Can We Teach How to Explore Substrates and Systems?

Substrates'26 Workshop at ‹Programming›, March 16-March 20, 2026, Munich, Germany

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

Tom Beckmann
tom.beckmann@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Abstract

To use any authoring system, you need to get familiar with its concepts and available tooling. Especially if we want to create systems that are malleable and adaptable, users need to find out what they can change - and how they can change it. But how do we enable this? Substrates aim to be malleable for everyone, not just system developers; this likely means users should find answers within the user-facing part of the system, rather than requiring a switch to a lower level of abstraction, such as the virtual machine. Additionally, substrates might explore new interaction methods that differ from most established IDEs, making easy explorability and discoverability even more critical.

However, in our software engineering courses, we noticed that students struggle to explore new systems that aim to contain their own documentation. Even if a system has a variety of tools and is self-documenting, it can be challenging for a new user to know what to look for (and how not to damage the system accidentally during exploration). In this submission, we will report on some approaches we have tried and the struggles we encountered in our courses. This concerns both concepts that are commonly found in programming systems (such as debuggers) and tools that are likely specific to a particular system (such as specialized live-editing tools). We want to use this as a basis for discussion to collect approaches and experiences from other members of the substrate community. Additionally, we would like to facilitate a collaborative design session or conversation on what the future of teaching "system exploration" might look like.

CCS Concepts

• **Software and its engineering** → Open source model; **Development frameworks and environments**.

Keywords

substrates, self-sustaining systems, computer science education

1 Motivation

Effective use of programming systems requires building an intimate understanding of their tools and concepts. For example, given a wealth of tools for exploration of runtime state, such as debuggers, inspectors, probes, and printf, which tool is best suited to help comprehension in a specific circumstance? Other aspects of learning how to use a system are almost akin to tacit knowledge, e.g., effective navigation between windows or effective use of its editing facilities.

Substrates, as systems that may deliberately change many of the assumptions programmers and users have grown accustomed to in other systems, face a significant challenge in training their prospective users to use them effectively. Similarly, substrates as systems that seek to empower their users, may also expose actions to users that can bring the system into a difficult or impossible to recover from state—especially when novice users without a strong mental model of the system's concepts explore its possibilities.

Common approaches to address these challenges also limit users' potential: systems can standardize workflows to make them easier to teach. They can also provide helpful secondary documentation sources, like text or video tutorials, which, however, also add inertia to every change of the dominant workflow, as it would require updates in the secondary documentation, too. To address recovery, systems can hide powerful but potentially dangerous actions and only expose limited functionality.

One design of a system, which we explore in this paper, stands in contrast to these common approaches: it encourages users to learn about a system and its capabilities by exploring it and not limiting users' access to those capabilities. It may add safeguards and recovery mechanisms, but would not fundamentally forbid access to a capability of the system.

In two mandatory undergraduate software engineering courses at our faculty, students use a system with many characteristics of a substrate to implement software projects in small teams. The used system implements this design, in which learning focuses on system exploration and all system capabilities are fundamentally accessible. We will explain more about the system, Squeak/Smalltalk, and the course in section 2. Based on our experience, we collected approaches (section 3) we used to enable students to explore our system and briefly discuss the main opportunities and challenges we encountered (section 4). These approaches are not mutually exclusive; we often use multiple simultaneously to cover different situations, and because different people may learn best in different ways.

From our perspective, abilities that enable users to explore a system include, but aren't limited to:

- Learning and using system-specific tools.
- Handling errors, for example, reading error messages and exploring reasons for errors.
- Being able to search the system to determine which functionality (e.g., base data structures, libraries, or applications) is already implemented.
- Configuring and adapting the system if desired or needed.
- Recovering from error states that prevent normal use of the system.

Depending on the computer science concepts the system requires, additional knowledge may be needed to use them. However,

