Contents

1 Context of our Class .................................................. 2

2 Suggestions on Teaching Each Chapter .......................... 3
  2.1 Chapter 1 .................................................................... 3
  2.2 Chapter 2 .................................................................... 4
  2.3 Chapter 3 .................................................................... 5
  2.4 Chapter 4 .................................................................... 6
  2.5 Chapter 5 .................................................................... 7
  2.6 Chapter 6 .................................................................... 7
  2.7 Chapter 7 .................................................................... 9
  2.8 Chapter 8 .................................................................... 9
  2.9 Chapter 9 .................................................................... 9
  2.10 Chapter 10 ................................................................. 10
  2.11 Chapter 11 ................................................................. 11
  2.12 Extras Lectures: Parsing, UML, Design Patterns, Virtual Machines, and Optimization ................................. 11

3 Project Suggestions ....................................................... 12
  3.1 Used at Georgia Tech ................................................... 12
    3.1.1 Personalized Newspaper — Spring 2000 ................. 12
    3.1.2 Tree Milestone — Spring 2000 .............................. 15
    3.1.3 Math Equation Editor — Summer 2000 ................. 15
    3.1.4 Jukebox — Spring 2001 ....................................... 18
  3.2 Other Ideas ................................................................. 20

4 Answers to Exercises .................................................... 22
  4.1 Chapter 1 .................................................................... 22
  4.2 Chapter 2 .................................................................... 22
  4.3 Chapter 3 .................................................................... 27
1 Context of our Class

I wrote *Squeak: Object-Oriented Design with Multimedia Applications* for the class that I teach at Georgia Institute of Technology, *CS 2340, Objects and Design*. The course is a required Sophomore course for Computer Science majors at Georgia Tech. The focus is on learning object-oriented analysis, design, and programming, with an emphasis on 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 when we discovered Squeak and began teaching with it in January 1998.

The course in which we use Squeak is an introduction to *Objects and Design*. Students in this class have already had:

- A one semester *Introduction to Computing*
- A one semester *Introduction to Object-Oriented Programming* in Java.
- 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. We introduce 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. I tell the students that the lectures are mostly there to enable them to complete the project — I expect them to do most of their learning because of and through the project. Because we’re 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.

\[\text{http://coweb.cc.gatech.edu/cs2340/1}\]

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

2 Suggestions on Teaching Each Chapter

These are my notes on teaching each chapter: What the chapter is about, how I present the lectures, and any particular stories or anecdotes that might help to enliven the lecture.

2.1 Chapter 1

I typically present Chapter 1 in a single lecture. The point of Chapter 1 is to give students a sense of the historical context of objects and how they came to be. There is a striking contrast between the punch card machines fed by a priesthood of programmers and Alan Kay’s vision developed at the same time of personal computing. But the point is not to belittle the accomplishments of the early computer scientists. Alan Kay has pointed out that Computer Science has a rich history, but we tend to ignore it.

Sketchpad is an excellent example of that history that gets ignored. Sketchpad is an advance upon most of the object-oriented drawing programs of today, including Visio, MacDraw, and Illustrator. Sketchpad offered an enormous canvas, the powerful idea of master and instance drawings, and even the ability to create animations — features that virtually no drawing program has today, and this was almost 40 years ago.

In his history of Smalltalk for the HOPL-II conference, Alan tells an interesting story of how he met the ideas of Sketchpad and Simula (a programming language for discrete event simulations which included basic ideas of classes), the merger of which led to the invention of object-oriented programming. As a new graphics Ph.D. student at the University of Utah, he was handed Ivan Sutherland’s thesis on Sketchpad to read, and he was given the gruntwork of making Simula run on their version of Algol at Utah. He talks about digging in deep into the Simula — printing out the program, spreading it across the floor, and crawling over it. And from this deep exploration of Sketchpad and Simula, objects were born.

Alan’s vision for Flex includes Logo, which is only mentioned in the slides. Logo was developed by Wally Feurzeig and Danny Bobrow at BBN Labs as a programming language for children. Seymour Papert developed the ideas around Logo, as a place for students to learn about representations of knowledge and about their own knowledge by programming with Logo. Seymour presented these ideas most famously in his book Mindstorms. Alan wanted those kinds of ideas in his system, the Dynabook. Early Smalltalks actually looked like Logo,
and remnants of those implementations are still in Squeak today, such as the dots/colon syntax in Smalltalk and the Pen which is basically a Logo Turtle.

In his book *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*, Michael Hiltzik (Harper Business: 1999) talks about the first demonstration of overlapping windows by Dan Ingalls at Xerox PARC. Dan had invented BitBlt (Bit Block Transfers) which allowed for things like overlapping windows. In his demonstration, Dan popped up a menu, made a selection, and the menu disappeared, replaced by the material that was “under” the menu. For Dan (as it would be for us today), it was a casual gesture, but for the audience that had never seen a pop-up menu before, it was utterly shocking. Dan had to stop his demo and explain what they had just seen — it was just a minor application of the BitBlt that allowed for managing “damage areas” in overlapping windows, but it was a paradigm shift for the audience.

### 2.2 Chapter 2

Chapter 2 is an intense chapter. I found that students at Georgia Tech, when faced with a new programming language (such as Smalltalk), immediately wanted to know how to do all the things that they would normally expect of a programming language. I tried to talk about object-oriented design, but I kept getting pestered with questions about “How do I get a substring in Smalltalk?” or “How do I open a file?” I gave in, and started providing the material which has become Chapter 2 — essentially, I am trying to answer all the questions that students ask me right away. Typically, this chapter maps to two or three lectures.

