Toward Studying Example-Based Live Programming in CS/SE Education

Eva Krebs
eva.krebs@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Toni Mattis
toni.mattis@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Patrick Rein
patrick.rein@hpi.uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
Potsdam, Germany

Abstract

Source code is inherently abstract. While this is necessary to capture the generality of a program, it poses a barrier to understanding and learning to use the underlying concepts. In education, especially in abstract subjects like maths, the use of concrete examples is considered instrumental to the acquisition of knowledge and a frequently explored direction for teaching computational concepts. Besides concreteness, the importance of examples being close to their abstract descriptions as well as the immediacy of feedback has been highlighted.

Babylonian Programming (BP) is a type of example-based live programming that fulfills all three criteria by introducing concrete values, moving them as close as possible to the code, and updating them immediately in response to changes of either the example or the code. This makes BP a promising tool in education, yet no studies on the suitability of BP in a learning context have been conducted. Hence, we propose to (1.) investigate usability of state-of-the-art BP to minimize the friction of introducing BP in education, and (2.) measure the learning effectiveness and quality of experience of a BP environment in undergraduate software engineering education. For these studies, we will use the Smalltalk-based Babylonian/S as our environment.

Besides clearer guidelines on the design of BP and example-based systems in general, we expect to shed light on the qualities that teacher-provided examples need to exhibit and the opportunities for learners to create their own examples during experimentation with unknown concepts.

PAINT '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0399-7/23/10.
https://doi.org/10.1145/3623504.3623568

CCS Concepts: • **Software and its engineering → Development frameworks and environments**.

Keywords: live programming, exploratory programming, example-based programming, babylonian programming, examples, squeak, smalltalk, education

ACM Reference Format:
Eva Krebs, Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2023. Toward Studying Example-Based Live Programming in CS/SE Education. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3623504.3623568

1 Introduction

The abstract nature of source code poses a challenge to teaching programming and computational concepts. Fortunately, abstract concepts can be conveyed effectively by the use of examples, as the use of *worked examples* in both math and CS education demonstrates. Most examples in programming education either have the goal to illustrate the workings of an algorithm or program using concrete values (code tracing examples) or demonstrate how a program is supposed to be constructed (code generation examples) [7]. In practice, both learning objectives are interwoven: understanding the parts of a larger program (e.g., the standard library) by seeing them operate on concrete values can help learners form sub-goals that help them solve new tasks requiring code generation.

Muldner et al. [7] identified the problem of making examples available and semantically close enough to the learners' task as a major open research direction. There is also little research in how to effectively empower students to *self-explain* and experiment with code and computational concepts themselves. Moreover, the focus of the majority of previous studies is on relatively self-contained algorithmic examples that provide excellent opportunities to make use of rich visualizations due to their small scope, but such approaches do not generalize to advanced topics, such as teaching design patterns or complex libraries.