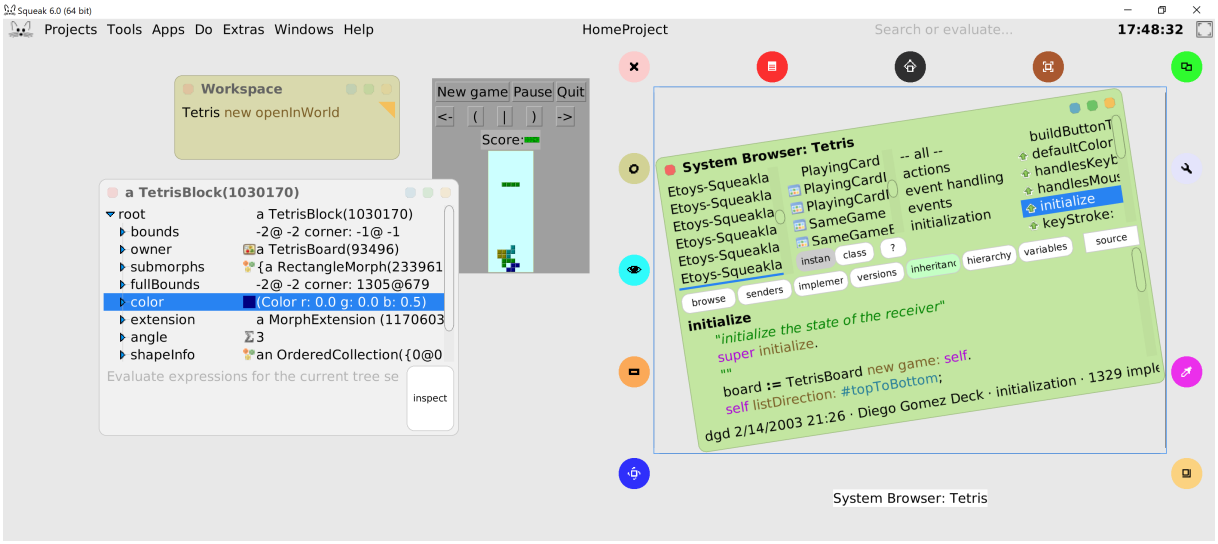

Figure 1: Squeak/Smalltalk and some of its tools

in this submission, the focus is on system exploration. Other relevant topics, for example, computational thinking, already have approaches in the form of research and shared educational materials that could also be used in the context of substrates. We will also only discuss approaches that directly aim to teach exploration; other factors, including but not limited to individual stress, the teacher's explanation style, and pre-existing views on programming tools, might, of course, also affect users, especially students in mandatory courses, in their usage of a system. We will briefly discuss related work (section 5), but mainly a small selection from the field of computer science education.

2 Background

The approaches and challenges discussed in this submission are based on our experiences with students. It is not an exhaustive list nor applicable to all substrate users; it is a discussion starter on how we can facilitate system exploration.

2.1 What experience do we have with this?

At our faculty, we offer two mandatory software engineering courses: a software architecture course in the third semester and a software engineering course on agile development in the fourth semester. Each course has about 100 enrolled students. In addition to lectures, students work on software projects in small teams. The students use Squeak/Smalltalk[3, 4].

Squeak/Smalltalk is a self-sustaining system written in itself that aims to be self-documenting; nearly the entire system can be read (and changed). Only a few base methods are considered *primitives* and invoke the underlying C code of its virtual machine. Anything else, from list implementations to web clients, can be explored in the system itself. Squeak/Smalltalk comes with tools that allow some configuration (a toolbar toggle, window optics, text field behavior like auto-enclosing brackets, and so on). Still, since the whole system is accessible, users can also adapt it beyond the pre-configurations. Additionally, the self-sustaining nature allows for both live programming and live inspection of system state. This also comes with dedicated tools, for example, the *halo* meta menu that can be opened for any graphical object (Figure 1).

Yet students struggle to explore the system. We encountered a wide variety of situations; it can be as simple as students assuming there is no *Matrix* class, meaning they used neither the search bar at the top nor the *implementors* search, which can be invoked on any text selection. I can be that students get an error message with a debugger, but close the debugger without reading anything, and try to solve the problem via trial and error. Sometimes students accidentally break a part of the system, causing an endless loop or similar behavior. Instead of saving their not-committed progress via available recovery mechanisms (or asking us for help), they download the lecture's Squeak/Smalltalk distribution again and manually recreate all changes. While some students embrace the available live editing tools, such as the *inspector* and *halo* (Figure 1), quite a few never use them beyond a tutorial at the start.

2.2 How does this relate to substrates?

The field of substrates encompasses many different environments, each with its own challenges. This means no system, whether wholly a substrate or not, can encompass all possible substrate challenges and opportunities.

However, we think the system used in our course has some core characteristics that make it relevant for aspects found in many substrates. Firstly, the system is as open as possible; users can both explore how it works and adapt it to a large extent. Secondly, it has tooling and concepts that differ from established, often file-based programming systems. Thirdly, the system is on the low-resource-language spectrum; there is less publicly available code, tutorials, and similar resources than for other systems. Recent substrates might have even fewer available resources, since Smalltalk has been used since the 1970s.

3 Exploring Systems

In computer science education, it can sometimes be difficult to separate learning about specific concepts (such as object-oriented programming) from general techniques (such as exploring a given system). Over the years, we have collected several approaches that primarily or secondarily try to teach and encourage system exploration. Often, we use several approaches in tandem, as different situations and different people might require different things.

