Making the abstract concrete in object-oriented programming through use of a serious game with narrative and visualization

Rolando Gonzalez[1], Bogdan Marculescu[1], Marcus A. Dahl[1], and Omid Mirmotahari[2]

[1] Kristiania University of Applied Sciences
[2] University of Oslo (UiO)

Abstract. Object-oriented programming (OOP) is central to computing curricula, but students have difficulty grasping its more abstract aspects. A key reason is that the runtime behavior of an OOP program is hidden: in a conventional development environment, students see code and console output, but not the objects, references, and state changes they are asked to reason about. This study examines a proof-of-concept serious game in which first-year students write Java code within a narrative, and in which the resulting objects, their properties, and the relationships between them are made directly observable. Rather than measuring learning gains, we ask how this observability shapes the way students reason about abstract OOP concepts. The prototype was evaluated with 23 participants – four individual in-depth interviews in June 2023 and three focus groups with 19 participants in May 2024 – and the material was analyzed thematically. Participants used the visual representation to connect the execution of a line of code with the creation of an object and with subsequent changes to its state, and the narrative supplied a domain vocabulary through which they discussed and reasoned about those operations. The findings suggest that making object creation, state, and relationships observable can reveal discrepancies in students' mental models that remain hidden in conventional programming environments.

Keywords: Object-oriented Programming · Serious Game · Notional Machines · Visualization · Higher Education

1 Introduction

Object-oriented programming (OOP) is a central paradigm in computing curricula, but it is also widely recognized as difficult for students to learn. When writing OOP code, students model real or imaginary phenomena as classes and objects, and much of the conceptual work involves grasping entities that are inherently abstract: the distinction between a class and its instances, object references versus property values, constructors, and the passing of arguments between methods. A systematic review ranks these notions among the most troublesome

in introductory OOP [2], and Sorva [8, pp. 8:18–8:20] identifies the dynamic behavior of programs at run time as troublesome knowledge in the sense of Meyer and Land [5, p. 373]: transformative, but hard-won.

A key source of this difficulty is that the runtime behavior of an OOP program is largely invisible. In a text-based integrated development environment such as IntelliJ, students see only code and, at best, console output; the objects, references, and state changes they are reasoning about have no direct visual representation. To program effectively, novices must therefore construct and run a mental model of an unseen abstract machine – what Sorva [8, pp. 8:2–8:3] terms a *notional machine*. A long line of research has explored visualization as a way to make these hidden dynamics explicit, ranging from program visualization systems [8] to visual analogies for OOP concepts [3] and conceptual frameworks connecting real-world phenomena to their representation in code [9].

Gamification and serious games have attracted growing interest in programming education, with recent reviews reporting positive effects on motivation and engagement [4, 6] and meta-analytic evidence of a moderate positive effect on academic performance under some conditions [10]. Much of this literature, however, emphasizes motivational and behavioral outcomes, and comparatively little attention has been paid to the specific design mechanisms through which a game environment can change what a student is able to see of a running program, or to what that added visibility reveals about the models students hold.

This paper addresses that gap. We developed a proof-of-concept serious game in which first-year students read and write Java code within a narrative spanning ten missions, and in which the objects they create, the values of their properties, and the relationships between them are displayed and updated in real-time. We then studied students' reflections and experiences with the game through individual in-depth interviews and focus groups, analyzed thematically [1]. Rather than measuring learning gains, our aim is to understand, from the students' perspective, what happens when the normally hidden chain from code, through runtime action, to object, state, and relationship is made observable.

The study is framed around one overarching research question with two subquestions, which separate the two design mechanisms the game combines.

RQ. How does a serious game that makes the runtime state of an OOP program observable within a narrative shape the way first-year students reason about abstract OOP concepts?
- **sub-RQ1.** How does making object creation, state, and relationships observable shape students' reasoning about abstract OOP concepts, and what mismatches in their understanding does it surface?
- **sub-RQ2.** How does the narrative context shape the way students interpret, discuss, and reason about OOP operations?

