

From Abstract to Concrete: Programming the Running System via Direct Manipulation and Live Feedback

Eva Krebs

Hasso Plattner Institute
University of Potsdam
eva.krebs@hpi.uni-potsdam.de



(a) Code (abstract behavior)

(b) Scene editor (blueprint)

(c) Game (running instance)

Figure 1 – Different views with varying levels of concreteness of the same program

Abstract

Code being abstract makes it powerful, but it is also more difficult to understand. Developers will have to map abstract code descriptions to the behavior of concrete, running program instances. Additionally, as many IDEs have different views for code and the running program, constant context switches are required. Some environments offer live programming capabilities to reduce the amount of context switches, but often have limitations that will require program restarts or a context switch to another view. In my research, I want to explore notations, tools, and systems that bring the abstract and the concrete closer together. I am exploring systems that provide concrete feedback directly where it is needed, require fewer context switches between views, and remove some of the limitations of other live programming systems. With my thesis, I plan to contribute the design and implementation of these systems as well as insights into the feasibility of the concepts via case studies, holistic analysis, and user studies. In the long term, my aim is to enhance the programming experience of developers by removing accidental complexity.

1. Motivation

Code is an abstract description of behavior, making it powerful by being able to address a variety of situations. However, this also makes it harder to understand for developers and requires a mental mapping between code and concrete behavior. Especially in domains that heavily feature user interaction, we want fast cycles of trying out the running program and incorporating the resulting feedback back into code (Calp & Akcayol, 2019). One possibility to fasten feedback cycles is live programming, where changes are materialized in the running program as quickly as possible (Tanimoto, 2013), which may in turn also heighten the developer's understanding of the system (Krämer, Kurz, Karrer, & Borchers, 2014).

However, several development aspects may still cause friction. Many IDEs consist of different views of the same program that require context changes, for example, splitting code, abstract UI definitions, and the running program (see Figure 1). Some live programming changes may not manifest without further intervention; for example, changes to initialization code will not apply to already existing instances.

In my thesis, I explore tools that address these and related challenges. I research both new ways of defining behavior as well as extensions of existing live programming concepts that reduce friction.

2. Background

In the following, I will introduce live programming concepts and tooling that my work builds upon. Afterwards, I will quickly introduce the Godot game engine that was the basis for example implementations of my tooling concepts.

2.1. Liveness and Direct Manipulation

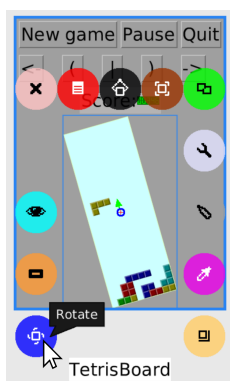


Figure 2 – Halo

Live programming usually refers to enabling developers to modify running programs and instances (Tanimoto, 2013). The capability of reloading code of running instances is also called hot reloading or live reloading in some systems. However, this does not mean all changes will immediately manifest in a running program. For example, initialization code of existing instances will not be re-executed, and code dependent on events or similar will still require the correct trigger/state to occur.

Direct manipulation encompasses tools that allow quick and easy changes to objects. For example, the live programming environment Squeak/Smalltalk (Goldberg & Robson, 1983; Ingalls, Kaehler, Maloney, Wallace, & Kay, 1997) features a concept called *halo*. A halo is a meta-menu that can be opened for any UI object. It provides tools available for that object ordered around its extent, see Figure 2. Common halo tools include, but are not limited to, object deletion, rotation, movement, and opening additional tools,

e.g., an inspector for the selected object.

Some of the tools presented in the approach are based on *babylonian programming* (Rein et al., 2019; Rauch, Rein, Ramson, Lincke, & Hirschfeld, 2019). Babylonian programming uses examples to enable developers to get concrete feedback on code behavior in the code editing view. For instance, developers can attach probes to expressions in code to receive domain-specific visualizations of the concrete values that the expression takes during example execution. A similar concept to probes are watches found in other IDEs that can visualize state of a running program.

2.2. Godot Game Engine

Game development is a domain that greatly depends on user experience and thus immensely benefits from faster feedback loops (Schell, 2019). Games are usually made with game engines, dedicated IDEs that bundle libraries and tools for recurring game development needs.

I am creating example implementations of my proposed tools for the game engine Godot (Foundation, Linietsky, Manzur, & contributors, 2007). Godot is an open-source game engine whose editor is largely self-sustained. This enables tool developers to both make powerful extensions without changing the underlying engine while also making engine changes possible if absolutely required.

Godot features three main views that are commonly found in game engines: a code editor, a scene editor (allows visual assembly of game elements in a scene tree), and the running game (see Figure 1). Additionally, it features some live programming functionality and related tooling:

- Godot supports **hot reloading**, meaning that code changes can be swapped into the running program.
- While a game is running, developers can use the tool **remote view** to see the scene tree of the running game.
- Godot features an **inspector** that can be used both in the scene editor but also to manipulate exposed variables of a running game.
- Furthermore, Godot supports common programming and debugging tooling, including but not limited to syntax highlighting, an integrated console that supports printing operations, and break-points.

