Using Squeak for Teaching User Interface Software

Mark Guzdial
College of Computing
Georgia Institute of Technology

Abstract

Squeak is a new programming language that is particularly appropriate for learning computer science. It offers an excellent infrastructure for interesting projects (e.g., multimedia, Web browsing and serving), and all source code is included (and written in Squeak) from the virtual machine, windowing, on up. Squeak is being used in a course on *Objects and Design* (focusing on the development of user interfaces), both to enhance the infrastructure for a course on, and to change how user interfaces are taught. Rather than teach a toolkit, the focus is now on teaching students *how to build* toolkits. This paper presents a pilot study suggesting benefits of our new approach.

1 Squeak's Beginnings

Squeak is a new programming language based on Smalltalk-80, but interestingly, skipping some 15 years of development [3]. Squeak is highly cross-platform, running on Windows, Macintosh, Linux, BeOS, and Windows CE devices (among others) bit-identically. It has been updated with modern features, such as web browsing and serving, 3-D graphics engine, and powerful sound synthesis. Squeak is an excellent pedagogical platform because it doesn't presume a windowing operating system. Instead, Squeak implements all of the windowing, multimedia, and other software in itself (including its own virtual machine), providing both a rich set of examples and a bare substrate on which one can explore and build user interfaces from scratch. When

We at Georgia Tech began teaching with Squeak in 1998 in our Sophomore, required course on object-oriented analysis and design emphasizing user interface design and implementation. We had been using ParcPlace VisualWorks, because we preferred the pure objects of Smalltalk to C++. But VisualWorks didn't run on all the platforms that our students were using (e.g., Linux) and was quite expensive, we were looking for an alternative. What we found in Squeak, however, was the opportunity to teach user interfaces in a new way. Because Squeak does not presume a windowing user interface, all the windowing software is written in Squeak. That means that it's possible to write over the windows (Figure 1) and even to construct one's own windows–or to program user-interfaces *without* windows. It's this "build from scratch" capability which has been used to change how we teach user interface software.

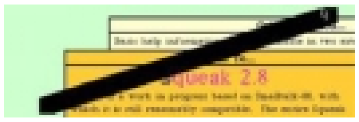

Figure 1: Drawing a line across window boundaries in Squeak

2 Squeak in Objects and Design

The course in which Squeak is used is an introduction to *Objects and Design*[1]. Students in this class have already had[2]:

- A one semester *Introduction to Computing* where some Scheme is taught.
- A one semester *Introduction to Object-Oriented Programming* in Java.

[1]`http://coweb.cc.gatech.edu/cs2340` is the class CoWeb, including lectures and example code

[2]This description focuses on the current state of the class under semesters. Previous to Fall 1999, Georgia Tech was under the quarter system, but a similar class was offered with similar pre-requisite classes

- A one semester course on *Languages and Translation* in C using tools like LEX and YACC to explore the issues of language implementation, from models of the bare processor up through tokenizing and parsing.

The goal of the *Objects and Design* course is to explore higher-level issues of design. The class is large: Typically 100 or more students a semester. Students are taught an object-oriented design process, which starts from analysis (with CRC Cards) and leads through design (using UML class diagrams). The course also serves as an introduction to the issues of user interface implementation and design. Modern user interfaces grew up with object-oriented programming (starting in Smalltalk), and using user interfaces to explore object-oriented concepts (e.g., aggregation, composition, inheritance, delegation) is natural and offers concrete examples. For example, how windows interact with their component objects allows for exploration of how messages get passed between peer objects, with the opportunity of visual experiments.

The course centers around a semester-long, team project. Because the class is using Squeak with its rich multimedia support, the team project often involves interesting manipulation of media. In one semester, students had to build personalized newspapers where stories were drawn from web-sites and laid out (multi-column with graphics). To encourage flexible designs, student teams are asked to serve their applications through more than one kind of interface. In the newspaper example, students had to be able to serve the newspaper via a Web interface (served from Squeak's internal webserver) and via PostScript (generated using Squeak's PostScript Canvas).

We[3] originally taught the class in VisualWorks, rather than C++, for the emphasis on pure object-oriented semantics. With the advent of Java, we have continued to use Squeak for several reasons:

- We don't have a standard language in our curriculum. More advanced classes are taught in C, C++, Java, and Lisp-based languages. The undergraduate curriculum at Georgia Tech is designed around an explicit decision to encourage our students to know multiple languages and paradigms.