It’s a firehose of material — I don’t actually expect students to learn all of the language from this brief introduction. But I do hope to answer the questions that they seem to be wondering about, and to give them some examples of what this is all about.

My strongest recommendation for this chapter is to do this lecture with a live Squeak image for all students to see. Actually try most of the examples in the chapter in class. I particularly recommend trying the collections methods: `select:`; `reject:`; `detect:`; and their brethren are very impressive for Smalltalk newcomers when they’re applied to both *Strings* and *Arrays*. This is a chapter where the BookMorphs version of the slides make a lot of sense: You’re already in Squeak as you’re presenting the slides.

The slides are split into five parts: (1) The basic rules for how Smalltalk works (presented in the context of a small piece of code that matches one’s intuition for what it would mean); (2) all those “normal language” things that students always ask me about; (3) how to do the “object” things in Squeak that you need to know (like *Blocks* and class definitions); (4) how to use Squeak’s user interface to run through an actual example; and (5) how to use Squeak’s tools to find things in Squeak.

Those last two points really cry out for live demonstrations. Letting students see the tools and how to use them has proven to be very useful. Many students (it seems) gloss over the text and graphic descriptions in the book, but to see
the teacher using the tools really seems to have an impact. Particularly show the Method Finder — many of the questions that students ask me in office hours are answerable from the Method Finder.

One of the common questions is “Can I use Smalltalk (or Squeak, specifically) to produce executables?” The last slide in the lecture answers that question. One answer is: Yes, you can merge the image and VM files to create a single executable file, which can end up being relatively small. The second answer is: If you think of the image as your OS, then “applications” are goodies that build on one another. Today, if you get a new application, you simply have a new monolithic component. If, however, your new application adds to the overall store of components, you’ve gained something. The third answer is: Think about cell phones. Are there applications in cell phones? This kind of embedded computing is something that Squeak is really strong at.

2.3 Chapter 3

Chapter 3 is the chapter that really starts to teach object-oriented thinking. It’s an extended example around Joe the Box — a curriculum originally devised by Adele Goldberg for Smalltalk-72! The slides go through the actual process in terms of Squeak’s tools, code, graphics methods, reading input from the user, object ideas (such as delegation), how to reuse an example in Squeak, and even iteration on the design and a brief glance at UML. This chapter is where the concepts of objects are really introduced.

Even though my students at Georgia Tech have already had a semester of programming in each of Java and C when I see them, I find that they get something out of this lecture. I relate it to Marvin Minsky’s claim that, “If you only know something in one way, you don’t know it at all.” Walking through Joe the Box does help students understand objects in a way that is probably quite different than they did previously.

Around slide 15, there are a set of slides on the Zen koan of Smalltalk, the Box class method:

```
new
↑ super new initialize
```

To understand this method is really to understand a lot of Smalltalk.

- Why is it super and not self? Because if it was self, it would be a very tight infinite loop — self new sending the message self new.

- Okay, so this method sends new up the class hierarchy until something actually creates a new object. But why is the new object a Box and not an Object or Behavior or whatever class actually ended up processing the new? Because the method that creates the new object always creates one of self, which remains bound to Box.

- Finally, what happens if NamedBox, a subclass of Box uses the exact same definition of class method new? Everything works fine — except that
initialize gets called twice. Trace it out and you’ll see it — I recommend tracing it out for the whole class, because it’s not obvious.

I recommend doing some of the Form examples live. Graphics is one of Squeak’s strengths, and it’s compelling and motivating to see how cool the examples are.

I pose the puzzle in class of what happens if you use NamedBox new instead of NamedBox named: ‘Jane’. I recommend tracing this example, to see why you get the effect that you do.

2.4 Chapter 4

Chapter 4 is the guts of object-oriented analysis and design. The example application is a Clock. The lecture slides iteratively design and re-design the Clock, through variations (like an Alarm Clock) and applications (such as a VCR).

Two forms of design notations are used in this chapter: CRC Cards and a subset of UML class diagrams. I find that students do class diagrams because I make them, but many of them actually do buy into CRC cards. I’ll see them discussing them with their group members in the back of class. I ask students to hand in their CRC cards for homework, and I love it when I get one wrinkled and dirty, as if it had been crumpled up and even stomped on. That was one well-argued object-oriented analysis!

Slide 41 comes after the complete OOA, OOD, and OOP cycle of the Clock, and just before the AlarmClock. The slide encourages the reader to try the AlarmClock design before going any further. I actually make the students do this. I stop the lecture and walk up the side aisleway of the large lecture hall. I bring the first row toward me, the second toward the other wall, the third toward me, etc. I make the groups actually come up with a class diagram. I wander around advising them. As groups finish, I hand them a transparency and ask them to transcribe onto that. Within 15 minutes or so, I have three real designs to put up, critique, and compare to the design that gets created in the rest of lecture. I find that this kind of activity encourages students to form opinions about the design. During the rest of the lecture, they’ll argue with me and tell me why their group’s design was better than the one I’m presenting. Oftentimes, I’ll agree with them — my goal is to have that discussion, not to present a perfect design.

There is a set of slides in the middle of this lecture (Slides 49-55) on how to work in groups using change sets in Squeak. I place this here because, in the order of our class, it’s about where students start to work in groups on their homework. I’m trying to provide them with the tools they need to make that effective. I encourage you to move these slides to wherever they make sense for your class.