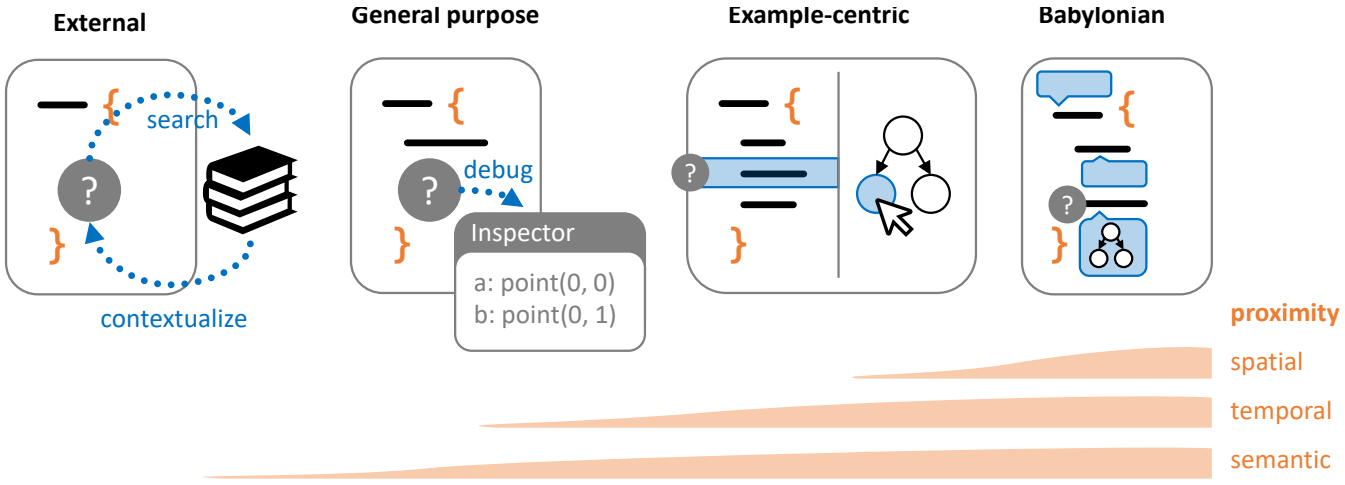

Figure 1. Spectrum of integrating examples into programming workflows.

What is known, however, is that the effectiveness of examples is highly influenced by their proximity to the abstract concept they illustrate: Examples that are separated from what they illustrate are less effective. This is known as the *split-attention effect* [3].

While the ideal situation of simultaneously presenting an example for an abstract concept can only be achieved by using sensorially different channels (e.g. listening to instructions while seeing a concrete example unfold), using the same medium, such as the computer screen, can likely benefit from bringing abstract and concrete elements closer together in *space* - e.g., demonstrating the effect of a line of code next to it - and in *time* - e.g., having up-to-date examples available immediately. A third dimension, *semantic proximity*, can play an important role if the goal is to understand a concept well enough to re-use it. Examples with a large semantic distance are not perceived as applicable to a problem (however, an exact semantic match invites copying and does not contribute much to a learning outcome). For instance, demonstrating a sorting algorithm with numbers helps understand the algorithm itself, but not neccessary how it can be used to sort composite data.

1.1 From Textbooks to Live Examples

Understanding source code involves mental simulation. Especially when learners lack sufficient experience to quickly recognize plans, fine-grained mental simulation can take most of the effort to understand a program - sometimes to a degree that requires pen and paper on the side. At the same time, abstractions they encounter or need to use to solve a task (e.g. standard library calls) might be opaque as long as the vocabulary is still unknown. Although concrete examples might be available (e.g. textbooks, documentation, or StackOverflow), learners must deliberately seek them out, recognize which are useful, and map them back into their current task context, which is a distinct skill on its own.

IDEs and examples. Modern general-purpose programming environments introduce interactivity to support both mental simulation and understanding of abstractions to some degree. A read-eval-print loop (REPL) might invite experimentation, but learners must come up with suitable examples to try. Automated tests can contain relatively complete examples that run parts of the system, but they are often located away from the code and thus harder to discover. The PyRet language [1], in contrast, motivates tests directly following the definition of a function via its `where`-syntax. Besides the challenge of generating or finding suitable examples, learners are also facing the challenge of learning about how the program operates on the example data. The print statement is a ubiquitous way to observe dynamic behavior, but the resulting log needs to be put back into context (which might require careful formatting to recognize which statement produced which output) and is typically limited to textual representations. Most debuggers allow stopping the program at any point and inspecting its state. This allows learners to access a rich representation of composite data at a particular point in time, unfortunately with little support to easily keep track over time or quickly return to certain point after changes have been made.

Example-centric programming. These problems have subsequently been addressed by treating examples as first-class entities in the programming environment rather than an artifact that needs to be manually constructed from, e.g., tests and print statements. Example-centric programming [5] first attempted to provide programmers with a side-by-side view of how state evolves as code is being run, e.g., from a

[1] https://pyret.org/

test case. In Newspeak, Exemplars [1] are annotations that provide examples to methods, so that code is always ready for evaluation. By restricting the domain to certain data structures, richer visual representations of state evolution can be used - an example is the Kanon [9] system that synchronizes data structure visualization with step-wise code execution, highlights changes, and maps them back to the code that caused them. By maintaining the impression that the observable behavior of the example immediately responds to changes to either the program or the example - which is called liveness [13] - a class of systems named Example-based Live Programming (ELP) encourages experimentation and exploration by avoiding re-compile and re-run cycles.