- That said, the majority of our courses do use a C-based programming language. By requiring the use of Squeak in a project-based course, we make sure that our students have serious programming experience in something not C.

[3]"We" being the developers and teachers of the course, most notably Rich LeBlanc, Gregory Abowd, and Jon Shilling.

- Our experience with the course suggests that students can complete in a single semester more sophisticated and interesting projects in Squeak than in Java.

2.1 A Tiny Taste of Squeak

For those not familiar with Smalltalk, I offer a brief taste of the language. All computation in Squeak is triggered by message sends. Even an expression like `3 + 4` is semantically[4] "Send the SmallInteger 3 the message '+' with the argument SmallInteger 4." All control structures are message sends, with the body of the control structure in a *block* delimited by square brackets, e.g., `1 to: 10 do: [:i | sum := sum + anArray at: i]` which sums the first ten elements of the array `anArray`.

Squeak, like Smalltalk, late-binds all that it can. There are no explicit type declarations nor type checking. Determining the method for a particular message send is done at run-time.

Squeak is implemented as a bytecode compiler for a virtual machine. Squeak's virtual machine interpreter is actually implemented in Squeak, but the interpreter is compiled to C (and then to the native machine platform) for practical performance. The virtual machine for the PowerPC is about one megabyte, and its performance (on a 233Mhz G3) is nearly ten million bytecodes/second and 780K message sends/second. The performance is good enough to do sound synthesis, including MIDI, all in Squeak.

Squeak's development environment is defined within itself. The compiler, class browsers, debugger, and even higher-level development tools such as change sorters for collaborative programming are all implemented in Squeak. While the downside of this approach is that students' carefully tweaked Emacs environments can't easily be used for Squeak programming, the upside is that literally any aspect of the programming environment can be customized from within Squeak.

2.2 Using Squeak for Teaching UI Software

Since the course has moved to Squeak, the approach to learning and teaching user interfaces has changed, to good effect. One of the challenges of teaching user interface software is helping students to understand the *Model-View-Controller* (MVC) paradigm. Basically, MVC describes a mechanism for flexibly connecting user interfaces to underlying object structures.

- Models are the application objects, drawn from an analysis of the problem domain.

[4]In actual practice, the compiler short-circuits such common messages into direct calls.

- Views are the user interface objects, including buttons, text areas, and the like.
- Controllers are objects that mediate user interface events (like mouse moves and keystrokes) and transfer them to the appropriate objects.

MVC has always been difficult for students to understand. John Carroll and his colleagues identified this problem in their work teaching Smalltalk in the 80's [1]. The problem is the complexity of a loosely-coupled system. Students want models to directly control views (or vice-versa), but the MVC paradigm creates layers of indirection which allow models and views to be modified separately.

Our Early Experiences Teaching MVC The teachers of *Objects and Design* identified the problem of learning early on, when we first started teaching this course in 1993 under the quarter system. We began tracking performance, by using similar problems on midterm examinations and comparing the results across terms. For example, in the Winter 98 midterm exam, students were asked to design part of the objects for a phone system, and then asked in a separate problem:

> **Displaying the Incoming Phone Number**. The central office now gives you a new feature: You can ask a network connection for its phoneNumber. Now, its possible to have a View on the phone that displays the phone number for the incoming call.
>
> How would you change your Phone class (or others) to take advantage of this feature, assuming an MVC approach? You can assume that the Phone is subclassed off Model. Be sure to tell me (a) how the PhoneNumber View finds out about an incoming call (e.g., once the call arrives at the Phone instance, how does the View know that it needs the phone number?) and (b) how the View finds the number to display (e.g., what does the View do to get the phone number?).

The correct answer to this problem was to identify that the model broadcasts a *change* to all of its views, and the views query the model for the updated value (in this case, a phone number). Partial credit was provided for any part of this answer, e.g., that models broadcast to the views, but without stating that a "change" is broadcast (an important aspect of MVC, since the model should not presume to send the actual data to the views and thus create a constraint on what the views might want given the particular change in the model). This exact same problem was used in the Spring 1997 midterm exam.

An alternative form of the problem was used in the Summer 1997 and Spring 1998 midterm exams. After designing the objects for an alarm system, students were asked:

Table 1: Results of Traditional Instruction on MVC

Term (Number of Students)	Average	StdDev
Spring97 (86)	0.44	0.36
Summer97 (48)	0.54	0.40
Winter98 (107)	0.54	0.34

> **Alarm Status System**. You are now designing the alarm status monitoring systemthat is, an interface to watch over the alarms. You decide to use a Model-View-Controller structure for your system. You decide that you will have an Alarm class which will be your model, and an AlarmView for displaying the status. Someone else is building your controller – you dont worry about that.
>
> A. How does the Alarm object alert the AlarmView that it must update?
>
> B. How does the AlarmView get the alarm status to be displayed?