The slides end with some critiques of the Clock designs, and some heuristics on good design.
2.5 Chapter 5

Chapter 5 is an introduction to user interfaces, that I usually spread across three lectures. Rather than start from users, which is where interface design should start, we start from code, which is where I think a programmer’s developmental understanding of interfaces starts from. The idea is to give students a concrete idea of how to build interfaces, and then introduce good UI design in Chapter 6 as a way of making choices among the interface options.

The first part of Chapter 5 is the creation and re-creation of a particular user interface for the clock from Chapter 4. The chapter code doesn’t presume any knowledge of user interface classes. Instead, it draws graphics right to the screen (using the same Pen graphics introduced in Joe the Box) and reads the keyboard directly.

We remake it three times:

- The first time is the worst possible engineering model: Simply slamming all the graphics and input code into the raw Clock code.
- Next, we break out the interaction objects (e.g., buttons).
- Finally, we break out the display object (e.g., a text area).

What we’re doing in these three steps is literally inventing Model-View-Controller (MVC). There’s a paper on the CD that presents empirical evidence that this works — students really do seem to learn MVC from this iteration.

I usually load up the code and run each of the iterations live, but it’s probably not critical to do that. The important part is to talk through the code examples, which is probably the longest code the students have seen so far. This can be a boring lecture, so live examples can help.

From there, the slides walk through the basic pluggable components in Squeak: Buttons, text areas, and lists. We re-do the Clock interface (a fourth time) in pluggable components in MVC.

Then, we switch gears into the newer UI system in Squeak: Morphic. We re-build the Clock interface a fifth time, this time in Morphic pluggable components. Then we walk through programming in raw Morphic components.

I do recommend doing the scripting interface example live in class (which is Slide 76), and show some of Morphic’s cool features. A great example is to take some object (maybe a RectangleMorph) and rotate and resize it. Then, keep a menu open, select it (to bring up the halos around the menu), and rotate and resize it! Now show that the menu items (at least some of them) still work! That’s a pretty compelling example of the power of an object-oriented user interface.

2.6 Chapter 6

Chapter 6 is a brief introduction to user interface design. I usually do this in a single lecture or two. The take home point is that there are choices to
make in building user interfaces, and that the choices have to be made from consideration of the users NOT by just rationalizing it.

There’s a great interactive activity that I use that isn’t in the slides. The point of the lecture is to prove to students that they can’t just sit down and make up a design, that they have to think about the user and that they are not always the best judge of what the user wants. This activity is designed to drive that home.

This is a true story that Alex Kirlik of Georgia Tech’s School of Industrial and Systems Engineering tells. He had a student who was hired by Macy’s to develop a sales kiosk for their women’s shoes department. What happens nowadays in Women’s Shoes? A woman walks up to a salesperson and says, “I want a shoe like this in size X” and hands the salesperson something, say, a white pump. The salesperson then disappears for 45 minutes, and returns saying, “I don’t have that, but I do have this” and hands the lady two or three boxes of shoes.

The question is: What is the organization in the back room that allows the salesperson to find two pairs of acceptable shoes that aren’t the exact match? What is the primary variable that organizes the back room such that a close match can be found?

I usually invite the students to suggest variables. I write down five to ten variables on the board: Style, size, color, manufacturer, etc. I then take a vote. Inevitably, size or style wins. In reality, Alex’s student found it was color. While it’s acceptable for the salesperson to come back with a half size off (or more), or a different style, the salesperson can’t come back with “I can’t find the white pump, but how about these red or brown shoes?” Thus, without asking the user, everyone in the class would have built a system that does worse than the current human salesperson.

Slide 17 in the lecture calls for interactive brainstorming: What are everyone’s favorite clock designs that deal with the failings of the clock interface (critiqued earlier in the lecture)? For example, one doesn’t have to deal with setting the clock at all if it get set automatically by radio or other mechanism. But then the challenge is making sure that error is impossible. A student of mine had a radio whose clock was set by the station he listened to at midnight. Unfortunately, he lived near a timeline and he liked a station across the timeline. If midnight rolled around and he was last tuned to that station, his car’s clock would be set an hour off — and there was no way to fix it until the next midnight when it would synch up again!

The rest of the lecture is about various design processes for interfaces and some evaluation mechanisms. The “stories from technical support” point on Slide 32 are all the old stories that everyone has heard, that are (surprisingly) actually true. I’ve seen users try to use their computer with the mouse upside down, rolling the little ball with their fingers. These technical support calls are actually a useful source of interface evaluation information.
2.7 Chapter 7

Chapter 7 is a collection of multimedia techniques in Squeak. My recommendation for using this chapter is to pick the pieces that:

- either (a) help students with whatever their homework/project assignment tasks are (e.g., if you’ve got them working on sound, go through the sound techniques).

- or (b) help to motivate the students to explore deeper. The 3-D facilities are particularly good for that.

Each of the media supports is presented in two ways: One as an end-user Morphic interface, and again, in terms of actual code that would allow you to use the media from within an application. Actually doing these in class make them much more exciting than just viewing the slides.

2.8 Chapter 8

Chapter 8 is the first of the four case study chapters. My suggestion for the case study chapters is to use them:

- (a) where they help with an assignment (just like Chapter 7);

- or (b) to make a particular point. Each case study has several potential points to make.

Chapter 8 is on the AudioNotes case study: A tool for recording and retrieving recorded notes. Here are some of the points that can be made from this chapter.

- AudioNotes doesn’t have a very good object design, so it’s a useful example to critique.

- The interface is fairly intuitive, but we use a cognitive walkthrough to show that there are confusing aspects of the interface. The case study works as a decent example of that interface evaluation technique.