Babylonian Programming. While side-by-side views of code and example can implement rich visualizations on the "example side", the program itself remains spatially separated from the concrete state and behavior. Babylonian Programming (BP) [10] is an ELP system that moves examples even closer to the code by displaying concrete data at expression level. The example is always executable and reacts to changes immediately, e.g., the displayed intermediate values are always reflecting the most up-to-date behavior.

In summary, examples in programming environments form a spectrum with regard to how close they are to the source code, with external documentation being the least accessible and semantically the least related (see Figure 1). General-purpose IDEs technically support examples but not very well. Example-centric environments are designed to support examples - often in a side-by-side view, and Babylonian Programming further integrates first-class examples into the code itself. However, BP has not yet been studied in educational contexts, although it optimizes several dimensions that can appeal to learners: proximity of examples minimizes the split-attention effect and displaying dynamic information over time supports mental simulation. BP can serve a double role in education as it allows educators to annotate code as a form of live documentation and invites learners to experiment due to its liveness.

2 Babylonian Programming

Babylonian Programming (BP) is inspired by the way ancient Babylonians expressed their algorithms - in terms of concrete examples right next to the instructions [10]. Fortunately, we are not restricted to clay tablets anymore. A BP-enabled IDE introduces several concepts: (1.) the Example[2], (2.) Probes as a way to observe concrete behavior, and (3.) Replacements to override expressions with user-controlled values as illustrated in Figure 2.

Example. In BP, an Example provides concrete values to be able to run a particular section of code and, optionally, display its final result. In object-oriented environments, this includes example instances of classes, so that a realistic value of `self` can be assumed, as well as example arguments needed by a method call. Examples can be created using specialized tools, e.g., by writing their set-up code manually or selecting and persisting concrete values observed during run-time.

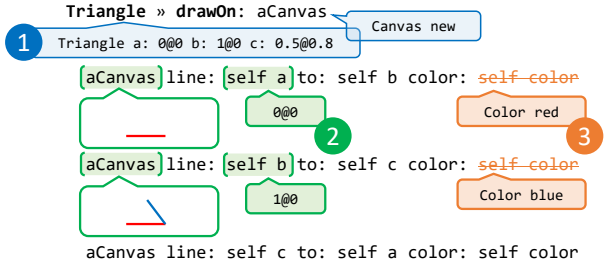

Figure 2. Core elements of Babylonian Programming: The programmer-curated Example (1) provides an instance of the class and arguments for the method call to construct a full execution context. Probes (2) render dynamic data captured during execution of the method – here, the content of the canvas – and are always updated immediately after code or Example changes. Replacements (3) allow programmers to isolate the Example from irrelevant state by skipping the execution of an expression and proceeding as if it evaluated to certain value.

Probe. A Probe can be attached to any expression, showing its value under the currently active Example(s). Probes are updated immediately on each change, i.e., the affected code path is being re-executed in the background. They can use rich, domain-specific visualizations, e.g. displaying the content of a drawing buffer to help users trace its evolution. If they are affected by multiple times (e.g. in a loop), they can visualize their value's evolution either textually or using domain-specific representations again, such as sparklines. Probes can be anywhere in the control flow of an example, allowing users to trace concrete behavior deep into method calls.

Replacement. A Replacement can override an expression and provide a fixed value instead. This can make an Example more self-contained by avoiding unrelated computation, e.g., bypassing user input by just assuming they provided certain input. It also encourages experimenting with what-if scenarios, e.g., by allowing to see the behavior if certain call returned something else.

2.1 Implementations of Babylonian Programming

Babylonian Programming was initially implemented in the web-based live programming environment Lively4 [6] for the JavaScript language as Babylonian/JS [10]. A subsequent

[2]We capitalize the term to refer to the live Examples in Babylonian Programming rather than the generic term

extension of the Language Server Protocol (LSP) enabled a language-agnostic implementation in Visual Studio Code based on the GraalVM runtime [8]. The implementation with the most complete integration into existing tools, however, is an implementation in Squeak/Smalltalk called *Babylonian/S* [12]. It provides access to examples, probes, and replacements directly in the code editor, debugger, and inspection tools. Figure 3 shows a screenshot of Babylonian/S. Due to its seamless integration, we will use *Babylonian/S* for our studies.

2.2 The Case for Babylonian Programming in Education

The design of BP targets programmers in general, but we argue that its core concepts lend itself to overcome learning obstacles rooted in the split-attention effect. Additionally, BP has the potential to support mental simulation by observing the effects of any statement or expression in terms of concrete values.