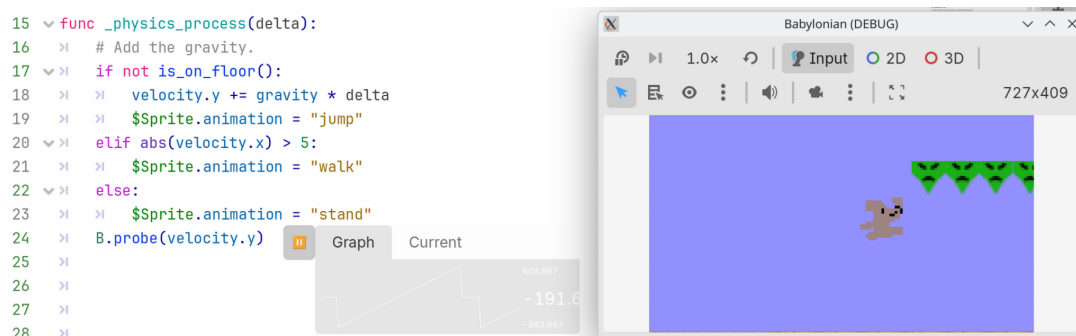


Figure 3 – Code augmented with concrete feedback through probes.

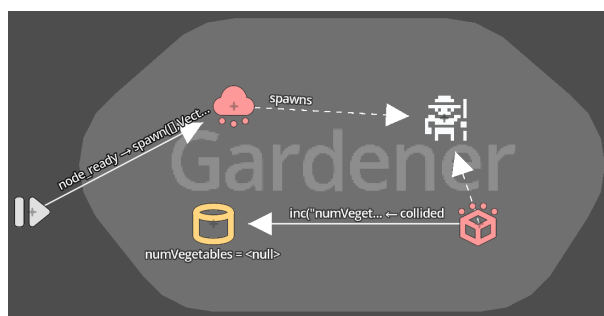
3. Approach: The Varying Levels of Concreteness

There are several ways of reducing context switches between abstract (e.g., code) and concrete (e.g., a running instance) of a program. In the following, I would outline some of the concepts I researching.

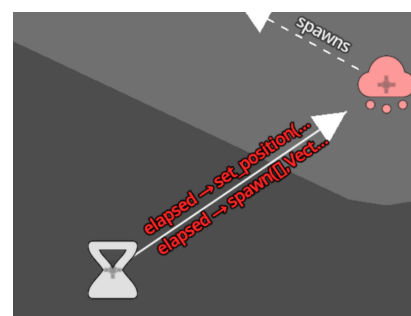
Probes and feedback Probes are an opportunity to bring concrete feedback to the abstract code view (Krebs et al., 2024). I am looking into how probe-based workflows compare to established workflows, e.g., using breakpoints or print statements. Additionally, I am exploring new types of probes, e.g., probes that are placed in the running program to enable further debugging/exploration while working on a concrete instance.

Prototyping in a single view In Pronto (Krebs et al., 2025), changes between code view and scene editing view are not needed (see Figure 4). All behavior is reified in the scene editor. Functionality that is commonly needed for game development is encapsulated in *Behaviors* that can be placed as visual elements in the scene editor. With *Connections*, developers can define how Behaviors should affect each other and/or game elements.

Keeping the running instance alive I am looking into removing friction from live programming sessions by making change materialization understandable, providing additional functionality to avoid unnecessary program restarts, and direct manipulation tools whose changes can be persisted on demand.



(a) Abstract behaviors are visually reified alongside game elements with connections between them.



(b) Connections flash when triggered in the running game.

Figure 4 – Pronto combines visual composition and behavior definition in a single view

4. Research Methods

My research aims to enhance programming experience by exploring different ways of reducing friction during both general development and live programming in particular. In the following, I will shortly outline my research goals and corresponding evaluation concepts.

4.1. Research Goals and Expected Contributions

I am exploring different ways of reducing friction during development. I am proposing tooling that addresses causes of friction, such as context switches between different program views and program restarts during live programming. The main underlying question I want to answer regarding my proposed systems and extensions is their trade-offs compared to state-of-the-art systems.

By addressing these challenges, I expect to make contributions in the following areas:

- Concepts for reducing friction
- Example implementations
- Insights on specific design challenges and tools from evaluations

4.2. Evaluation Concept

I plan to evaluate my proposed concepts and extensions via the following methods:

- I will show their feasibility through **reference implementations** and **case studies**.
- I will provide an initial argument for their impact on programming experience via **holistic analysis** with a focus on the cognitive dimensions of notations (Blackwell, 2005) and **cognitive walkthroughs** (Lewis, Polson, Wharton, & Rieman, 1990).
- Further insights on their impact on programming experience will be obtained via **user studies**.