- If you want students to work with recording and playing back sound, the AudioNotes case study serves as a useful example of how to do that.

- And finally, it demonstrates the use of the pluggable components in Morphic.

2.9 Chapter 9

Chapter 9 is on the built-in webserver in Squeak, the Pluggable Web Server (PWS), and on the collaboration tool provided, the Swiki (or CoWeb).

- If you want students to do any networking code, this chapter introduces much of Squeak’s networking support (both transmission and receiving).
• The chapter also describes how the Web works and how dynamic and interactive elements on the Web generally work.

• The PWS is used to provide examples for creating dynamic and interactive Web pages.

• The Swiki is an interesting object design: More complicated than AudioNotes, but with its own (more subtle) flaws.

Slides 29-31 are about **EmbeddedServerActions** that allow for HTML files containing code (templates), as one can do with many other languages. I usually tell a story here about one of the early Swiki releases and the bug hunt that we were led on. Swiki serves all of its pages through code-embedded templates. After one release, Swiki servers just refused to stay up for very long. Since it’s an open source project, I received bug reports from all over the world about this problem. It was a huge bug hunting effort. Somebody realized that the image was running out of file handles, which was a great clue, but we couldn’t find anywhere that we were opening a file without closing it. I searched the image for every file open and close, but without success. Then Bijan Parsia of U. North Carolina at Chapel-Hill noticed that file handles were disappearing after page *edits*. That was the key finding — we then figured out that a file was being opened but not closed *from inside the template*. Since it was outside the image, we couldn’t set breakpoints to find it and none of the code searching tools were useful there. The lesson is that embedded code is easy to write, but can be a pain to maintain.

2.10 Chapter 10

Chapter 10 is the case study on the Multimedia Authoring Toolkit (MAT) by Aibek Musaev.

• If you want students to work with text and the text editing classes in Squeak, MAT is a great example of manipulating these pieces.

• MAT also serves as an interesting but flawed object design. MAT has some interesting trade-offs in its design. It really only creates one class and modifies one other one! It doesn’t use subclassing because the changes that would been necessary would have been enormous due to very tight relationships between the text editing classes. But even without subclassing, some additional classes would make MAT easier to understand, such as, in its HTML generation.

• MAT is also a demonstration of creating drag-and-drop interfaces in Squeak.

• MAT had real users and real evaluation data is presented, so it serves as a case study of a small but valuable empirical evaluation effort.
2.11 Chapter 11

Chapter 11 is the case study around prototyping a play-writing workbench. I wrote the code for Chapter 11 explicitly for this book, because I wanted a case study that was about prototyping but still included interface evaluation.

- Chapter 11 is interesting for its use of an iterative design effort. I do an interface sketch and a simple CRC card analysis, then jump into coding, and I end up recoding part to add functionality.

- The actors in the play-writing workbench actually do speak and have heads with mouths that synchronize with the speech. In terms of cool multimedia, it’s fun.

- We use two different kinds of interface evaluation with the play-writing workbench: A brief expert evaluation and then a real trial with a real user (my son).

2.12 Extras Lectures: Parsing, UML, Design Patterns, Virtual Machines, and Optimization

The Extras folder on the class CD contains a few extra Powerpoint slide sets:

- The parsing.ppt slides introduce some ideas of tokenization and parsing. They can be used to describe T-Gen (the YACC-like tool for Squeak) as well as the built-in parsers for HTML and Squeak (the Squeak compiler is all in the image).

- The uml.ppt slides present some of the other UML diagrams not used in the book. Most of the images are directly copied out of the 1.3 Release reference — which is why the enormous copyright notice appears on the last slide of that file.

- The dp.ppt slides present a handful of design patterns. For each of the design patterns presented, examples are taken from the Alpert et al. book The Smalltalk Companion to the “Gang of Four” Design Patterns book.

- The vm.ppt slides introduce some of the ideas of virtual machines: Why you’d want them, how they’re constructed, how the Squeak VM is constructed, how it is extended (via the primitives mechanism), and how the Squeak and Java VMs differ.

- The optim.ppt slides introduce some of the techniques for benchmarking Squeak code (both in terms of time and in terms of numbers of message sends) and for speeding up Squeak code. This is a data structures lecture in disguise. The punch line of the lecture is on how choosing the right Collections can mean a lot in terms of code execution. There is also a discussion in this chapter on boxing-and-unboxing objects and why that slows down the execution of primitives.
3 Project Suggestions

These are ideas for projects that you might use in teaching this course. Several of the exercises in the book, especially in the latter classes, are sufficient for projects, too.

3.1 Used at Georgia Tech

These are projects we’ve used in our course in previous semesters. Typically, each milestone on the semester-long project takes about two weeks in a team of 3-4 students. Typically, the first milestone in a semester is an individual project, so that everyone does some programming, and the third is a design-only assignment. The last couple of assignments are not provided to the students until two weeks before, so that there is some design recovery involved. Example solutions to all of these projects can be found at http://cweb.cc.gatech.edu/cs2340/17.

3.1.1 Personalized Newspaper — Spring 2000

For Spring 2000, your project is to create a personalized newspaper that users can print.

There are two major parts to the project: Specification of news sources, and generation of the newspaper.

Specification of News Sources

Users are to be given the option of several Web-based news sources. These must include:

- [http://www.slashdot.org](http://www.slashdot.org) for Technology news.
- [http://espn.go.com](http://espn.go.com) for Sports Headlines and TV Listings (SEPARATE options – a user could ask for just Sports Headlines, or TV Listings, or neither).
- [http://www.cnn.com](http://www.cnn.com) for World, US, Business, Space, Health, Entertainment, Books, Travel, and Food news. (All of these SEPARATE options must be offered.)
- [http://www.georgia-navigator.com](http://www.georgia-navigator.com) for Traffic news. Obviously, this would change for other locale.

