To what extent are programming novice university students *faster* at solving introductory object-oriented programming exercises with a visualisation of the connection between reality and code (in comparison to a control group without this visualisation)?
- **RQ3 (Motivation):** To what extent do programming novice university students find the visualization of the connection between reality and code more motivating (in comparison to a control group without this visualisation)?

The visualisation environment is inherently designed to interactively provide feedback on whether a programming task has been solved correctly. For this reason, virtually all students

[4]https://sites.google.com/view/visualisingr-c/start

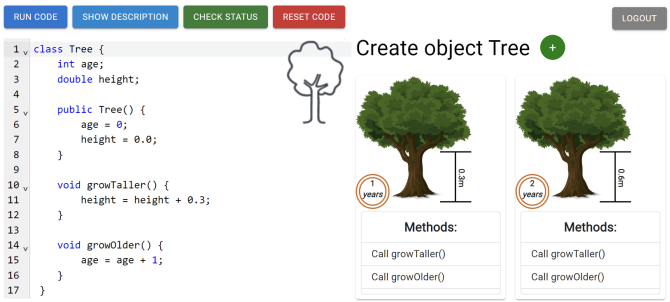

Fig. 4: Screenshot of the prototype in the `Tree`-scenario.

eventually will find a correct solution to a programming exercise (in our case, above 95% for all exercises). This obviously impacts how *correctness* and *speed* are most appropriately quantified.

Correctness is measured as the number of attempts necessary to solve an exercise. This appropriately measures the correctness relative to when students initially believe they have solved an exercise. *Speed* is measured as the time from an exercise is started until it is *eventually* correct. This appropriately measures the total time it takes for a student to engage with the learning tool and complete an exercise.

B. Context

We conduct the experiment in the context of the CS1 introductory programming course at the IT University of Copenhagen. The course is a 15 ECTS[5] mandatory first-semester (autumn) course on the three-year Bachelor of Software Development. The course introduces fundamental programming and object-orientation using a *conceptual framework* that emphasises a strong correspondence between reality and code via object-oriented modelling of reality [8]. The course uses JAVA as its programming language and, currently, BLUEJ [5] as its interactive learning environment.

C. Subjects

The experiment was run on September 6, 2022 where N=138 CS1 students participated in the experiment. Most students were in their early twenties. At the time of admission, the median age was 21 years. Some of the students had (limited) prior programming experience, while others had never programmed before. Students had only superficial experience with BLUEJ (during the first days of the course).

D. Tasks

Before the experiment, the teaching used a scenario involving *cars* (Figure 3); for the tasks in the experiment, we switched to a scenario involving *trees* (Figure 4), modelling the age and height of growing trees as to test students' ability to use the tool on a new concept.

In anticipation that our approach could provide benefits, differentially depending on the nature of a task, we deliberately incorporated a notion of *progression* into the tasks participants had to perform. We designed the tasks so that they would accompany the learning progression of students. To that end, we based the design of the tasks on the USE-MODIFY-CREATE (UMC) framework by [27], which organises tasks into a hierarchy of progressively more demanding competence. The idea of UMC is that, first, a student learns how to merely *use*; e.g., a method. The student learns to merely invoke (*use*) a pre-existing method to, for instance, get an object into a particular state. Later, a student progresses into learning how to *modify* the code of an existing method to make it do something slightly different. Finally, a student progresses into the ability to *create* an entirely new method. Table I provides a description of the three tasks (TASK 1–3) corresponding to each of the three levels of UMC.

TASK 1	**USE** the program to instantiate objects and invoke methods to achieve a program state where there is a tree with an *age* of `1` (years) and a *height* of `0.3` (meters), a tree with an *age* of `2` and a *height* of `0.6`, a tree with an *age* of `3` and a *height* of `0.6`, and finally a tree with an *age* of `4` and a *height* of `0.9` (meters).
TASK 2	**MODIFY** the code in two ways; first, so that the trees now instead have an initial *height* of `1` (meter); second, so that the trees now *double* in *height* every time they grow, once they are *older than* 2 years.
TASK 3	**CREATE** a new method `cut()` which should reduce (cut) the *height* of the tree by `0.5` (meters) whenever the trees are taller than 1 meter; otherwise they should be cut in half.