A third mechanism emerged from the material without having been anticipated in the research questions: because the game supplies an object model the players cannot modify, progress depends on reading and understanding existing code rather than on writing code from scratch. We report this as an emergent finding in Section 4.4 and return to it in the discussion.

Section 2 presents the theoretical framework and Section 3 the game, the data collection, and the analysis. Section 4 reports the findings, which are discussed in Section 5. Section 6 sets out the limitations of the study and Section 7 concludes.

2 Theoretical framework

2.1 Difficulties in learning OOP

Decades of computing education research document the difficulties novices face with OOP. Gutiérrez et al. [2] provide a systematic ranking of problems and solutions in teaching and learning OOP, covering the understanding of objects and their dynamic nature, classes and methods, and the development of abstract thinking more generally. These difficulties are not only about syntax: many concern the conceptual model students hold of what a program is and does, a recurring instance being the conflation of classes with objects, and of objects with the references that point to them [3].

Some of these notions have been characterized as *threshold concepts*: transformative, irreversible, and integrative ideas that learners experience as troublesome, and before which they may remain for an extended period in a state of partial understanding [5, pp. 373–377]. Sorva [8, pp. 8:18–8:20] argues that the runtime dynamics of programs constitute precisely this kind of troublesome knowledge, and that novices frequently develop non-viable mental models of it. We use this literature to motivate our focus on runtime dynamics; the present study does not set out to establish whether any particular threshold was crossed.

2.2 Mental models, notional machines and observability

To reason about a running program, a learner must operate a notional machine: an idealized, abstract model of how the language executes code [8, pp. 8:2–8:3]. Synthesizing research on programming misconceptions, mental models, constructivist learning, and threshold concepts, Sorva [8] argues that the ability to program is closely tied to the ability to reason about what happens in memory as code runs, and that the effectiveness of program visualization depends less on the technique itself than on how actively learners engage with it [8, pp. 8:22–8:23]. A notional machine is not itself a visualization; it is the abstract machine that a visualization or verbal explanation may be designed to make explicit.

This distinction motivates the term we use throughout the paper. What a learning environment offers is not simply pictures, but *observability*: the extent to which the chain from a line of code, through the runtime action it triggers, to the object created, its state, and its relationships to other objects, is available to inspection. A conventional IDE makes almost none of this chain observable without deliberate instrumentation by the student, such as print statements or breakpoints; an environment that externalizes selected parts of the notional machine makes the same chain available by default. Visual analogies have been used to help students map unfamiliar OOP concepts onto familiar ideas [3], and

conceptual frameworks have been proposed to make explicit the path from real-world phenomena, to concepts, to classes, to objects [9]. Our game builds on this tradition by rendering objects, their properties, and the effects of method calls directly, so that students can see the consequences of the code they write rather than infer them from text alone.

2.3 Serious games in programming education

It is useful to distinguish *gamification*, the use of game design elements such as points, progression, and feedback in non-game contexts, from *serious games*, which are games designed for purposes beyond pure entertainment [10, pp. 2480–2481]. Our application is best characterized as a serious game, since it embeds learning within a self-contained narrative game world, although it also employs gamification elements such as missions, progression, and immediate feedback.

Recent syntheses report that gamified approaches in programming education can enhance motivation, engagement, and cognition [4], with comparable findings in other disciplines [6]. Meta-analytic evidence is consistent but qualified: across 22 experimental studies, Zeng et al. [10, pp. 2490–2496] report a moderate positive pooled effect on academic performance ($g = 0.78$), with substantial heterogeneity and with outcomes depending on instructional design, subject area, and how well the game elements are matched to the targeted learning. We position our work within this literature not as a test of motivational effects, but as an inquiry into the design choices: observability and narrative, through which a serious game can make the abstract concepts of OOP more concrete. That the artifact is a game, is secondary to what its interface makes visible.

3 Method

3.1 The serious game

The game is a web application developed in the JavaScript framework Vue by three of the authors. It consists of 10 chained missions taking players through the basics of object instantiation and method calling in Java. The in-game narrative sets the player the goal of populating a fantasy village: giving the new inhabitants different roles, building houses for them to live in, and constructing a forge in which to make armor for a knight who defends the village.