Assisting sub-goal formation through live documentation. Examples can be used to equip existing functionality (e.g. from the standard library) with a live documentation that helps learners identify the building blocks needed to solve a programming task. By providing an example-based big picture of the environment and its concepts, we expect examples to provide cues that positively affect the formation of sub-goals [2] during programming tasks, e.g., seeing a concrete demonstration what certain functions do might be more approachable than traditional documentation and learners might be encouraged to try out functionality they might have missed or re-implemented otherwise. Hence, we will explicitly study situations in which learners need awareness of code available in their environment to solve a programming task.

Assisting self-explanation through easier simulation. Effective learners often engage in self-explanation [4], a process that draws on existing background knowledge and newly generated hypotheses to make sense of a problem or a solution presented as worked example "in their own words". This may even include visualizing processes using pen and paper. In the context of teaching programming, learners are frequently observed to give explanations in terms of the concrete dynamic behavior they observe - for example the results of print statements [14]. BP has the potential to assist self-explanation by making it easier to mentally trace what a program does, possibly eliminating the need to use print statements or minimize situations that demand pen-and-paper self-explanations.

Learning non-localized concepts. Previous approaches to improve the learning experience and effectiveness focus on individual algorithms and smaller programs. This is needed for beginners and allows the use of rich specialized visualizations in live examples. BP has the potential to better support advanced learning goals because Examples can be traced through arbitrarily nested calls, allowing realistic examples to co-exist with small examples that illustrate basic principles. For example, collection functions might be documented using lists of integers as examples, while a part of a game is equipped with a realistic example simulating player behavior. If that part in turn uses collection functions, learners can explore realistic usage scenarios with live data as well. Teaching architectural concepts, e.g. design patterns, can benefit as well as BP easily scales to concepts spanning multiple components.

Approaching other domains through programming. The learning objectives supported by BP are not constrained to programming concepts alone. Many programs eventually model real-world domains. Working on such programs can be an effective way to teach that domain (e.g. teaching basic laws of ecosystems using a cellular automaton). BP allows teachers to introduce executable domain-specific examples. From this perspective, the program is a notation to formalize the phenomena in the domain and Examples make this notation approachable. Future implementations of BP might even support domain-specific visualizations.

3 Studying Babylonian Programming

Babylonian Programming is a novel concept not yet established in an industrial or educational context. Thus, we propose two studies: A study that aims to provide a broad understanding of how Babylonian Programming can be used and a second study that is specifically designed to test its use in an educational context.

While there are specific use cases for Babylonian Programming and its tools, there is currently little knowledge on when and how it is used in general because of its novelty. In order to study the effect of examples and Babylonian Programming in education, we first need to enhance our understanding of it in general, such as which programming domains it is most suitable for, in which ways developers create and use examples, and so on. This study would also uncover possible limitations of Babylonian/S, our chosen Babylonian Programming environment, and its tools that should be addressed before conducting a more specialized study.

3.1 Study 1: General Usage of Babylonian Programming

In this first study, we plan to observe how participants use Babylonian Programming. We both aim to see in which situations or domains Babylonian Programming is especially applicable for later studies as well as deepen our understanding of how Babylonian concepts are used at all. This will also enable us to address certain limitations of the concept or the