TABLE I: The *use*, *modify*, and *create* tasks of the experiment.

Task 1 requires little beyond understanding the concept of instantiating objects from classes, manipulating objects via method invocation, and inspecting object state. The task mostly involves clicking (on the user interface) to instantiate objects from classes and clicking to invoke methods. Task 2 incorporates select code modifications, requiring an understanding of how modifications impact dynamic objects at runtime, class vs objects, and compile-time vs runtime. Task 3 involves writing additional code in the form of creating a brand new method.

E. Treatments

Students were divided randomly into two groups: the *treatment group* versus the *control group*. The treatment group used SHOWMYCODE *with* the R⇌C visualisation of the connection between reality and code. In contrast, the control group used BLUEJ$^{\text{ISH}}$ which is a variant of SHOWMYCODE, but *without* visualising the connection between reality and code.

To control for unwanted non-visualisation related interference from differences in learning tools, we devised BLUEJ$^{\text{ISH}}$ as an adapted variant of SHOWMYCODE resembling BLUEJ. BLUEJ$^{\text{ISH}}$ is identical to SHOWMYCODE in all respects (using the same code) except for the visualisation. For visualisation, it adopted the conventions of BLUEJ for displaying objects and their state as well as for the creation and manipulation of the objects themselves (instantiation and invocation). Figure 5

[5]60 ECTS (European Credit Transfer and Accumulation System) = 1 year.

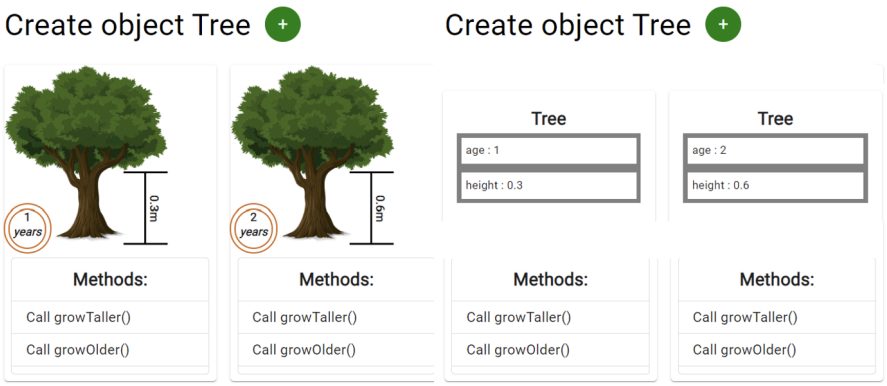

Fig. 5: Representation of two `Tree`-objects with reality-based visualisation (left) vs without this visualisation (right).

shows the two tool variants side by side using a scenario involving trees.

We decided to adopt BLUEJ's runtime memory-layout-based visualisation as the baseline in our experiment because it, is to the best of our knowledge, the most widely adopted learning environment for teaching introductory (CS1) object-oriented programming [22].

F. Design

The experiment was designed as a CS1 introductory lecture on object orientation wherein the students would be shown examples, and do small exercises, using either SHOWMYCODE or BLUEJ$^{\underline{\text{ISH}}}$. This also functioned as an introduction to the tool. Students were randomly assigned to the treatment or control group: 71 students were assigned randomly to the treatment group using SHOWMYCODE; 67 were assigned to students to the control group using BLUEJ$^{\underline{\text{ISH}}}$. Each group were then shown (in isolated rooms) a recording of the object orientation lecture wherein the only difference was the visualisation applied. Hereafter students worked individually on various car-scenario exercises.