The player interacts with the game world by writing and executing lines of Java code to accomplish objectives: instantiating new objects from existing classes, assigning new values to object properties, running methods associated with those instances, and creating connections between existing objects. Requirements are expressed in domain terms – for example, that a woodcutter is required to collect wood – but must be accomplished using programming constructs, here by setting the *role* property of a villager instance to the value that satisfies the condition guarding the relevant method. Missions are chained, so instances created in one mission remain available in the next.

Separate panels give the player the code input, available classes for the mission, created objects, values of the properties of a selected object, and the mission description and progress (Fig. 1). When a line of code is executed, the interface updates reflecting the new state: instantiating a villager makes a new villager appear on screen, its properties set either by default parameters or according to the constructor invoked, and subsequent changes appear immediately.

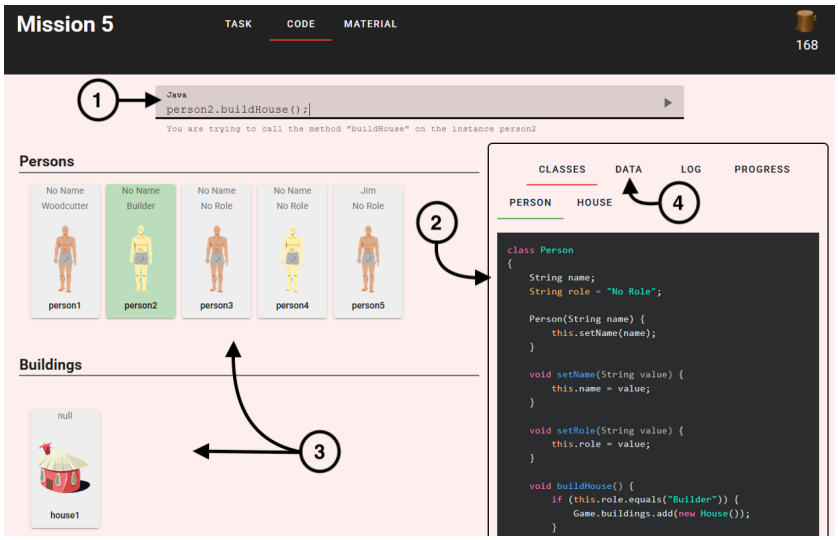


Fig. 1. The game interface during a mission, showing (1) the code panel, where the player writes and executes a line of Java code, (2) the classes panel, giving access to classes for the mission, (3) the visual output of the person and house objects created by the player, and (4) the data panel, showing property values of object instance.

3.2 Data collection

Participants were recruited from the course Object-Oriented Programming at Kristiania University of Applied Sciences, a compulsory course in the second semester of the Bachelor in IT programme. Data was collected in June 2023 and May 2024 after the course had been completed. This is a deliberate design feature as our interest is in what the game reveals about the models students hold, not whether it can teach OOP from scratch. Recruitment was by open invitation. A message was sent to all students registered in the course in the learning management system Canvas – 586 students in 2023 and 744 in 2024. All who volunteered were included. Age and gender were not recorded.

The June 2023 instance produced four individual in-depth interviews, each participant having 30–45 minutes with the game followed by an interview of 30–45 minutes. The May 2024 instance involved 19 participants across three focus

groups. Participants worked with the game in teams of 3–4 for 30–45 minutes and were encouraged to discuss and reason together; each session was followed by a group discussion of approximately 45 minutes. The format was chosen because discussion between students would lead them to express their understanding of OOP more spontaneously than individual interviews would. The total across both instances is 23 participants. The respondents varied in proficiency, some completing all missions in under 25 minutes, other did not finish.