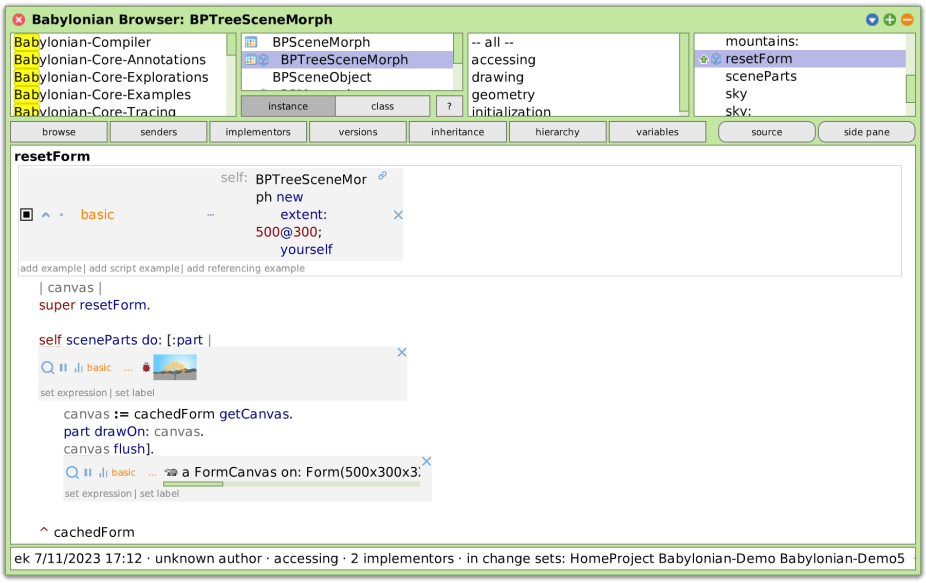

Figure 3. A screenshot of Babylonian/S. Examples are defined at the top of the method, probes are added for visualizations directly in the code itself.

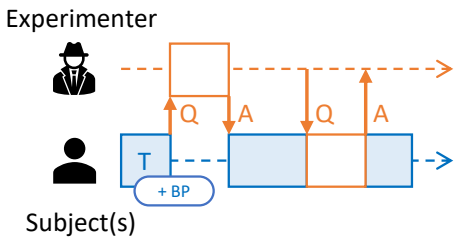

Figure 4. Exploratory study design: Participants working on a self-chosen task can request assistance and experimenters can ask clarifying questions.

concrete tools before conducting further studies with and on Babylonian/S.

Study Structure. We plan to observe participants working in a Babylonian/S system provided by us. We might ask questions during the study to confirm why participants decided to do a specific interaction with the system, if they feel blocked or unable to do something(see Figure 4). From this, we hope to gather insights on when and how participants use Babylonion Programming. Interesting topics include:

- In which program domains is it used? Is Babylonian Programming suitable for the task the participant is working on?
- In which programming situations is it used? Do participants use it to explore the system? Is it used for debugging?
- How do participants get examples? Do they e.g. write scripts or they use live objects from the system? Do they have trouble with creating examples?
- Which Babylonian/S tools and features are used?
- What do participants want to do but cannot?

Participants. We plan to recruit participants with varying knowledge levels from our faculty. As later studies, such as the second study detailed in subsection 3.2, are aimed at undergraduate students, people from that group will also be recruited for this study. All participants will receive a short training on Babylonian/S and the general live programming features of Squeak/Smalltalk so that no pre-existing familiarity with the system is required.

Task Design. As we want to observe how participants use Babylonian Programming with as much freedom as possible, we will try to observe them exploring the system or working in self-chosen project. However, should we not find enough participants with projects suitable to out study or should all projects fall into only one or two domains, we will intervene by preparing tasks that cover a large variety of domains.

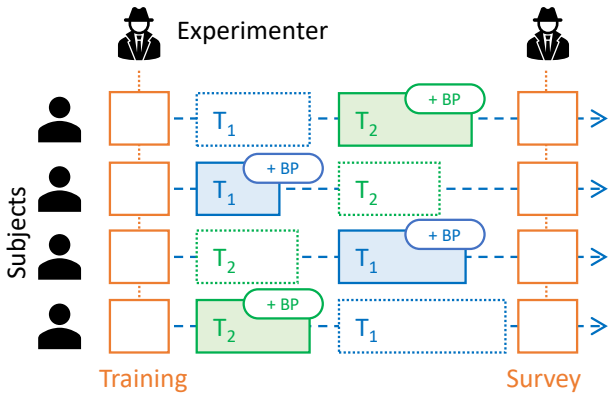

Figure 5. Controlled study design: Participants receive training and perform distinct tasks either under control conditions (dotted boxes) or with Babylonian Programming and Examples enabled (+BP) using a within-subject design. Follow-up surveys sample their experience.

3.2 Study 2: The Effect of Babylonian Programming on Educational Tasks

This study aims to evaluate if Babylonian/S has the desired effects on the completion of educational tasks by students. More specifically, the goal of this study is to gain insights on two major aspects of coding in an educational context: (1) We would like to see whether Babylonian Programming influences the learning effectiveness and (2) we would like to see how it impacts the students programming experience. We plan to focus on the following questions for evaluation:

- Does Babylonian/S improve the correctness of results that students create?
- Does Babylonian/S improve how engaging students perceive tasks to be?
- Does Babylonian/S decrease frustration with the task in students?
- Does Babylonian/S improve the confidence of the students in their solution?
- Does Babylonian/S influence how long it takes students to finish a given task?