The students were then tasked with solving the three tree-scenario tasks (presented in Table I) sequentially, with no time constraint. Students pressed a button "`Start`" to start a task, and "Check Solution" to test whether they completed the task (this button was always visible). Using the "Check Solution" button shows a popup informing the participant whether the solution was correct, returning the student to the task if the task was incomplete. If the solution was correct, students would be allowed to continue. The tool automatically logged information regarding *correctness* and *speed*. A follow-up survey investigated student *motivation*. The survey asked students whether they found the exercise motivating and/or frustrating (using a Likert scale) with the option to expand on their answer. The same questions were asked for the visualisation. In total, 122 of the participants expanded on their answers (88%). Both versions of the tool ran as a web-based platform. Due to the web-based nature of the platform, some events were not logged, leading to the loss of start/stop time for 16 tasks (in total). These (post-mortem identifiable) data points are therefore excluded from the study.

G. Ethical Considerations

The experiment was granted ahead-of-time ethical approval by our institution. We informed the students that we were not interested in the performance of individual students, but only the aggregated data comprising all students randomly allocated to a particular treatment; i.e., SHOWMYCODE vs BLUEJ$^{\underline{\text{ISH}}}$ visualization. Students not consenting to their anonymized experiment data being used, would have resulted in the experiment not recording any of their data. (In the end, all students consented to participate.) The differential exposure to one of the two versions of the tool was limited to a single lecture and exercise class early in the course. We therefore expect differential effects of the experiment to be minimal and confined to the early parts of introducing object orientation. After the experiment, all students were provided access to the recording of the lecture that used SHOWMYCODE; i.e., with the visualisation of the connection between reality and code. Later in the course, all students were exposed to substantial and, importantly, *identical* teaching and teaching/learning activities on object-oriented programming including the *conceptual framework* (see Section II) on which our visualizations were based. The remainder of the course used BLUEJ as development environment. Grading in the course was based on an oral exam which took place more than four months after the experiment had been conducted; by that time, we expect any differential treatment to be completely evened out.

H. Analysis

For comparing the *correctness* and *speed* of solutions from subjects assigned to the *treatment group* (using SHOWMYCODE) versus those assigned to the *control group* (using BLUEJ$^{\underline{\text{ISH}}}$), we use the U-test for comparing vectorized data and Z-test for comparing ratios (e.g., whether $x_1/y_1 > x_2/y_2$). For all statistical analyses, we use two-tailed tests and adopt a conventional 95% confidence interval (i.e., α = 5%).

VI. EVALUATION

We present first the results for *correctness* (RQ1); then those for *speed* (RQ2); hereafter those for *motivation* (RQ3). Finally, we summarise our findings.

A. Correctness (RQ1)

For each of the tools, we quantify *task correctness* as the number of incorrect submissions (aka, failures). An incorrect submission is when a student uses the "Check status" button, asking the tool for verification, but their solution is incorrect.

TASK	BLUEJ^{ISH}	SMC	P-VALUE	RESULT
USE:	0.24 fails	0.36 fails	p=0.562	inconclusive
MODIFY:	3.58 fails	4.06 fails	p=0.711	inconclusive
CREATE:	**1.53 fails**	**0.98 fails**	**p=0.0193**	**significance**

TABLE II: Average number of incorrect submitted solutions for USE-, MODIFY-, and CREATE-tasks under the treatments.

Table II provides an overview of the average number of incorrect submissions per treatment for the USE-tasks (top row), MODIFY-task (middle row), and CREATE-task (bottom row). For the CREATE task, we see an average of 1.53 failures (incorrect submitted solutions) using the BLUEJ^{ISH} tool. When adopting the SHOWMYCODE tool, this number drops to 0.98 failures. The effect is statistically significant with a p-value of 0.0193 according to a U-test. (For the USE and MODIFY tasks, the results are inconclusive with larger p-values ≥ 0.562.) We summarise our findings regarding *correctness* as follows:

> **OBSERVATION 1 (Correctness):** The student programming novices solve CREATE tasks *more correctly* using a reality-code visualization compared to a baseline without this visualization.

We speculate that novice programmers can take advantage of their understanding of the problem specification which, importantly, is phrased in terms of the *real world* (problem domain) in order to program the new method, `cut()`. CREATE tasks begin with the specification (problem domain) as opposed to MODIFY tasks that are initiated within the existing code (solution domain) wherein their domain knowledge presumably plays a more secondary role. Thus, we speculate this is why the visualisation does not impact the MODIFY task.