5. References

- Blackwell, A. F. (2005). Cognitive dimensions of notations. In *2005 IEEE symposium on visual languages and human-centric computing (VL/HCC 2005), 21-24 september 2005, dallas, tx, USA* (p. 3). IEEE Computer Society. Retrieved from <https://doi.org/10.1109/VLHCC.2005.26> doi: 10.1109/VLHCC.2005.26
- Calp, M. H., & Akcayol, M. A. (2019). The importance of human computer interaction in the development process of software projects. *CoRR, abs/1902.02757*. Retrieved from <http://arxiv.org/abs/1902.02757>
- Foundation, G., Linietsky, J., Manzur, A., & contributors. (2007). *Godot engine*. Retrieved 2026-05-07, from <https://godotengine.org/>
- Goldberg, A., & Robson, D. (1983). *Smalltalk-80*. Harlow, England: Longman Higher Education.
- Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. C. (1997). Back to the future: The story of squeak - A usable smalltalk written in itself. In M. E. S. Loomis, T. Bloom, & A. M. Berman (Eds.), *Proceedings of the 1997 ACM SIGPLAN conference on object-oriented programming systems, languages & applications, OOPSLA 1997, atlanta, georgia, october 5-9, 1997* (pp. 318–326). ACM. Retrieved from <https://doi.org/10.1145/263698.263754> doi: 10.1145/263698.263754
- Krämer, J., Kurz, J., Karrer, T., & Borchers, J. O. (2014). How live coding affects developers' coding behavior. In S. D. Fleming, A. Fish, & C. Scaffidi (Eds.), *IEEE symposium on visual languages and human-centric computing, VL/HCC 2014, melbourne, vic, australia, july 28 - august 1, 2014* (pp. 5–8). IEEE Computer Society. Retrieved from <https://doi.org/10.1109/VLHCC.2014.6883013> doi: 10.1109/VLHCC.2014.6883013
- Krebs, E., Beckmann, T., Geier, L., Grenda, J., Ramson, S., & Hirschfeld, R. (2025). All in one: Rapid game prototyping in a single view. In N. Yamashita et al. (Eds.), *Proceedings of the 2025 CHI conference on human factors in computing systems, CHI 2025, yokohamajapan, 26 april 2025- 1 may 2025* (pp. 266:1–266:17). ACM. Retrieved from <https://doi.org/10.1145/3706598.3714251> doi: 10.1145/3706598.3714251

- Krebs, E., Mattis, T., Dörbandt, M., Schulz, O., Rinard, M. C., & Hirschfeld, R. (2024). Implementing babylonian/g by putting examples into game contexts. In E. Söderberg & L. Church (Eds.), *Companion proceedings of the 8th international conference on the art, science, and engineering of programming, programming companion 2024, lund, sweden, march 11-15, 2024*. ACM. Retrieved from <https://doi.org/10.1145/3660829.3660847> doi: 10.1145/3660829.3660847
- Lewis, C. H., Polson, P. G., Wharton, C., & Rieman, J. (1990). Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In J. C. Chew & J. A. Whiteside (Eds.), *Conference on human factors in computing systems, CHI 1990, seattle, wa, usa, april 1-5, 1990, proceedings* (pp. 235–242). ACM. Retrieved from <https://doi.org/10.1145/97243.97279> doi: 10.1145/97243.97279
- Rauch, D., Rein, P., Ramson, S., Lincke, J., & Hirschfeld, R. (2019). Babylonian-style programming - design and implementation of an integration of live examples into general-purpose source code. *The Art, Science, and Engineering of Programming*, 3(3), 9. Retrieved from <https://doi.org/10.22152/programming-journal.org/2019/3/9> doi: 10.22152/PROGRAMMING-JOURNAL.ORG/2019/3/9
- Rein, P., Lincke, J., Ramson, S., Mattis, T., Niephaus, F., & Hirschfeld, R. (2019). Implementing babylonian/s by putting examples into contexts: Tracing instrumentation for example-based live programming as a use case for context-oriented programming. In *Proceedings of the workshop on context-oriented programming, cop@ecoop 2019, july 15-19, 2019, london, UK* (pp. 17–23). ACM. Retrieved from <https://doi.org/10.1145/3340671.3343358> doi: 10.1145/3340671.3343358
- Schell, J. (2019). *The art of game design* (3rd ed.). London, England: CRC Press.
- Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In B. Burg, A. Kuhn, & C. Parnin (Eds.), *Proceedings of the 1st international workshop on live programming, LIVE 2013, san francisco, california, usa, may 19, 2013* (pp. 31–34). IEEE Computer Society. Retrieved from <https://doi.org/10.1109/LIVE.2013.6617346> doi: 10.1109/LIVE.2013.6617346