During the Spring 1997, Summer 1997, and Winter 1998, the class was taught using Coad and Nicola's *Object-oriented programming* (1993, Prentice-Hall) with VisualWorks. MVC was introduced using their examples and explanations. The case study around which MVC is introduced by Coad and Nicola is the construction of a user interface for a "Counting" device (a window displaying a count and increment, decrement, and reset buttons). The concepts of model, view, and controller are introduced, and they're pointed out as the user interface is created. In a following chapter, a vending machine is created. The vending machine's user interface is created using a set of classes on top of VisualWorks' UI classes, where the Coad and Nicola classes and methods are named to make explicit the model, view, and controller roles more explicit than VisualWorks' basic UI support. In both of these chapters, user interfaces are built on top of VisualWorks' existing abstractions.

The average and standard deviation for the MVC Midterm Problem in each of Spring 1997, Summer 1997, and Winter 1998 appears in Table 1.

A New Approach to Teaching MVC The Spring 1998 term, however, was the first one in which I tried teaching user interfaces in a new way. Rather than simply describe MVC with examples, I "built" MVC in class with live walkthroughs of code and live demonstrations of the results. I led students in lecture through three iterations of a user interface for an application already designed in class (a Clock). In all three iterations,

Table 2: Results of New Approach Midterm Problem on MVC

Term (Number of Students)	Average	StdDev
Spring98 (103)	0.86	0.27

I used no existing UI classes. Instead, "windows" and "buttons" were drawn (using the Logo-turtle-like `Pen` object in Squeak) directly on the `Display` object, and user interface events were read by polling the `Sensor` object.

- In the first iteration, the interface is built the most obvious way, by hacking the application objects to create a user interface. The students are introduced to the concept of an *event loop* to poll for events and dispatch them appropriately. But the event loop and all display objects were smack in the middle of the domain models, which defeats good object-oriented principles of appropriate responsibility.

- In the second iteration, *window* and *button* objects are created. Now, the event loop sits inside the window (ClockWindow) which dispatched the events into the buttons. The buttons sent messages to the model objects. But the updating of text feedback from the model (as described in the midterm problems) was still occurring within the models themselves.

- In the third iteration, the *changed-update* broadcast mechanism in Squeak is introduced which relates models and views by dependencies, not directly. Still without using any existing UI classes, we added a *text* view to our window and button classes, creating a tiny but relatively modern mini-UI system.

The results in the Spring 1998 midterm were markedly different on the Alarm version of the problem, as seen in Table 2. The exact same graders and exact same grading scheme were used between the Winter and Spring 1998 terms. A t-test comparing the two terms showed the difference to be significant $p < 4.4E - 13$.

This is not a rigorous experiment—not all relevant variables are being controlled. Students were not randomly assigned to the conditions. The Spring 1998 students may just have been better than the Winter 1998 students. Our intuition is that that's not true. We did compare other problems of a similar nature between the two terms, and found that on other problems, the average favored the Winter 1998 students. But it's still possible that the Spring 1998 students were more amenable to learning MVC. Nonetheless, the findings are promising and suggest that continuing with this approach makes sense.

Over the semesters the course has used variations of this approach. In one semester, students were to build programs in lab based on the classes developed in the iterative process described (e.g., ClockWindow). In another couple of semesters, students were offered the option of creating their own user interface toolkits as part of completing their team project. The toolkit-building students were given the opportunity to present their toolkits to the whole class. If any other peer student groups used one of the students' toolkits, the builders received extra credit.

It's worth noting that replicating the approach of building user interface toolkits from scratch is fairly hard to do in most other languages and operating systems. Most operating systems do not allow you to write directly to the screen without a lot of low-level hacking. While it's certainly possible to get a window from the operating system and write new windowing software within that window (which is essentially what Squeak does), the practical reality that all that code already exists in Squeak makes it appealing. Further, Squeak works on virtually all modern platforms, which means that individual windowing system differences don't enter into the problem as they might.

2.3 Using Squeak for Infrastructure and Case Study

My research is on computer-supported collaborative learning (CSCL), with an emphasis on communicating through multimedia, where Squeak is a natural platform. By using Squeak in the course as well, the class builds not only upon the developing experience of myself and my graduate students, but the class is also set up for an interesting use of Squeak as infrastructure. Students in *Objects and Design* use my lab's tools as part of the normal practice of the class, and then we use the tools as a case study to critique the tools' interface and object design.