Semi-structured interview guides were used in both instances, and all sessions were conducted in English by the first and second authors, neither of whom taught the participants in the course. They were present throughout to observe, answer questions, and provide support where needed. Questioning was directed at participants' experience of the game and their reflections on how it related to their understanding of programming, rather than at their performance on the missions. Data were recorded as written notes and care was taken to record the participants' words as closely as possible. The citations in the Findings chapter are based on these notes and record the participants' answers verbatim. In Section 6 we consider the consequences of not recording video or audio.

3.3 Thematic analysis

The material was analyzed using thematic analysis following the six-phase approach of [1]. The analysis was inductive, data-driven, semantic, and conducted manually by the first and second authors, who also conducted the interviews, working both individually and together.

Both analysts first familiarized themselves with the full set of notes, recording initial observations, and then coded the entire data set independently of one another, staying close to the participants' own formulations. The two sets of codes were brought together and collated into candidate themes, which were reviewed against the coded extracts and against the full material; themes not supported across the material, or duplicating one another, were merged or set aside. The remaining themes were defined, named, and their scope agreed, and the sixth phase consisted of writing the account in Section 4.

The two analysts arrived at largely overlapping codes and themes. Where they differed, the differences concerned the boundaries and naming of themes rather than the interpretation of individual extracts, and were resolved in discussion. No formal measure of inter-rater agreement was computed.

3.4 Ethics and data privacy

Participation was voluntary, with no academic consequence of participating or withdrawing. Participants were informed of their rights, including the right to withdraw at any point, and how their data would be used. All data was anonymized and all gave written informed consent. The Norwegian Agency for Shared Services in Education and Research (SIKT, reference number 946827, dated 26.03.2024) approved the study.

4 Findings

We report five themes. The first three address the research questions: what participants could see, how the narrative affected what they saw, and what the combination revealed about their understanding. The fourth, and emergent theme, concerns pre-existing code, and the fifth concerns feedback and self-assessment.

4.1 Theme 1: Making runtime state observable

The most consistent theme across the data concerns what the game let participants see, and the contrast to what the IDE showed. Participants expressed that the lack of visualization in Java makes it difficult to relate to, and how its abstract nature can lead to "faulty" understanding, making learning more difficult. The participants recognized the concepts from the OOP course in the game, but said the visualization made them easier to grasp: Java, lacking visualization, makes it harder to understand what is "happening in the code", how it is run, and what happens when changes are made to it. Several described compensating for this themselves: picturing a cat while writing a cat class, connecting "class" to games they play, or drawing on a lecturer's metaphors, for example *The Sims*.

In the game, this work was done for them. A person "popping up" was a concrete item of feedback that "made the abstract concept of 'object' concrete". The data panel displayed the properties of a selected instance with their set values, which allowed participants to identify object references and property names more easily and to discuss the difference between them; the effect of assigning a new value – giving a person a role, for instance – was directly observable there.

Participants said that seeing this direct result structured the information and presented the state of the objects in memory. By contrast, in IntelliJ the internal state of individual objects is hidden and there is no way to keep track of which instances exist in memory at any given point; several described the code in the course as "cluttered" for this reason. Interview participants likewise reported that as the concepts were made concrete they had a firmer grasp of the difference between a class and an object, between instances, objects and properties, and of the use of methods with and without the passing of arguments.

4.2 Theme 2: The narrative gives code a domain meaning

Where the first theme concerns what participants could see, the second concerns what it meant to them. Interview participants described using OOP techniques within a narrative in which people and house objects are visualized, and in which the task is to "give people a house", as making the work concrete, more enjoyable, and more relatable. They described the missions in terms of creating resources with which to solve tasks, which in turn motivated the creation of objects, and said the context helped them understand what to do next and why the different techniques were needed. Adding the first person object was a turning point, showing the connection between narrative, action, and visualization; so was seeing the knight appear newly equipped after they added his armor.

In the focus groups, the narrative gave participants a vocabulary for discussing requirements, and they saw the requirement that a person with a particular role could build a house from several perspectives at once. The most obvious is the code: the *Person* class had the method to build a house object, guarded by a check that the person held the appropriate role. Some groups discussed how to give a person a house, identified the *setOwner* method in the *House* class, and worked out that the owner – a person object – could be sent as a parameter to set the appropriate value to the *owner* property of the house. They understood this as creating a "connection" between house and person, an idea strengthened by the house's data panel showing the owner property set to a person (Fig. 2).