B. Speed (RQ2)

For each of the tools, we quantify the *speed* (of a task) as the amount of time elapsed from the beginning of the task (participant pressing "`Start`") until the task was solved correctly. As for the *speed* (for a tool), we simply *aggregate* the numbers to compute the *average* for the participants that eventually solved the task correctly. When looking at the aggregated time results (pertaining to a tool), we thus disregard incorrect attempts at solving a task.

TASK	BLUEJ^{ISH}	SMC	P-VALUE	RESULT
USE:	**242"**	**144"**	**p=0.00124**	**strong significance**
MODIFY:	425"	367"	p=0.242	inconclusive
CREATE:	252"	236"	p=0.070	inconclusive

TABLE III: Average time (in seconds) to eventual correctness for USE-, MODIFY-, and CREATE-tasks under the treatments.

Table III provides an overview of the average speed using the two tools for USE (top row), MODIFY (middle row), and CREATE (bottom row). For the USE task, we see that it takes an average of 242 seconds (4'02") to complete correctly using the BLUEJ^{ISH} tool; whereas this number drops to 144 seconds (2'24") when adopting the SHOWMYCODE tool. The effect is strongly statistically significant with a p-value of 0.00124 (via a U-test). For MODIFY and CREATE, students are faster when adopting SHOWMYCODE (from 7'05" to 6'07" for MODIFY and 4'12" to 3'56" for CREATE), but not significantly so. We summarise our findings regarding *speed* as follows:

> **OBSERVATION 2 (Speed):** The student programming novices are significantly *faster at solving* USE tasks using a reality-code visualization compared to a baseline without this visualization.

We hypothesize that the improvement is indeed due to programming inexperienced students taking advantage of their knowledge of the real-world in order to effectively manipulate all objects (via appropriate method invocation) into their intended states. We speculate that this is because it is easier for novices to get a quick overview of the *state* of an object on the left-hand side of Figure 5 (with the reality visualisation) than on the right-hand side of Figure 5 (without the reality visualisation) where attributes are represented by strings.

C. Motivation (RQ3)

In the quantitative data (based on a five-step Likert scale), there were no significant differences between students who used SHOWMYCODE vs BLUEJ^{ISH} on whether the students found the visualisation either *motivating* or *frustrating*.

In the qualitative data, among all the (optional) comments left, 9 (out of 62) made positive references to the BLUEJ^{ISH} visualisation, in comparison to 21 (out of 70) for SHOWMYCODE. The difference is statistically significant (p = 0.0340) according to a Z-test of the two ratios. One student commented on the way SHOWMYCODE *visualised* the object orientation concepts (which was precisely our intention with the tool):

> *"It was a good exercise which visualized the concepts behind OOP really well."*

Another student specifically mentioned the role of *reality* (which, after all, was one of the key ideas behind the tool):

> *"The exercises were good fun. I like that it is something that you can recognize from the real world."*

Yet another student emphasised the connection between visualisation and learning about *classes* and *objects*:

> *"It seemed cool with the option to concretely see what the various methods did to the trees through the visualization. I feel that this is good way for people who haven't programmed before—or with minimal experience—to understand how classes, objects, and methods work."*

For BLUEJ^{ISH}, 3 (out of 29) highlight in the comments that the visualisation aided them in *understanding* object orientation;

for SHOWMYCODE, this number increases to 8 (out of 30) for SHOWMYCODE. Although more than double, the difference is not statistically significant ($p = 0.107$) according to a Z-test.

In general, students using SHOWMYCODE highlight the connection between method calls and the particular object it affects, how it makes the more 'abstract' code pieces more 'concrete,' and the connection between objects and their attributes. We summarise our findings from student responses:

> **OBSERVATION 3 (Motivation):** Students appear to *value a visualisation* that shows the connection between reality and code more than a generic visualisation without this connection.