One of the tools that we use in *Objects and Design* is the *CoWeb* (Collaborative Website), also known as *Swiki* since it's a Squeak interpretation of Ward Cunningham's WikiWiki Web[5]. The CoWeb is perhaps the simplest possible collaboration tool: Every page is editable by anyone (via an edit link on the page), and anyone can easily create new pages and links between pages (Figure 2). By typing `*A New Page*` on any page, a new page is created and linked in with the name `A New Page`. Surprisingly, such a structure does *not* lead to anarchy. Instead, it is now in use by some 120 groups at Georgia Tech, across ten servers, and is being adopted by other schools around the world[6].

[5]`http://w2.com/cgibin/Wiki`

[6]The CoWeb is released as an open source project. For more information, see `http://pbl.cc.gatech.edu/myswiki`. Research on the

GeorgiaTech
Squeakers
Welcome to the Georgia Tech Squeakers Front Page!
Links
Edit squeakers: Front Page

Figure 2: A CoWeb – normal view on left, and edit view on right

We use the CoWeb in several ways in *Objects and Design*.

- Students are asked to create pages for themselves with their own name (e.g., by typing `*Mark Guzdial*` on the *Who's Who* page). From then on, they can "sign" their postings with their name, and that name links to their page where they can introduce themselves.
- Each assignment has its own Q&A page. Such a persistent structure as conversation on a Web page has been shown to lead to more extended discussion [2].
- The class CoWeb also has a number of small activities, like the *Surprises* page where students are asked in their third or fourth week of class to leave notes to the next class about what they wish they had realized in the first week of class.
- The most popular activities in the CoWeb are the exam reviews. For each exam, a set of example problems is posted with a question/comment page linked to each problem. Students are welcome to ask for help on the problem, post solutions, or critique each other solutions. The pages are monitored by teachers and teaching-assistants, but the "correct" answers are never posted by them. Wrong answers are pointed out, but correct answers are met often with silence or "Yes, that would work, but there are other (perhaps better) solutions." This encourages students to plug away at a problem (not just memorize the first answer), yet provides useful feedback. In interviews and surveys (as well as the raw measure of participation rates), students identify this as one of the most useful activities in the CoWeb. In a sense, it uses CSCL to create a class-wide study group.

Since the CoWeb is in Squeak (built on a webserver also in Squeak), we are able to use it as a case study. Typically, the class disassembles the CoWeb before students have to build their own web interfaces. We critique the object design of the CoWeb and its user interface. For example, the original CoWeb was not well-designed for the variety of different looks-and-feels that users want in their CoWebs, so it serves as a point of discussion for how to create flexible object structures. Also, the interface of the CoWeb is based around HTML, which has not been welcoming to Mathematics classes whose central medium (equations) are difficult to express in HTML.

There is a definite synergy about critiquing a tool which the students themselves use, which they have complete access to, and which they can take apart and reuse in their own class projects. Students in *Object and Design* frequently start running their own CoWebs on their own computers. Some also participate in the Squeak and CoWeb open source communities, contributing features and bug fixes.

3 Conclusion

The benefits of Squeak in a user interface class are important but unusual. Squeak offers real toolkits and infrastructure for building interesting and complex projects. But at the same time, Squeak offers this infrastructure on top of lower levels of itself. Using the same language, students can build sophisticated high-level projects, or explore the lowest levels of the windowing code itself. Thus, what Squeak offers is an interesting combination of being able to build from the bottom up, yet build intriguing things, all within the same language and development environment. As the experience described here shows, there seems to be learning benefits to the "from scratch" approach of teaching user interfaces.

References

[1] Carroll, J. M., Singer, J., Bellamy, R., and Alpert, S. A view matcher for learning smalltalk. In *Proceedings of CHI'90: Human Factors in Computing Systems*, J. Chew and J. Whiteside, Eds., vol. Seattle, April 1-5. ACM Press, New York, 1990, pp. 431–437.

[2] Guzdial, M., and Turns, J. Effective discussion through a computer-mediated anchored forum. *Journal of the Learning Sciences*, To appear (2000).

[3] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*. ACM, Atlanta, GA, 1997, pp. 318–326.

CoWeb in higher-education classes including Computer Science is funded by the National Science Foundation and the Mellon Foundation.