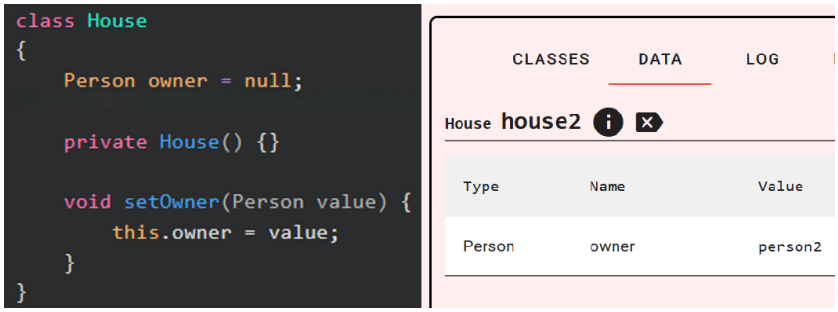


Fig. 2. The *House* class with its *setOwner* method, alongside the data panel showing the resulting owner of *house2*.

The narrative supplied the domain view: to build a house, a builder is required, and *setOwner* created an ownership relationship – a more concrete notion in the minds of participants than the aggregation or association relationships described in OOP. Discussing further mechanisms was easier when supported by both perspectives at once: participants understood subclassing both technically (*Forge* extends *House*) and narratively (a forge is a kind of building). The groups thus discussed not only the pragmatic view – "if one can change the code, one can change the role" – but also what those changes meant in terms of the domain.

Separating the domain from the mechanisms in this way gave participants a basis for reasoning about which mechanisms were relevant. Asked how they would add an address to the houses, the groups weighed holding it as a String in the *House* class against creating a separate *Address* class – arguing from default values and later adjustability on one side, and from the amount of information an address carries on the other – and settled on the second option.

4.3 Theme 3: Observability exposes mismatches in students' models

The third theme is, in our view, the most consequential. Participants appeared to have a good grasp of basic OOP concepts, but some misunderstandings persisted, and the feedback from the game surfaced them. Where instantiation and

object creation behaved as participants expected, the visualization confirmed their understanding and strengthened their confidence in it. Accessor methods, identity and uniqueness, and terminology did *not* fully match expectations, and were opened up for discussion as a result.

Terminology was one recurring site of mismatch. Asked directly, participants could provide good definitions for class and instance; in discussion around the topic, they used the same terminology inexactly or incorrectly, ranging from forgetting the meaning of "scope" to confusing instantiation with inheritance. Confusion persisted between a class, an instance of a class, an object, and the reference to an object: participants described instances as "instances of the same object", and in one case an instance was described as "the child" of its class.

Object identity was another. One interview participant, reflecting on the fact that the persons in the game were drawn with different appearances, described the expectation the visualization had contradicted:

> I thought they would be identical, that the persons would be identical. I thought it was like a simple button to generate a random number. Here it is rather a random person... not to be a racist, but a different person.

In the focus groups, an issue surfaced through comparison with other courses. Participants saw parallels between OOP and databases, treating classes and tables as equivalent and objects as table entries, but used the terminology and narrative of the game to clarify some of the distinctions: one stated that "a [database] table can't cut wood". Misconceptions about uniqueness and identity across OOP and databases were harder to resolve, but were identified and made discussable. Participants also compared Java to the JavaScript they had used the previous semester, describing JavaScript as more visual, incremental, and reactive, and Java as "more theoretical and harder to understand because it is less visual".

What is notable in this theme is not that participants held misconceptions, but that the misconceptions became visible, to us and themselves, in the course of ordinary interaction with the game.