D. Summary

It is interesting to see orthogonal differences between the effects on USE vs CREATE tasks as a function of the two treatments (visualisation tools). When switching from BLUEJ^{ISH} to SHOWMYCODE: USE-tasks strongly significantly improve in terms of *speed*; whereas, CREATE-tasks significantly improve in terms of *correctness*. Students appear to value visualisation of the connection between reality and code (based on the conceptual framework of object orientation; see Section II). In retrospect, adopting the UMC framework and making distinct experiments with USE, MODIFY, and CREATE tasks permitted the identification of this important nuance.

VII. THREATS TO VALIDITY

A. Construct Validity

Participants knew what to do? We mitigated this threat by clear instructions; in particular, for the tasks, it was always possible to re-inspect the task description (textual specification of what to do). Also, there was a teaching assistant present in each experiment room, available for settling doubts about task problems (not solutions).

Time to eventual correctness as a proxy for speed? We believe time-until-eventual-correctness is a better metric for measuring the *speed* of solving a task, than time until first-attempt (as that may not be correct). To avoid polluting this data with incorrect answers, we filter out participants not solving the task correctly when looking at aggregated time.

B. Internal Validity

Tool differences beyond visualisation? The risk is that the two tool treatments (unintentionally) behave differently, aside from the (intended) differential visualisation. To mitigate this threat, we deliberately decided not to use BLUEJ but rather an imitation of the visualisation style of BLUEJ as a visualisation variant of our tool, named BLUEJ^{ISH}. Importantly, the two tools (BLUEJ^{ISH} and SHOWMYCODE) have a *shared code base*, except for the visualisations, which constitute *variation points*.

Bias in teaching materials? The risk is that the teaching would somehow, even unintentionally, favour one tool over the other, due to implicit biases in the teaching materials. We used ahead-of-time produced video lectures that were identical apart from the variation points involving the two tools to alleviate this threat. To further mitigate, a third-party *external* teacher unaffiliated with our university and not previously involved in this project was recruited to conduct the teaching. (Incidentally, the teacher ended up joining this project and paper, although, importantly, *after* the execution of the experiment.) Specifically, this teacher was chosen due to his expertise in the area of OOP.

C. External Validity

Beyond JAVA? Since none of the tasks nor visualisations depend on JAVA-specifics, we expect the results to generalize to any conventional *object-oriented* programming language with *classes*, *objects*, *fields*, *methods*, and *constructors*.

Beyond novices? Note that the tool was designed *specifically* for the first weeks of introductory (CS1) object-oriented programming. We expect diminishing returns later in courses as students get more exposure to object-oriented code and presumably start developing an intuition which essentially bridges the gap between reality and code.

VIII. CONCLUSIONS

The findings of our experiment of visualising the connection between reality and code resulted in three key observations:

1) For USE tasks, learners are *faster* (but nor more accurate) when using the reality-code visualisation;
2) For CREATE tasks, learners are *more accurate* (but not faster) when using the reality visualisation; *and*
3) Students appear to value a visualisation of the connection between reality and code.

Hence, usage of our research-*to*-practice programming environment prototype SHOWMYCODE visualising the conceptual framework for OOP by incorporating the connection between reality and code provided improvements to either speed or accuracy, depending on the nature of the task. These findings suggest that our hypothesis holds so that using visualisation to emphasise the connection between reality and code in an interactive educational development environment bears the potential to improve the learning performance of fundamental knowledge in OOP, at least, in the *early* (CS1) learning stages.

While SHOWMYCODE uses fixed example cases (trees and cars) that are hard-coded into the tool, we believe recent advances in generative AI image generation, allow the development of a more general version of SHOWMYCODE. Tools like MIDJOURNEY[6] could generate images for *concepts & phenomena* based on the source code entered by the student (programmer) which, in principle, ought to make the tool work with user-defined scenarios (modulo AI).

REFERENCES

[1] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Computer*, vol. 22, no. 2, pp. 63–70, 1989.

[6] https://www.midjourney.com/home.

[2] J. Bennedsen, M. E. Caspersen, and M. Kölling, *Reflections on the Teaching of Programming: Methods and Implementations*. Berlin, Germany: Springer, 2008, vol. 4821.