Study Structure. Tasks will be completed in one of two experimental conditions: in the control condition, students have access to a standard Squeak/Smalltalk image including its live programming tools and features. In the experimental condition, students are provided with a Squeak/Smalltalk image that also includes Babylonian/S with all its features and pre-prepared examples that are relevant to the given task. These conditions aim to give us insights on whether a Babylonian Programming system with some examples given, as they could be e.g. by a teacher, has an effect on the programming experience of the students (see Figure 5).

Participants. Our target demographic for this study are undergraduate students in their fourth, or a later, semester of study from our faculty. Because the participants are undergraduate students, we expect them to be at a relative novice level of programming that are still learning about some of its core concepts. Because we are familiar with the teaching program of our faculty, we can anticipate which base concepts the students have already encountered in previous courses and prepare tasks accordingly. In particular, we can plan on all students having at least worked a little with a Squeak/Smalltalk environment in a mandatory lecture in their third semester.

Operationalization. In this study, we will equate "learning effectiveness" to whether students are able to correctly solve the given task. We want to enable students to correctly understand and answer the given tasks without, if possible, slowing down the task solving process so considerably that it becomes unviable in actual courses. To measure the impact on learning effectiveness, we plan to record the time students needed to create their solution as well as if their solution provides the behavior or functionality required by the task.

But since education is not exclusively focused on the results but also on how students achieve these results and how sustainable their learning experience is, we also plan to study their "programming experience" with post-task questionnaires. We aim to provide question that gauge how frustrating completing the task was, how "fun" or engaging it was, and how high their confidence in their result and understanding of the system is.

To understand some underlying aspects of our environment, we also plan to record how often participants switch context (in this case, methods or tool windows) and how often as well as why they use Babylonian features. Context switches usually negatively impact performance, as developers need to do a mental switch; if Babylonian reduces the need for such switches, that might also influence participant performance. The measures on our Babylonian/S tools will provide us some background information on how they influenced the participants; for instance, if a participant decides not to use Babylonian tools even if provided, a performance boost in that task would not be the result of our system.

Lastly, while we will be able to predict some characteristics and pre-experience of the expected participants, their exact knowledge may vary. Participants may for instance have additional experience from hobby projects or jobs outside of the university syllabus. Also, other factors such as exactly how many semesters the students have been studying, which courses they successfully passed, and their own confidence in their skills may vary as well. Because of these possible variations, we will include a questionnaire for the participants that besides demographical data will collect data on their experience and prior knowledge.

Task Design. This study will include multiple educational tasks suitable for our target demographic, e.g. feature creation tasks for a small game, a domain students would be familiar with from previous courses. The exact tasks and domain depends on the insights from our first study and previously outlined criteria[11] to ensure adequate complexity and answer times. To get insights on each educational task with both of our experiment conditions, we plan conduct this study as a factorial experiment and as a within-subject study.

Ideal would be tasks whose complexity is simple enough for the participants to grasp in a session, while being complex to include e.g. dynamic behavior or state. Enabling an understanding of dynamic behavior of a system is both a core potential of Babylonian Programming as well as integral to our targeted participants, computer science students that just finished their programming introduction courses.

4 Outlook

Study implementation. First, we will design a concrete plan for our first study design. We will then recruit fitting participants and run the study. As this is an exploratory study, we might change the study set-up between study runs based on new insights. The study results will then both be used to alter and improve Babylonian/S and gather knowledge about the applicability and usefulness of Babylonian Programming. The Environment used for the study will be equipped with fine-grained monitoring facilities that allow us to learn whether, when, and how often participants interacted with certain features of the base system and the elements introduced by BP.

Using the insights gained from our first study, we will then design tasks for and create a concrete plan for study. This set-up will be tested with a few students in a pilot study to find and fix flaws or problems that might occur. The finalized plan will then be used to run the study with recruited undergraduate students.

Based on the results, further studies could potentially be designed. For example, an entire seminar or lecture using a BP-enabled programming environment in a longitudinal study.