4.4 Theme 4: Constrained code shifts attention from writing to reading

This theme was not anticipated and emerged from the material. Programming education often focuses on writing code, which leaves students with less practice reading existing code and working under its constraints. The game requires reading the mission text and the code provided in order to address the task: the narrative supplies the domain, the existing code the technical context. Participants noted that the game "not necessarily focuses so much on writing code...", but that "in the game you still need to understand things".

The respondents contrasted this with tasks given after lectures and at the examination, where they were in control of the code and could make any adjustment a solution seemed to require. The existing class structure focused the tasks

and made them more incremental, but also imposed constraints to work within. One participant said the narrative made them understand what to do but not how, because it took time to realize that they had to make use of the existing code: "I did have the right method but didn't understand where to put it".

4.5 Theme 5: Immediate feedback supports self-assessment

The final theme concerns the granularity of the feedback participants received and what they did with it. Participants reported that getting feedback per line of code afforded understanding. In addition to the images, they received explanations of the code, "mission complete" messages, and error messages. This contrasts with IntelliJ, which in their description offers only "red messages and everything crashes", and with course exercises, where feedback may arrive only once the entire solution is in place. The modular structure of the missions, combined with the visualization, allowed multiple feedback points and made it easier to locate and fix mistakes.

Error handling was a related difficulty. Participants said identifying errors in the course was hard: the IDE supports syntax errors, but most errors are subtler and harder to diagnose without suitable visualization. Some described trying to visualize the stack trace and work backwards to the source; others worked out what to print to the console. Seeing which parameters held faulty values made the error clearer, and helped participants understand which parameters to check.

Interview participants used their progress in the game as an indicator of their skill level, saying it created awareness of which techniques they knew and which they did not. Some felt frustration at not being able to use methods as well as they had thought: the immediate feedback exposed a mismatch between expectation and actual performance. Asked how confident they would be explaining OOP to a peer using the game, three of the four reported increased confidence, and pointed to concepts they found hard to communicate without visualization: parameter passing, and the use of *Scanner* to read from the command line.

The feedback was not uniformly helpful. Some participants got stuck on minor syntax errors, which created frustration, and would have liked hints. They also discussed that having the game available in parallel with the course could have helped them understand concepts earlier, and expressed interest in it covering loops, inheritance, and database connections.

5 Discussion

5.1 Externalizing the notional machine

When a person acts in the real world, there is usually a perceivable result. When a student runs an object instantiation statement in the IDE, the process is hidden, there is no feedback. To see an effect, the student prints the object's properties to the console. It is not obvious that connecting instantiation to a printout helps construct a model of what objects are and how they work.

Our participants reported something different. A person or house object appeared after they ran their code, connecting the execution of a statement with a concrete manifestation of the object created, and the object's state was shown in the data panel and updated immediately after changes to its properties. The interaction between student and program was thus continuous rather than deferred to the end of a solution.

We interpret this in terms of the distinction drawn in Section 2.2. The notional machine is not visualized as such; it externalizes parts of it and makes the resulting program state observable. The chain from code, to runtime action, to object, to state, to relationships between objects is normally traversed only in the student's head, and only if they have a viable model to traverse it with.

Making that chain inspectable removes the need to hold all of it in mind at once, and gives the student something external to check their reasoning against. This also accounts for the reports of "cluttered" code in Section 4.1: participants noted that they could collect instantiation statements to get an overview, but code is not the state of the program, it is the instructions that produce it.

Design implication. An environment for teaching OOP should make more of the runtime state observable by default, rather than requiring students to instrument their own code to see it. Console output makes one value visible at one moment; what our participants used was a persistent, inspectable representation of which objects exist, what they hold, and how they are connected.

5.2 The narrative as a domain scaffold

Beyond the difficulty of understanding objects as distinct units of information and behavior, there is the difficulty of understanding how they interact. The narrative gave participants a language for referring to interactions between the phenomena of the game world: how a person object with the role of Builder can create a House object, and how a person object can then be set as its owner.