3.1 Asking the system: It's self-sustaining, isn't everything already available?

Since (basically) everything in Squeak is written in Squeak, students in theory already have access to everything they could ever want to know. But if we are new to the system, how do we even know what to look for? Or how to look for it?

For searching alone, Squeak/Smalltalk has several functionalities: There is a searchbar on the top right, you can do an *implementors/senders* search on any text selection and in any code browser, you can to fulltext search of text selection via a mouseclick context menu, a method finder where you define example input(s) and output, and so on. But having tools available isn't enough if users don't know they exist or which one is best for their use case. A full-text search might not find the method they are looking for if, for example, naming conventions differ across systems. The method finder could address this, but might seem more complicated to use than a text search or a natural language request.

If you are used to systems that are not malleable (or only through a shop-like plug-in interface), it can be difficult to start actively changing the system. This includes both deciding to adapt the system at all and actually making the change. If users try something and encounter an error that breaks the system, they might be discouraged from exploring the system further. Especially if they are not aware, or intimidated by, error recovery mechanisms. This creates a difficult balance: the system as a whole might aim to be as open and malleable as possible, but certain areas may require safeguards and warnings.

3.2 Asking a person: The Mentor

If people who are already familiar with the system are available, they can mentor a new generation to become familiar with the system. In our courses, we usually use both synchronous and asynchronous means for this.

At the start of each semester, we have a synchronous, in-person tutorial session with the teaching team present. Additionally, students may visit our chair or schedule a video call to ask questions. Additionally, we are available asynchronously via a Slack channel and email.

Synchronous formats often make it easier for educators to ask their own questions, as there is no potential wait time. Based on our experience, it is also easier to guide students to figure out the answer themselves, for instance, by guiding them through using the right tool instead of directly giving the answer. As with pair programming, it is very important to let them use the tools and not to "drive" them for them. However, this is only possible if the educators have enough time and energy for this. For instance, a single tutorial at the start does not replace regular use of tools and techniques. Also, not all students learn best in in-person formats or ask questions in such a direct manner.

Asynchronous formats work better for students who want to ask questions outside set times and for those who prefer indirect communication. Students might struggle to ask the "right" question, which can make it harder to guide them to general principles. Additionally, since educators can't just look at the student's live system, determining the best feedback for the student might be more difficult.

3.3 Asking, or consulting, a tutorial: From text to videos and riddles

Several kinds of tutorials can be used. We have received good feedback on video tutorials with dedicated chapter names. This combines on-demand, somewhat searchable information with visual information. Additionally, we provide textual tutorials, such as a digital version of the book Squeak by Example available via Moodle, the Terse Guide to Squeak, available directly in Squeak, and an FAQ based on students' questions from previous years. These tutorials support text searches, making them a useful on-demand tool. However, it can once again be difficult for students to keep an overview of what is available and how they can best search for their current challenge. Additionally, there are interactive tutorials, for example, for Squeak language basics. We use such tutorials in tandem with mentoring to give small, concrete goals to start working on. However, too many tutorials feel exhausting; people want to start working on their project. All types of tutorials must be maintained, as out-of-date material erodes students' confidence in the available material.

3.4 Asking technology: Stack Overflow, Google, and ChatGPT

In contrast to human mentors, technological solutions are always available to answer questions. Please note: in the context of this section, we only look at questions already asked on Stack Overflow and already written Google results. Students using technology to ask other humans questions falls under subsection 3.2.

In addition to availability, these tools have one notable advantage: they support natural-language questions. While it can still be problematic if students prompt in a wrong direction, they are in general freer in formulating questions than with a basic full-text search.

However, the actual learning experience can differ widely based on how these tools are used. Are the Stackoverflow answers, Google articles, and ChatGPT answers actually read? Or is the code/tooling description in them only followed, without reflection? Especially in low-resource systems, there might not be enough data to train or fine-tune LLMs. When using commercial LLM systems, the way the system answers may also change over time, requiring different prompts. LLMs may also hallucinate, though approaches to mitigate this are currently being researched.

Ethical Considerations Regarding LLMs. While LLM-based tools like ChatGPT and agentic tools offer exciting new possibilities, system communities should also reflect on the use cases they intend to use them for, if at all. These systems become more powerful the

more training data they have; training data that is unclear whether it should have been used in the first place, as often no permission was asked. When using commercial solutions, you might also become dependent on privately owned (currently US-centric) companies. Given the current climate crisis, we should also reflect on whether the power consumption during training and use is worth it (though this should, of course, also be asked in relation to other resource-intensive software).