You may include other options if you would like. The user must be allowed to choose any combination of the above. The user must be able to save the choices...
so that a new personal newspaper can be generated daily without re-specifying the news sources.

The user must also be allowed to choose the primary news category for them:

- National or World News
- Sports
- Local News
- Weather

If a user selects NO news sources, at least two general news sources must be included in the paper.

**Generation of Personal Newspaper**

To generate the paper, you will visit the selected news sources, gather headlines, text, and graphics from those sites, and compose a personal newspaper. You will generate the newspaper as Postscript, named `newspaper.ps` so that the user can simply print it after generation. After a request for generation, the postscript must be generated within five minutes on a Pentium-100 class or better machine.

Each story that you include from the news sources must come with a title that you will include above the story and in a larger font size and boldfaced. If at least three headline stories are available for any news source, all three must be included in the paper (up to the newspaper size limit), but you need not gather news from more than three. You must also include the source URL for each story as a byline. You must attempt to access every requested news source, and your code must work for most days and for most cases for every available news source. Your newspaper must not be larger than 20 pages. You can cut-off stories, but only at a paragraph break (not mid-sentence or mid-word).

You must have at least three columns on the front page!. You may have more than three, and you may decide to group columns for a wider story on the front page. But at some point on the front page, a user should be able to identify three separate columns. You will probably find that you cannot fit most stories onto the first page (as real newspapers find). You should flow stories from the front page onto subsequent pages, with a continuation remark at the end of the page on the front page (e.g., "Story continued on page 3"). You may have any number of columns on subsequent pages. Not all pages need to start on the front page, or even be referenced from the front page. The headline stories from primary news category sources (e.g., if Sports is selected as the primary news category, CNN Sports and ESPN Sports are the primary news sources) MUST be on the front page.

You must include at least one graphic (e.g., photograph) on the front page, and it must come from one of the selected news sources. You can define your own rules for how you pick which graphic and how and where you place it on your page. There must be at least one other graphic on subsequent pages.

You can pick whatever fonts you want for generating your newspaper, but they must be readable by the user.
Hints and Warnings

- *Designing a multi-column newspaper is an NP-complete problem.* You cannot build a perfect solution that always works and completes in reasonable time. However, you can build a reasonable solution that always works. That’s your goal.

- Web pages change! If something dramatic happens, CNN, ESPN, and others modify their format. Sometimes servers go down. *You must still generate a newspaper!* Fill from other sources or categories. Create flexible parsing and formatting techniques.

- Flowing text and postscript generation are built-in to Squeak, as is Web access to text and graphics and HTML parsing. See Chapter 7 of the Guzdial text for details.

- We reserve the right to change the assignment definition at any point during the term! We will be more reasonable than a real client, in that we will make changes in time that they can be reflected in your project before the due date, but you can be sure that we *will* make changes. You should design flexibly so that you can cope with these changes. The changes won’t come out of the blue: They will be related to course lecture topics.

- Deal with the unusual cases. What happens if the user selects Sports as primary, but you can’t reach any Sports news services? What if there’s not enough content on a given day?

Addendum

Sometimes the user is away from their office computer, and still wants their newspaper. But they may not want all of the same news sources. You are to provide them with a web-based interface *served from their office computer* which will enable them to (a) remove some of their regular news sources from newspaper generation (no saving to the news sources file) and (b) generate a Web-based version of their newspaper.

When the user executes *Newspaper webstart*, the Web access should be enabled. *Newspaper webstop* should stop Web access. When the user visits http://mycomputers.ip.edu:8080/newspaper she should be served a page from which she can unselect news sources and generate a newspaper, or simply generate a newspaper given the standard news sources and primary news category. *You must use a Squeak-based webserver!* You’re encouraged to use PWS, since that’s what I’m lecturing on, but Comanche is okay, too.

The Web-generated newspaper does *NOT* have to have multiple columns. All the same stories must be presented, but they can be linear. At least one graphic must be included. Primary news category sources must be presented first.

http://comanche.swiki.net and http://minnow.cc.gatech.edu/swiki
3.1.2 Tree Milestone — Spring 2000

This is a mini-project used at the start of the newspaper project as a warm-up assignment.

Tree Milestone The first project milestone is an individual project to demonstrate individual competence with Squeak. You must create a Tree class in Squeak that can store any object at its nodes. You must be able to display the tree instance as a tree.

Example usage:

| myTree |
myTree := Tree new.
myTree content: 'I am the parent.'.'Put this string in info of root”
myTree addNode: Tree new. "Put a subtree beneath the root”
myTree first content: #(1 2 3). "Content of subtree root is array”
myTree addNode: Tree new. "Another subtree”
(myTree at: 2) content: 34.5. "Put float as content at root of second subtree.”
myTree display. “Display the tree graphically,
printString of each content at the node,
lines from roots to subtrees”

Display might look like:

‘I am the parent.’

![Diagram of tree with content and nodes]

Here is the basic interface that you must provide.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>content:</td>
<td>Set content for tree</td>
</tr>
<tr>
<td>addNode:</td>
<td>Add subtree at end of list</td>
</tr>
<tr>
<td>first</td>
<td>Return first subtree</td>
</tr>
<tr>
<td>last</td>
<td>Return last subtree</td>
</tr>
<tr>
<td>at:</td>
<td>Return subtree at given index</td>
</tr>
<tr>
<td>display</td>
<td>Display tree graphically on a white space starting at 00 extending to 300300</td>
</tr>
</tbody>
</table>

3.1.3 Math Equation Editor — Summer 2000

Your project is to make it possible to create well-formatted graphical equations.

