Online Object Observation
Connecting Interactions with Code

Tim Felgentreff

End-User Development, Software Architecture Group, Hasso-Plattner-Institut,
Universität Potsdam, D-14482 Potsdam, Germany,
tim.felgentreff@student.hpi.uni-potsdam.de

Abstract. Interactive, graphical systems offer rich opportunities to manipulate and program with graphical objects. Oftentimes, in systems such as Squeak or the Lively Kernel, end-users are able to "program" or "script" said objects through simple, graphical interactions, drag and drop gestures, and drawing connections between objects. In this paper, we explore *online object observation* as a means to ease the transition from graphical interaction to code. We argue that tools to observe objects can ease the transition from casual scripting to real program. Along the way, we experiment with different techniques to filter and refine tracing as well as help the user to find exactly the calls he is looking for with minimal overhead.

1 Introduction

Enabling non-programmers to employ computers for their specific means has driven the IT industry since the beginnings of widespread personal computing [1]. The IT industry is still young, and the sudden and dramatic increase in users in the late nineties [2] has left the industry vastly understaffed. Since that time, efforts to incorporate means for the end-user to extend and tailor software to their needs through "programming" have increased [1]. Today, 12 million people in the U.S. say they "do programming" at work, using spreadsheets, databases and other systems for end-user development [3] (EUD). Yet, there are only 3 million actually working as programmers within the U.S.

For non-programmers, languages with weak, dynamic typing have proven easiest to master, as can be seen especially recently with the rise of JavaScript [4]. In addition, most EUD is being conducted using simple, accessible concepts such as loops or declarative expressions, rather than more involved concepts. Recursion, model-view-controller or inheritance come to mind.

Sometimes, however, simple "programming" interactions become too limited to express the user's intent, and the underlying programming system has to be used directly.

As EUD technologies have evolved, the underlying systems and programming languages have become more advanced. This comes at a price, however. Where previously the step from scripting frequently used views into a financial database systems to writing actual code to store and retrieve data and connect to other

databases was but a minor one, today's systems hide so much of the underlying details that a user cannot gradually "outgrow" the controlled, scripting world and program on the metal. Even motivated individual might be discouraged when they discover that to advance beyond a certain point, they suddenly have to understand classes, inheritance and calling conventions. At this point, a user experienced with the interactive scripting system will have to re-learn how to create many of the interactive effects in code.

In modern environments geared towards EUD, the translation of manual steps into runnable code is not straightforward. In this paper, we explore object observation as a means to help users in this translation process.

2 Interactive Scripting Environments

In this paper, we will focus on systems for rapid-prototyping, that enable users to express reasonably complex ideas and build graphical tools to aid with daily tasks. Two such systems are the Lively Kernel [5] and the EToys [6] system build into Squeak/Smalltalk [7].

2.1 Squeak/EToys

The EToys scripting system runs on top of the Squeak/Smalltalk implementation. Development originally started at Disney in 1996 [7], and is now at Viewpoints Research Institute (VPRI). The primary goal of the language is to enable children to explore powerful scientific and mathematical ideas in an interactive, "life" environment. Today, it has grown into a full-featured media 2D/3D authoring environment.

EToys has been used successfully in different schools [6]. Especially for science experiments, teachers can provide pupils with a recording of an experiment and the children can try to deduct the laws that govern it by re-creating the essentials in EToys. They can import the video into their environment and draw shapes over it to represent the important objects. They can assign "scripts" to the shapes using simple drag and drop interaction. Those scripts can, for example, cause the shapes to move about the screen.

If children play with the parameters to their scripts, they can match the behavior of their shapes to the frames of the video backdrop and thus build an actual simulation of the real-world environment.

Furthermore, EToys can be used cooperatively over a network connection, allowing for immersive teaching experiences. Finally, as EToys builds on top of the Squeak system, the Smalltalk programming language is available for those who want to dive deeper into actual programming.

The Smalltalk system consists of a large class library, however, and for users of the EToys system the sheer number of classes and methods can be overwhelming — not to mention the widespread use of object-oriented patterns throughout the system, like observers, commands and method hooks.

2.2 Lively Kernel

The Lively Kernel is a Javascript environment that runs off a web page. It was originally developed at Sun Microsystems Laboratories. It consists of a Morphic based UI, a Smalltalk inspired IDE and provides rich shape composition and scripting facilities.