[3] S. Holland, R. Griffiths, and M. Woodman, "Avoiding object misconceptions," in *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 131–134.

[4] C. Holmboe, "A cognitive framework for knowledge in informatics: The case of object-orientation," in *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 17–20.

[5] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003.

[6] M. Kölling, "The greenfoot programming environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, Nov. 2010.

[7] O.-J. Dahl, "The birth of object orientation: the simula languages," in *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, O. Owe, S. Krogdahl, and T. Lyche, Eds. Berlin, Germany: Springer, 2004, pp. 15–25.

[8] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-oriented programming in the BETA programming language*. Reading, MA, USA: Addison-Wesley, 1993.

[9] T. Howles, "A study of attrition and the use of student learning communities in the computer science introductory programming sequence," *Computer Science Education*, vol. 19, no. 1, pp. 1–13, 2009.

[10] M. Guzdial, "Why is it so hard to learn to program?" in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds. Sebastopol, CA, USA: O'Reilly, 2010, ch. 7, pp. 111–124.

[11] M. W. Corney, D. M. Teague, and R. N. Thomas, "Engaging students in programming," in *Conferences in Research and Practice in Information Technology, Vol. 103. Tony Clear and John Hamer, Eds.*, vol. 103. Australian Computer Society, Inc., 2010, pp. 63–72.

[12] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, and C. Szabo, "Introductory programming: A systematic literature review," in *Proc. 23rd Annual ACM Conference on ITiCSE 2018*, 2018, p. 55–106.

[13] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education*, vol. 13, no. 4, Nov. 2013.

[14] E. Justo, A. Delgado, C. Llorente-Cejudo, R. Aguilar, and J. Cabero-Almenara, "The effectiveness of physical and virtual manipulatives on learning and motivation in structural engineering," *Journal of Engineering Education*, vol. 111, no. 4, pp. 813–851, 2022.

[15] A. Skulmowski and G. D. Rey, "The realism paradox: Realism can act as a form of signaling despite being associated with cognitive load," *Human Behavior and Emerging Technologies*, vol. 2, no. 3, pp. 251–258, 2020.

[16] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259 – 290, 2002.

[17] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. A. Velázquez-Iturbide, "Exploring the role of visualization and engagement in computer science education," in *Working Group Reports from ITiCSE 2002*, 2002, p. 131–152.

[18] J. Sajaniemi, P. Byckling, and P. Gerdt, "Metaphor-based animation of oo programs," in *Proc. ACM Symposium on Software Visualization (SoftVis 2006)*, 01 2006, pp. 173–174.

[19] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, Nov. 2010.

[20] B. Du Boulay, "Some difficulties of learning to program," *Educational Computing Research*, vol. 2, no. 1, pp. 57–73, 1986.

[21] M. Homer and J. Noble, "Lessons in combining block-based and textual programming," *Journal of Visual Languages and Sentient Systems*, vol. 3, no. 1, pp. 22–39, 2017.

[22] BlueJ, "BlueJ and Greenfoot Statistics," Available from https://bluej.org/newstats/, 2021.

[23] J. Sajaniemi, M. Kuittinen, and T. Tikansalo, "A study of the development of students' visualizations of program state during an elementary object-oriented programming course," *Journal of Educational Computing Research*, vol. 7, no. 4, Jan. 2008.

[24] J. Sajaniemi and M. Kuittinen, "Program animation based on the roles of variables," in *Proc. ACM Symposium on Software Visualization (SoftVis 2003)*, 2003, p. 7–ff.

[25] P. Byckling and J. Sajaniemi, "Roles of variables and programming skills improvement," in *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 413–417.

[26] S. Nevalainen and J. Sajaniemi, "An experiment on the short-term effects of engagement and representation in program animation," *Journal of Educational Computing Research*, vol. 39, no. 4, pp. 395–430, 2008.

[27] I. Lee, F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith, and L. Werner, "Computational thinking for youth in practice," *ACM Inroads*, vol. 2, no. 1, p. 32–37, Feb. 2011.