Your users want to be able to quickly and easily assemble equations with:

• Variables and numbers. This includes multi-character variables like CO.
• Other characters that the user may want to type, e.g., [], ().

• Standard operators, e.g., + − /∗

• Superscripts (e.g., \(x^2\)) and subscripts (e.g., \(x_2\)). The superscripts and subscripts themselves won’t have subscripts and superscripts, but any equation could be in the superscript or subscript and subscripts/superscripts can appear anywhere (e.g., \(H_2O\) and \(x^{x^2}\)).

• Large fractions, e.g. \(\frac{\sin(x)}{x}\), and parentheses around large fractions, e.g., \((\frac{\sin(x)}{x})\). The numerator and denominator can include subscripts and superscripts.

• Greek letters

• Extensible so that roots, summations, derivative signs, and integral signs can be added to the system.

In the first milestone, each person will create a set of methods for constructing graphical representations of all of the above structures. Users will type in Squeak code to generate different structures, e.g., `MathEquation from: (MathFraction from: 'sin(x)' over: 'x')` display should generate the Form representation of \(\frac{\sin(x)}{x}\).

In the second milestone, your team will create a user interface for constructing equations. You will provide a palette of options that the user will select from in order to assemble an equation.

Next, your team will turn in a team plan for how you will assemble your whole system. This plan should include a complete object-oriented analysis and design, as well as a team plan with:

• A listing of who will be responsible for what

• A set of internal milestones for your team that will detail when things will get done in order to meet the final milestone.

In the third milestone, you will allow users to type in \(\LaTeX\) input to generate equations. You will provide an object that will accept input of the form:

\[ \frac{\sin(x)}{x} \]

and return \(\frac{\sin(x)}{x}\).

In the fourth milestone, you will deal with equations for the Web. One of the hardest things to represent on the Web are equations. You are to provide the user a Web page so that they can enter \(\LaTeX\)-style input and you will return a GIF of the formatted equation.

**Milestone P1: Basic Math Equations**

This is an INDIVIDUAL milestone - everyone must turn in a separate project.

You are to create a series of classes which will allow us to create a Form that is a mathematical equation.
• **MathEquation** will be the class that understands `from:` and will return an object that responds to `display` and `displayOn:`. We don’t care really about it being a FORM, it just has to be displayable. If it gets a string, it just returns a graphical representation of that string. However, if it gets a different mathematical object, it will return that object AS A DISPLAYABLE OBJECT.

• **MathFraction** has a class method `from:over:` that constructs a fractional object. MathEquation must format that properly.

• **MathOperation** has a class method `operator:with:and:` that allows math objects to be combined. Operators are just strings. The arguments to with:and: should be strings or math objects, e.g., `MathOperation operator: '+' with: (MathSuperscript from: '(x+2)' with: '2') and: '2'

• **GreekSymbol** has a class method `named:sized:` that takes a symbol and returns a math object of the appropriate symbol at the appropriate size (10, 12, 14, or 18, only). See `greek.pdf` for tables of Greek character names (which you should use for symbols, e.g., II and α) and their corresponding images.

• **MathSuperscript** and **MathSubscript** understand `from:with:` which creates a mathematical object of the from: argument either superscripted or subscripted with the with: argument.

To do this, you’ll need to filein ypatia.st from `ypatia.zip` and execute `Ypatia installAsTextStyle`. Using control-K, you’ll be able to choose Ypatia as your text style, to turn your Workspace into Greek.

To learn how to do this under program control, explore the classes `FontSet, StrikeFont, TextStyle, Paragraph, and DisplayText`

Hint:

```plaintext
('abcdefghijklmnopqrstuvwxyz' asParagraph
textStyle: (TextStyle named: 'Ypatia')) display
```

**Milestone P2: MathEquation UI**

You must construct a user-interface for assembling an equation. The user should be able to choose from base objects (e.g., basic strings, Greek letters, fractions, superscript/subscript) and be able to define where these pieces go (e.g., a string in the numerator of the fraction, an superscripted equation in the denominator). We strongly recommend some kind of a pallette (a floating window/morph with button options) that the user chooses from, but you can choose to do something else if it’s usable. It must be possible to generate a ImageMorph containing the Form of the end equation.

The user will start your system by executing `MathEquationUI start`.

**Milestone P3: Accepting LaTeX**

Rather than construct a bunch of classes, your users want to be able to specify equations via text input in the LaTeX format.
You will create a class `MathEquationLaTeX` which accepts a string via `from:`, and it will return a Form of the formatted equation. You should also implement `explore:` which accepts the same string input, but returns the MathEquation object after the parse, so that the internal objects are inspectable.

The kinds of LaTeX that you will accept includes:

- Normal letters, numbers, and most punctuation.
- Greek letters (the symbols preceded with `\`, e.g., \( \alpha \))
- Curly braces must be escaped with `\{`, e.g., \{ and \}.
- Backslash must be escaped itself, e.g., `\`.
- The other punctuation that must be escaped to display themselves are #, $, \%, \&, ; \_ \_ and :.
- Fractions are created with \( \frac{x}{y} \) where X and Y are the numerator and denominator respectively.
- Superscripts are created with `^`, as in \( x^2 \) to define x-squared.
- Subscripts are created with `_{s}`, as in \( x_i \) to define x-sub-i.