The interpretive point is that the narrative did not merely make the exercises more engaging. It supplied a domain in which the relationships between objects already have names and consequences. An ownership relation between a person and a house is something participants can reason about without first having mastered the vocabulary of aggregation and association; and having reasoned about it in domain terms, they had something to attach the OOP vocabulary to. The same held for subclassing, approached from both sides at once: *Forge* extends *House* in the code, and a forge is a kind of building in the world.

Design implication. A narrative is useful to the extent that its domain relationships map onto the program relationships the student is meant to learn. The value lies in the correspondence, not in the fiction, and exercises built on a coherent domain may achieve much of this without being games at all.

5.3 The game as a diagnostic environment

The finding we consider most significant is reported in Section 4.3: participants gave good definitions of class and instance, yet used the terms inexactly when

reasoning about the objects in front of them. This suggests a gap between what participants could state and what they could operate with – between a definition retained from teaching and the model actually recruited when reasoning about a concrete situation. A conventional exercise is poorly suited to detecting this gap, because a student who produces working code has demonstrated only that the code works. The game surfaced it because participants had to talk about what they were seeing while they were seeing it, and because what they saw sometimes contradicted what they expected. The participant who expected the generated persons to be identical is the clearest instance: the expectation was articulated only because the visualization gave it something to collide with.

The comparisons participants drew with other courses point in the same direction, though we treat them more cautiously. Participants mapped classes onto database tables and objects onto table entries, and some drew a parallel between a row's primary key and the value of an object's properties. Prior experience with visually responsive environments and with tabular data may shape what students expect execution to make perceptible, and how they think uniqueness is established. Our material is too thin to establish where these models came from, but enough to show that the game brought them into the open.

Design implication. An environment of this kind can be used deliberately as a diagnostic instrument, not only as practice. The value to a teacher lies less in whether students complete the missions than in what they say while completing them, which suggests using it in supervision and group work, where a teacher or student assistant has a shared object to point at and can ask what happens when a particular line runs. Our participants already used it this way with each other: the focus groups clarified the tasks to one another, and three of the four interview participants reported increased confidence in explaining OOP to a peer.

5.4 Feedback and self-regulated learning

The way participants used the game to gauge their own understanding invites a self-regulated learning (SRL) perspective, which we introduce as an interpretation of the findings rather than as a framework the study was designed around. Zimmerman [11, pp. 65–69] models self-regulation as a cycle of forethought, performance, and self-reflection, and Russell et al. [7] argue that fostering it depends on making self-regulation visible to students. Read through this lens, the missions supply goals, the visual output and error messages supply performance feedback, and progression indicates competence. Modularity matters here: writing one line and immediately learning whether it was right localizes any error to that line, whereas a course exercise may yield no feedback until the whole solution is in place, at which point the error could be anywhere in it.

We are cautious about how far to push this. Participants reported using their progress as a measure of their skill, and some noted the discomfort of discovering they could not use methods as well as first thought, something consistent with the self-reflection phase of Zimmerman's cycle – but we did not test it.

Design implication. Feedback granularity is a design variable in its own right. Reducing the distance between an action and the information about whether it succeeded appears to be valued, independently of the game framing around it.

6 Limitations

This study explores students' experiences of a prototype rather than learning outcomes. The material is qualitative and the sample small: 23 participants recruited by open invitation from two cohorts. The two instances also used different formats, so the two data sets are not directly comparable, and the group setting will have shaped what participants said.

Participants interacted with the prototype for 30–45 minutes, enough to form impressions but not to observe development over time, and they had completed the OOP course. Our interest was in what the game reveals about existing models. There was no pre- or post-measure: we report what participants said and did, not gains in understanding. The researchers were present throughout, likely influencing the interaction and what participants chose to report.

The analysis in this paper does not separate the effects of visualization and those of the narrative, taken individually. Further research could investigate the degree to which each of these components influences the observed effects.

The material was documented as detailed written notes, not audio transcripts, so quotations are close renderings, not verified verbatim speech. Much of the reported understanding is self-reported rather than demonstrated, a divergence that is itself a finding. The results therefore provide exploratory evidence of perceived affordances and conceptual tensions, not causal learning effects.