4 Discussion

We have not yet found an ideal approach for teaching exploration that works for us. We cannot (and do not want to) monitor students the whole time they work on their project. However, students might not know what to use when a problem occurs. Because of inertia, they might not regularly use tools they have only seen once.

The Future. For our course, we are currently considering creating short challenges that teach different aspects of exploration and problem-solving. However, there could be many more approaches, such as creating material for student tutors (who have already taken the course once) that enables them to act as mentors. Easy-to-integrate natural language inquiries could make system exploration available to more people.

While Squeak differs from established IDEs, at its core, it is still a text-based editor. Structural editors, visual notations, and other non-text (or "not-only-text") interfaces found in substrates might face different challenges and opportunities.

5 Related Work

This related work section does not aim to be exhaustive. Firstly, because we originally encountered this topic in our role as educators and not in an academic context. Secondly, the field of potentially related insights is very broad. Computer science education often includes project work whose IDE or tool exploration may be related. There is also research specifically on the environments used in education, such as a comparison between IDEs and command-line development [1]. Education research in general has explored several directions, including different kinds of tutorials, students' inquiry behavior, and materials for specific topics such as debugging. Some of these topics, such as differences between tutorial types, have also been studied in the context of learning software tools [5]. The field of human-computer interaction is exploring interface design, from nested meta menus to voice interfaces. There might be even more fields; every field with authoring systems or exploration behavior could yield insights that help shape substrates and systems in general.

In computer science education, several tutorials or exercises aim to be as close to "real" development as possible and thus often include system usage. To learn the basics of a new programming language, for instance, there is a concept called *Koans* (consisting of programming riddles). Computer science koans were originally created for LISP as a list of riddles that build upon each other [2]. Today, many more koan tutorials have been created for more languages. Many of these tutorials can be solved interactively using standard coding techniques, such as the Squeak Koans tutorial [7]. A similar concept is the *Kata* tutorial, a term popularized by Dave Thomas [8]. There are several kata tutorials. Unlike koans, kata tutorials go beyond programming languages; for instance, there are katas to learn test-first programming. Additionally, educators are building their own IDE-integrated tutorials [6]. This field could be relevant to teach exploration as 1) these tutorials can be done in / using systems, and 2) the modular structure of many of these tutorials makes them convenient to integrate into existing material.

6 Conclusion

Among the approaches we tried, there is neither a perfect approach nor a perfect combination of approaches. While in-person, synchronous communication with a mentor can yield many insights for a student, it might not always be feasible, and exploration techniques, like many other things, have to be tried again and again to be truly learned. Asynchronous approaches struggle when students do not ask questions or cannot formulate the best question for the given situation.

We are debating whether dedicated training challenges focused on exploration and related concepts might help students and users in general learn to use the system better. There are challenges, for example, named Koans or Katas, for other aspects of using the system, such as programming language properties or various testing techniques.

At the workshop, we would like to learn about other people's views on this topic, potential approaches that have been tried, and other ideas for the exploration of future education.

Acknowledgments

We want to thank the rest of our teaching team, including but not limited to Patrick Rein, Marcel Taeumel, and our supervisor Robert Hirschfeld.

References

[1] Hussain Aljafer and Gary Cantrell. 2020. Learning and Teaching Undergraduate Introductory Programming Courses in Java – The Use of an IDE VS Command Line. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*. 992–997. doi:10.1109/CSCI51800.2020.00184

[2] Linda Weiser Friedman. 1989. *The Little LISPer* (3 ed.). Pearson, Upper Saddle River, NJ.

[3] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley.

[4] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 1997*, Mary E. S. Loomis, Toby Bloom, and A. Michael Berman (Eds.). ACM, 318–326. doi:10.1145/263698.263754

[5] Verena Käfer, Daniel Kulesz, and Stefan Wagner. 2017. What Is the Best Way For Developers to Learn New Software Tools? An Empirical Comparison Between a Text and a Video Tutorial. *The Art, Science, and Engineering of Programming* 1, 2 (April 2017). doi:10.22152/programming-journal.org/2017/1/17

[6] Olli Kiljunen. 2021. Teaching Students to Fix Programming Errors with Tutorials Embedded in an IDE. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research (Koli Calling '21)*. ACM, 1–3. doi:10.1145/3488042.3489969

[7] Github SWA Teaching Organization. [n. d.]. *Squeak Koans.* https://github.com/hpi-swa-teaching/Koans Accessed: 2026-02-20.

[8] Dave Thomas and Community. [n. d.]. *Coding Dojo - Kata.* https://codingdojo.org/kata/ Accessed: 2026-02-20.