While not meant for children, it can be used to rapidly prototype graphical concepts or web mash-ups, and has been used to such ends [8]. Shapes can be combined and referenced by name, connections can be drawn to other shapes and those connections can be extended with scripts, to transfer and convert data between the different widgets.

This, and the fact that such "applications" can then be saved as a web-page, makes it very easy to create and distribute a simple application. Integration with existing Javascript libraries is also possible, and the step from scripting to full-on Javascript development is not quite as far as EToys is from Smalltalk. However, the environment is different from what developers are used to in a browser. Graphical widgets can be much more complex in Lively than the native widgets the browser offers. The inner workings of Lively widgets are not trivial to understand for casual Javascript developers.

3 Online Object Observation

To alleviate the aforementioned problems – wading through a large implementation trying to find a specific feature or trying to figure out the workings of a particular object – we explored live tracing in the Squeak environment.

Requirements Our first goal was to create a simple, intuitive way to observe a graphical object. We opted for a tool in the *Parts Bin*, an object repository users of the EToys systems are familiar with. The tool can be dragged out of the Parts Bin and be dropped onto the morph of interest. Because the morphic system uses the composite pattern to build complex shapes from simpler ones, we opted to not only trace into the morph that received the drop, but also its submorphs.

Secondly, we wanted to make sure that interactions and method activations could be correlated in a time-based manner, i.e., the instant in which a method is triggered through user interaction, the user should be able to see it printed to the tracer log.

Finally, we made it a requirement to keep the overhead of the tracing as small as possible, in order to allow it to be used in any situation, without sacrificing the interactivity of the Squeak system.

To sum, our goals were, in that order of importance:

1. Familiar, intuitive user interface
2. Interactivity
3. Small overhead

3.1 Tracing in Squeak

The usefullness of tracing for understanding the dynamic flow of a program has been well established. There exist a variety of solutions for tracing in Squeak already.

MessageTally The MessageTally tool is built into the Squeak core. It allows to trace code execution and presents the program flow as a callgraph after the code has finished running, together with the time spent in each node.
MessageTally is specifically made to find performance bottlenecks in software and as such is only moderately useful for understanding the inner workings of a program or framework, as it provides only a textual call graph with the time spent in each message. Overall, MessageTally does not meet any of our initial requirements.

ContextS Tracing and logging is one of the classic motivations for context-oriented programming. ContextS allows to wrap methods and execute additionaly code before and after a message send, allowing to not only record execution time, but also message names, parameter- and return-values.
Due to the dynamic properties of contexts, ContextS is a very good canditate for building the drag and drop interface for tracing a particular, graphical object we had in mind. Recording different properties of the executed methods in before and after advice is easy, and providing life-output as well. Contexts have a fairly high execution overhead [9].

MethodWrappers The wrappers are the basis for ContextS. A MethodWrapper can be installed in place of a method, providing the ability to execute code around the actual method invocation, just as ContextS does. A single MethodWrapper is easily installed, and since MethodWrappers are just normal Objects (different from CompiledMethods, for example), we can subclass them to add additional behavior, which is why we based our evalution on this framework.

3.2 Usage Scenario in Morphic

We looked at different problems people might try to solve when working with Squeak. The focus has been primarily on programmers who are new to the Squeak environment and the graphic framework Morphic.

Students who are introduced to Squeak during their studies at HPI often wonder how to reproduce effects they see in the environment. One particular example of this was an attempt to reproduce the behavior of a code holder morph as seen in the code browser.

Code holders allow scrolling their content, highlight contained source code, and recompile the method into the system. Triggering those actions happens through mouse and/or keyboard shortcuts, so it's not immediately clear how to find out what is going on under the covers.

Usually, we tell students about the event system in Morphic, so in order to find out how a code holder scrolls its contents, they might start by looking

at the code for mouse events. If the look at the different implementations for the mouse event hooks through the code holder's inheritance chain, they learn how keyboard focus and selection works, but they will not find anything about scrolling. That is, because the Squeak VM translates scrollwheel input into the key combinations `Ctrl-Up` and `Ctrl-Down`, so the place to look at is in the keyboard event hooks, not the mouse events.

3.3 Motivating our Goals

The requirements set in section 3 were motivated with the above use case.