Expected improvements. We expect that such a study helps us improve several aspects of BP itself, our Babylonian/S implementation, and the opportunities students have to learn a novel programming system like Smalltalk.

A common issue with tools that deviate from the standard tool set is that they are unexpected and need to be designed with affordance in mind. When viewed under the educational lens, we expect to be able to tell helpful from less helpful features, learn how to make Examples and the tools to work with them more discoverable, which obstacles might prevent learners from creating their own Examples, and eventually arrive at an environment where Examples do not feel like a separate tool.

While the quality of the BP environment and the examples provided by teachers can confound learning effects, our double study is designed to mitigate this effect in the first study by allowing us to run the second study with a BP-enabled environment devoid of serious flaws detected earlier.

Conclusion

The evolving integration of examples into programming environments increasingly affords teachers and learners opportunities to illustrate abstract programs with concrete examples. Babylonian programming is a newer technique that promises to empower teachers and learners due to the proximity of examples to code, liveness, and applicability to larger programs.

In this paper, we outlined the opportunities that come with Babylonian Programming to support mental simulation, sub-goal formation, and self-explanation in the context of educational material and exercises. We subsequently proposed an early design for two studies aimed at exploring this promise and elaborated on the necessary preparations, potential outcomes and consequences for both teaching and programming environments.

Acknowledgments

This work is supported by the HPI-MIT "Designing for Sustainability" research program[3].

References

[1] Gilad Bracha. 2021. Enhancing Liveness with Exemplars in the Newspeak IDE. https://newspeaklanguage.org/pubs/newspeak-exemplars.pdf.

[2] Richard Catrambone. 1998. The Subgoal Learning Model: Creating Better Examples so That Students Can Solve Novel Problems. *Journal of Experimental Psychology: General* 127, 4 (Dec. 1998), 355–376. https://doi.org/10.1037/0096-3445.127.4.355

[3] Paul Chandler and John Sweller. 1992. The Split-Attention Effect as a Factor in the Design of Instruction. *British Journal of Educational Psychology* 62, 2 (1992), 233–246. https://doi.org/10.1111/j.2044-8279.1992.tb01017.x

[4] Michelene T. H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. 1989. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science* 13, 2 (April 1989), 145–182. https://doi.org/10.1016/0364-0213(89)90002-5

[5] Jonathan Edwards. 2004. Example Centric Programming. *ACM SIGPLAN Notices* 39, 12 (Dec. 2004), 84–91. https://doi.org/10.1145/1052883.1052894

[6] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a live development experience for web-components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience, PX/17.2, Vancouver, BC, Canada, October 23-27, 2017*, Luke Church, Richard P.

[3] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

Gabriel, Robert Hirschfeld, and Hidehiko Masuhara (Eds.). ACM, 28–35. https://dl.acm.org/citation.cfm?id=3167109

[7] Kasia Muldner, Jay Jennings, and Veronica Chiarelli. 2022. A Review of Worked Examples in Programming Activities. *ACM Transactions on Computing Education* 23, 1 (Dec. 2022), 13:1–13:35. https://doi.org/10.1145/3560266

[8] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-based live programming for everyone: building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020*. ACM, 1–17. https://doi.org/10.1145/3426428.3426919

[9] Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. 2018. Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. Association for Computing Machinery, New York, NY, USA, 72–87. https://doi.org/10.1145/3276954.3276962

[10] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[11] Patrick Rein, Tom Beckmann, Toni Mattis, and Robert Hirschfeld. 2022. Toward Understanding Task Complexity in Maintenance-Based Studies of Programming Tools. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (Programming '22)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3532512.3535223

[12] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts. In *Proceedings of the Workshop on Context-oriented Programming - COP '19*. ACM Press. https://doi.org/10.1145/3340671.3343358

[13] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (July 2018), 1:1–1:33. https://doi.org/10.22152/programming-journal.org/2019/3/1

[14] Vivian van der Werf, Efthimia Aivaloglou, Felienne Hermans, and Marcus Specht. 2022. What Does This Python Code Do? An Exploratory Analysis of Novice Students' Code Explanations. In *Proceedings of the 10th Computer Science Education Research Conference (CSERC '21)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3507923.3507956

Received 2023-07-17; accepted 2023-08-07