7 Conclusion

We studied how first-year students engaged with a serious game in which the runtime state of an OOP program is made observable within a narrative. Participants used the visual representation to connect the execution of a line of code with the creation of an object, including its subsequent state changes, and drew a sharp contrast with the IDE, where those effects are hidden. During the engagement, discrepancies in their understanding became visible: participants who could define class and instance correctly when asked used the same terms inexactly when reasoning about the objects in front of them, and expectations about object identity and uniqueness surfaced only when the game contradicted them. The narrative contributed a domain in which the relationships between objects had names and consequences, giving participants a vocabulary in which to reason about operations that are otherwise purely formal. An unanticipated mechanism emerged: because the object model was supplied and could not be modified, progress depended on reading existing code rather than on writing it.

The design implication we draw is that environments for learning OOP should make more of the runtime state observable by default, rather than leaving stu-

dents to instrument their own code, and that an environment which does so can serve as a diagnostic instrument as much as a practice tool.

To explore if the game produces measurable learning goals we would use a controlled design with pre- and post-measure alongside the course. We would also pursue a supervision setting: how conversations while using the game work to build models and reveal misconceptions. Extending the game to other techniques would widen and bring the range of misconceptions into the open.

References

1. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qualitative Research in Psychology **3**, 77–101 (01 2006)
2. Gutiérrez, L.E., Guerrero, C.A., López-Ospina, H.A.: Ranking of problems and solutions in the teaching and learning of object-oriented programming. Education and Information Technologies **27**(5), 7205–7239 (2022). https://doi.org/https://doi.org/10.1007/s10639-022-10929-5
3. Lian, V., Varoy, E., Giacaman, N.: Learning object-oriented programming concepts through visual analogies. IEEE Transactions on Learning Technologies **15**(1), 78–92 (2022). https://doi.org/10.1109/TLT.2022.3154805
4. Mellado, R., Cubillos, C.: Gamification improves learning: Experience in a training activity of computer programming in higher education. Journal of Computer Assisted Learning **40**(4), 1959–1973 (2024). https://doi.org/10.1111/jcal.13000
5. Meyer, J.H.F., Land, R.: Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. Higher Education **49**(3), 373–388 (2005). https://doi.org/10.1007/s10734-004-6779-5
6. Nylén-Eriksen, M., Stojiljkovic, M., Lillekroken, D., Lindeflaten, K., Hessevaagbakke, E., Fløløø, T.N., Hovland, O.J., Solberg, A.M.S., Hansen, S., Bjørnnes, A.K., Tørris, C.: Game-thinking; utilizing serious games and gamification in nursing education – a systematic review and meta-analysis. BMC Medical Education **25**, 81 (2025). https://doi.org/10.1186/s12909-024-06531-7
7. Russell, J.M., Baik, C., Ryan, A.T., Molloy, E.: Fostering self-regulated learning in higher education: Making self-regulation visible. Active Learning in Higher Education **23**(2), 97–113 (2022). https://doi.org/10.1177/1469787420982378
8. Sorva, J.: Notional machines and introductory programming education. ACM Transactions on Computing Education **13**(2), Article 8 (2013). https://doi.org/10.1145/2483710.2483713
9. Staugaard, J., Bennedsen, J., Seidl, C., Nicolajsen, S., Fink, M., Brabrand, C.: Visualizing the conceptual framework of object orientation for novice programmers. In: 2024 IEEE Frontiers in Education Conference (FIE). pp. 1–9 (2024). https://doi.org/10.1109/FIE61694.2024.10893141
10. Zeng, J., Sun, D., Looi, C.K., Fan, A.C.W.: Exploring the impact of gamification on students' academic performance: A comprehensive meta-analysis of studies from the year 2008 to 2023. British Journal of Educational Technology **55**(6), 2478–2502 (2024). https://doi.org/10.1111/bjet.13471
11. Zimmerman, B.J.: Becoming a self-regulated learner: An overview. Theory Into Practice **41**(2), 64–70 (2002). https://doi.org/10.1207/s15430421tip4102_2