Intuitive Usage New users use the malleability and interactivity of the system to discover its features. They use the tools provided to get an intial idea of what is possible. We believe adding an easy to use tool for them to point at a particular feature will help them to understand the system quickly. Making the tool easy to apply by drawing on the familiar drag and drop interactions users know from other systems should lower the barrier to use the tool freely and frequently.

Livelyness of Observation The above example is very typical when building graphical systems. We questioned different Squeak/Smalltalk programmers and most of them said that they feel comfortable reading up on some frameworks API, but are often at a loss when it comes to event systems, oberservers and other implicit connections between some action and its result.

In real life, if the connection between an action and a result is unclear, we try to repeat that action and observe the surroundings of the system in question, to find hints as to what goes on when we interact with it [10]. Adding immediate feedback to our tool helps users to connect their actions with the execution of code in the system. When something is not immediately clear, the interaction can be repeated to form theories on what is going on specifically, and these can be verified simply by repetition and observation of the call log feedback.

Low Overhead In the example in 3.2 we expressed the desire to observe compilation and highlighting of code. Both can be fairly slow in the Squeak system, especially when looking at long (maybe generated) methods. We could just tell our users to try a different methods, but we felt there will be other use cases where similar performance considerations might be an issue. Additionally, if the tool we provide had enough overhead to make the system feel even slightly sluggish, users might refrain from using it freely.

4 Tracing Systems

Having set our goals and after deciding on MethodWrappers for the foundation of the tool, we looked at the specifics of tracing in the Squeak system.

4.1 Particularities of our Tracing approach

Naively tracing in Squeak with MethodWrappers is slow. Squeak methods are held in method dictionaries, which are shared among all instances of a class. Per default, when wrapping a method, all calls to this method will be routed through the wrapper. For oft-called methods, especially in the UI framework, this means that most of the environment might end up being wrapped, slowing the UI to a grinding halt.

Luckily for us, Squeak provides an easy to use API to pick a particular object and decouple its method dictionary from other instances of that class. This way, we only trace the object(s) the user is interested in, and not all objects in the system that have the same type.

How do we decide what to trace? The Morphic UI draws itself using a scene graph. The root of this graph is called the *World*. When a user drops our tracing behavior onto a graphical object, it can be difficult to decide which root object to trace from, and if we should trace into the scene graph. Additionally, some Morphs have associated *Models* and *MorphExtensions* to which they delegate behavior. How do we know when to trace those as well?

For simplicity, we opted to restrict ourselves to tracing the immediate object that received the drop event and all its submorphs. We realise that this heuristic might not be optimal. A discussion of this can be found further down in section 4.2.

Tracing in circles is dangerous. If we blindly trace connected objects, those connections might be circular and lead to multiple levels of method wrappers installed on the same object. To side-step this issue, we currently do not allow multiple traces on the same object. However, this also means that a user cannot trace the same object twice in different contexts. If, for example, someone started by tracing just the code holder in a class browser, and then wanted to trace the browser itself, the code holder would not show up in the call log for the class browser, unless the first observation is stopped.

4.2 Evaluation

Trace filtering To reach our goal of low overhead, we had to apply heuristics to filter our trace and dynamically unwrap methods.

Removing oft-called methods We assume that users are most likely to observe objects not to get a thorough picture of how they work, but to find a way to reproduce effects through code, i.e. the (perhaps implicit) API of an object is more important than how it is executed. API methods are usually called only once and in turn execute internal methods to achieve their purpose. With this assumption in mind, we unwrap methods that appear very often during a trace because they are unlikely to be the most simple, "API-like" entry point to the functionality. This has the added advantage that the greatest sources of slowdown are eliminated as well, because often called methods will not go through method wrappers most of the time of an observation.

Removing "private" calls Similar to the above, calls that are only ever made to *self* are not very likely to be meant for outside use. As in Squeak there are no private methods, it is customary to prefix the method categories for internal methods with the word "private" to signal that such methods should not be called from the outside. We use this idiom to trim down our call log further.

Trace selection While the scene graph under a particular morph often includes interesting enough morphs to find out about some functionality, we currently ignore any other important connections the morph might have. This should be made into an option for the user to decide.

To select the root of the traced scene graph, we believe that a selection policy similar to the morphic halos would work very well, as well as being in line with the rest of the system. This would make application of the tracer a two-step process (dropping the tracer and selecting into the morph), but the end user would know exactly what is being traced.