**Milestone P4: Equations on the Web**

The real hassle of handling equations these days is getting them up on the Web. You will create a Web interface to your equation tool, such that the user types in LaTeX and gets back an equation as a GIF. The user will start your system with `MathEquationWebUI start`. When the user goes to the URL `http://yourmachine:8080/mathequations/`, they should get a text area where they can enter a LaTeX specification of the equation. When they submit it, they should get back a GIF of the formatted equation.

### 3.1.4 Jukebox — Spring 2001

The goal for this project is to make it easier to play your collection of sound files (WAV, AIFF, MIDI, or MP3), at your computer or via a Web interface if you’re away. Your goals are:

- To easily create named groups of sound files
- To do searches to create new groups of sound files
- To create play lists defined from all or subsets of file groups
- To play play lists
- And to do all of the above from either your own computer or via a Web-based interface with your computer acting as a server
Playing WAV, AIFF, and MIDI files is documented in the textbook. See Playing MPEG Files in Squeak for information on playing MP3’s — MPEG support in Squeak is newer than the textbook.

**Milestone P1: Creating and Playing SoundFileGroups**

This is an INDIVIDUAL milestone - everyone must turn in a separate project.

Define a SoundFileGroup class that can respond to the following protocol:

- \( s := \text{SoundFileGroup} \text{match: } #('.mp3' '.midi') \text{ from: } '{e: \ soundfiles}' \) binds \( s \) to a SoundFileGroup of all files whose filenames contain the match substrings anywhere in the subdirectory structure starting with the from path.
- \( s \text{ printString} \) returns a String containing the full pathname of each of the files in the SoundFileGroup \( s \).
- \( s \text{ fileArray} \) returns an array of strings, each of which is the full pathname of each of the files in the SoundFileGroup \( s \).
- \( s \text{ playAndWait} \) plays each of the found files, and doesn’t return control to the user until all the found files are done.
- \( s \text{ startPlaying} \) starts playing the found files and returns access to the user.
- \( s \text{ stopPlaying} \) stops playing the found files that are playing in the background.

**Milestone P2: An Interface for SoundFileGroups**

You must now provide an interface for P1 (you can pick any team members’ P1 to extend, or you can start from scratch, or you can combine team members’ P1). From the command JukeBox \text{ start}, the user should be given an interface where she can:

- Choose a directory to start a search from
- Type some strings to serve as search patterns
- See the files resulting from the search
- Start playing the files
- Stop playing the files.

**Milestone P4: Web-based Interface**

Your team is now to implement all of the functionality of P2, but as a Web-based interface. After the user executes JukeBox \text{ playStart}, users should be able to access your computer as a server from Netscape or IE with \text{ http://yourIP:8080/jukebox }. They should be able to do all of the same things as P2, but from a web-based interface. You can use PWS or Comanche, but don’t assume that the TAs have
anything more than a raw 2.8 image (i.e., if you want Comanche, include it in your fileout).

**Milestone P5: PlayLists**

Your users don’t want to do searches every time that they want to play songs. Rather, they’d like to:

- Do a search and name the result as a PlayList
- Be able to do more searches and include the results into a PlayList
- Be able to selectively remove files from a PlayList
- Be able to save and load PlayLists

You are to provide the above functionality, for both your Squeak-based and Web-based interfaces.

**Milestone P6: The Web is Your Disk**

Your users no longer want to just have their own disk as their source. They want to do searches on sound websites like [http://www.mp3.com](http://www.mp3.com) and [http://www.listen.com](http://www.listen.com). Provide the user the ability to:

- Select a website from the ones that you provide (at least these two must be provided).
- Provide a field for the user to specify HOW MANY matches to return (to avoid returning thousands of responses to the search string ”mp3”).
- Do a search
- Add the files found into the PlayList
- Play the files from the PlayList

You are to provide the above functionality, for both your Squeak-based and Web-based interfaces.

**Milestone P7: Disk or Web is Invisible**

You must now provide the PlayList functionality (saving/loading, deleting files, everything) for files from the Disk or the Web. In other words, whether the files are coming from the server’s local disk or the Web should become invisible. In addition, add at least one other website that has MP3 or MIDI sound files on it.

You are to provide the above functionality, for both your Squeak-based and Web-based interfaces.

### 3.2 Other Ideas

These are ideas we’ve had for projects that we haven’t used yet.
• **Build Agentsheets or StageCast Creator**. AgentSheets and StageCase Creator are visual programming languages. Users specify visual rules which define IF-THEN rules: IF this particular graphical pattern is noted, THEN replace it with this other graphical pattern. It’s possible to create fairly complex simulations and video games with these kinds of programming languages.

An interesting project would be to support this kind of programming in Squeak. An early milestone would be to do the graphical matching. A later milestone would be to provide a user interface for specifying graphical rules. Other milestones would include:

– Being able to save rules to a file
– Allow for different grid sizes for the graphical rules
– Support different kinds of THEN conditions, such as playing sounds or setting variables.

• **JavaScript or SqueakScript**. One variation of this project is to implement a subset of JavaScript in the Squeak Scamper web browser. Another variation is to implement SqueakScript (like JavaScript, but based on Squeak syntax, and implemented in Java). Make it good enough that Netscape implements it...

• **Make tables work in Scamper**. And after that, use them to implement a spreadsheet.