The first two heuristics work very well for our particular use-case, but might be innapropriate and surprising for many users. Because we want the tool to be intuitive, we believe that adding a small UI, like a "control-panel", when tracing might alleviate that surprise. In this control panel we can show these heuristics (thus telling the user about them when he uses the tool for the first time) and allow the user to (de-)activate them, if he is interested in private methods or methods that are being called more often. Additionally, we should allow the user to enter the parameters for these heuristics, e.g. how many calls per second is "often-called", what should the ignored prefixes be and so on.

One could also imagine opening the tracer heuristics up to the users in the spirit of open implementations. Exposing all the information we collect while tracing, the users themselves could add conditions under which methods could be unwrapped, or displayed more or less prominently, to ease visually associating methods with interactions.

5 Related Work

Scoped Method Tracing [11] implements a very similar tool for the Lively Kernel. The focus is less on understanding how to use a system, instead focusing on understanding how the system actually works. The other goals, however, a very similar to our approach, from low overhead to livelyness.

Macro Recording tools can be found in editors like Vim [12] and Emacs [13], the operating system Mac OS X ships with the Apple Automator [14], and the Microsoft Office suite [15] has macro recording abilities as well. While the secondary goals from section 3 are similar to ours, the primary goal is not in understanding how to reproduce an interaction in code, but simply in recording it to be able to replay it later.

Callgraph Analysis , while similar in its primary purpose, is inherently "post-mortem". The lack in livelyness however, allows for better filtering capabilites and more structured analysis. Instead of repeating interactions with different filtering strategies, users can record once and then filter multiple times with different heuristics, without having to redo the steps.

6 Conclusion

Without conducting a user story, we cannot say whether our approach is useful for a significant part of the possible users. We were able to put our prototype to use in different private and university projects, and were able to successfully solve the motivating use case as well as a few others. Judging from the feedback we *have* gathered, the fundamental ideas seem good for our purpose, and the ease of use is a big plus compared to the current tools that ship with the Squeak IDE.

The major remaining issue, seems to be that users have no way of knowing how the heuristics applied to the call trace work, or how to change them, a problem to which we have proposed a solution in section 4.2.

References

1. Eisenberg, M., Fischer, G.: Programmable design environments: Integrating end-user programming with domain-oriented assistance. In: Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence, ACM (1994) 431–437
2. Kanellos, M.: Pcs: more than 1 billion served (June 2002)
3. Lincke, J.: Eud lecture slides. Lecture slides during End-user development lecture at HPI, Summer 2011 (May 2010)
4. Miller, R.: End user programming for web users. In: End User Development Workshop, Conference on Human Factors in Computer Systems, Citeseer (2003)
5. Taivalsaari, A., Mikkonen, T., Ingalls, D., Palacz, K.: Web browser as an application platform: The lively kernel experience. Sun Microsystems Laboratories Technical Report SMLI-TR-2008-175 (2008)
6. Kay, A.: Squeak etoys, children & learning. online article **2006** (2005)
7. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of squeak, a practical smalltalk written in itself. In: ACM SIGPLAN Notices. Volume 32., ACM (1997) 318–326
8. Nyrhinen, F., Salminen, A., Mikkonen, T., Taivalsaari, A.: Lively mashups for mobile devices. In: Mobile Computing, Applications, and Services: First International ICST Conference, MobiCASE 2009, San Diego, CA, USA, October 26-29, 2009, Revised Selected Papers. Volume 35., Springer Pub Co (2010) 123
9. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A comparison of context-oriented programming languages. In: International Workshop on Context-Oriented Programming, ACM (2009) 6
10. Perscheid, M., Steinert, B., Hirschfeld, R., Geller, F., Haupt, M.: Immediacy through interactivity: Online analysis of run-time behavior. In: Reverse Engineering (WCRE), 2010 17th Working Conference on, IEEE 77–86

11. Lincke, J., Krahn, R., Hirschfeld, R.: Implementing scoped method tracing with contextjs. (2011)
12. Moolenaar, B.: Vim, the editor (July 2011)
13. FSF: GNU Emacs (July 2011)
14. Myer, T.: Apple Automator with AppleScript Bible. Volume 662. Wiley (2009)
15. Cornell, P.: Using macros to speed up your work (July 2011)