• **Build Boxer**. Boxer is a fascinating programming language that combines a unique user interface metaphor with an end-user scriptable programming language. It’s been developed by Andrea diSessa at the University of California at Boxer and is described in his book *Changing Minds* (MIT Press: 2000). Boxer currently only runs on Macintoshes, and while a version is being developed for Windows, no version for Linux is planned. Creating a small version of Boxer in Squeak would both help to further these great ideas and would make them accessible on more platforms.

• **Build SqueakFlash**. SqueakFlash is meant to be a vector-based graphical tool, like Macromedia Flash, but it only plays through a Squeak plugin (see [http://www.squeakland.org](http://www.squeakland.org)). Could be much more powerful than existing Flash.

• **Build a Flash Editor**. Currently, we can’t build Flash animations in Squeak, but we can play them, and there is a start of classes that can write out Flash files. An interesting assignment would be to support definition of Flash Characters, then combine animation and sound, and finally write them out as .swf files. An interesting hitch would be to create a user interface for support of creating Flash presentations, like the Timeline used in Macromedia Flash.
4 Answers to Exercises

Below are answers to the exercises for Chapters 1 through 5. Chapter 6’s exercises are about having students go out and gather some evaluation data. Chapter 7 and beyond exercises are project ideas (as are some of the earlier chapters). The case study exercises, in particular, are pretty hard and can be multi-week projects.

4.1 Chapter 1

1. Objects as cells emphasizes the key notions of encapsulation and messaging. Encapsulation says that each object maintains its own data and contains its own behaviors. Messaging says that interactions between objects occur in well-defined mechanisms. Software as simulation emphasizes the key notion of an object-oriented program modeling the real world. Objects as little computers gets back to the messaging idea.

2. Inheritance isn’t really that critical of a feature. It does allow for extending the system easily, but in reality, most object-oriented software makes more use of aggregation (containing objects within objects) than inheritance. Cells do not really inherit from one another in any physical sense.

3. Biological cells do have kinds. Kidney cells are different than skin cells. Cells get manufactured, but not from one factory, so the basic notion of a class as an object factory fails. But a class as blueprint for a kind of object, defining its behavior and data, does exist metaphorically in biological cells.

4. Many possible solutions exist for this question. The main point is to argue from the key features in the chapter.

4.2 Chapter 2

1. whileTrue: is defined on BlockContexts. Integer addition is implemented in the class Integer.

2. The two kinds of statements that break the rule so far are assignment and return. aVariable := 3 is not a message send. This means that its not possible to override assignment. ↑returnValue is not a message send, either. In some sense, its the end of a message send. ↑ might be read as “give this back to the object who asked for me.”

3. (a) There are several ways of doing this. Here’s only one:

```
(aString := 'squeak'.
  nuString := "
  aString do: [:character |
    (character isVowel)
      ifTrue: [nuString := nuString,'-']
```

22
ifFalse: [nuString := nuString, character asString]].
^nuString

(b) Again, multiple ways, and here's one.

aSum := 0.
aCount := 0.
#(12 32 52 61) do:
[number | aSum := aSum + number.
       aCount := aCount + 1].
^((aSum / aCount) asFloat

4. Create FriendlyMuppet class as a subclass of Muppet, then simply create the method:

   greet
   ^'Well, howdy!'

5. Same as above, but with a different greeting (or greet) in each.

6. (A kinetic exercise.)

7. Surprisingly, its not Object. You can find it by using a System Browser to look at the class side of Object. You won't find new there. Now, choose spawn protocol. You'll find that new is actually implemented by class Behavior.

   How does that happen? Choose hierarchy to see the hierarchy of the class side of Object. The parent of the class side of Object is the class Class, whose parent is ClassDescription, whose parent is Behavior.

8. The below code works in Squeak 2.7, but not in Squeak 3.0. In file ch2-8.st.

   'From Squeak 2.5 of August 6, 1999 on 30 August 1999 at 2:02:03 pm'!

   !Array methodsFor: 'enumerating' stamp: 'mjg 8/30/1999 14:01'!
   switchOn: argument
       "Provide a case—like statement:
   #( 'a' [Transcript show: 'An a was input']
       'b' [Transcript show: 'A b was input'])
   switchOn: 'a'

   The tricky part is that a literal array returns the block as individual
string elements of the array.
"

| firstCouplet blockToEvaluate |
firstCouplet := self at: 1.
(argument = (firstCouplet at: 1))
  ifTrue: [
    "Now we have to translate the rest of the block"
    "We ask the compiler to evaluate the string formed by concatenating the rest"
    blockToEvaluate := Compiler evaluate:
      (firstCouplet copyFrom: 2 to: firstCouplet size).
    blockToEvaluate value.
  ]
ifFalse: [(self allButFirst) switchOn: argument]
!!

Squeak 3.0 redefines how symbols are printed. In file ch2-8-sq3.st. This version works:

**switchOn: argument**

"Provide a case−like statement:
#(('a' [Transcript show: 'An a was input'])
('b' [Transcript show: 'A b was input']))
switchOn: 'a'

The tricky part is that a literal array returns the block as individual string elements of the array.
"

| firstCouplet blockToEvaluate |
firstCouplet := self at: 1.
(argument = (firstCouplet at: 1))
  ifTrue: [
    "Now we have to translate the rest of the block"
    "We ask the compiler to evaluate the string formed by concatenating the rest"
    blockToEvaluate := Compiler evaluate:
      ((firstCouplet copyFrom: 2 to: firstCouplet size)
        inject:'' into:
        [:s: p | s , ''].
      ((p isKindOf: Symbol)
        ifTrue: [p asString]
        ifFalse: [p printString]])).
    blockToEvaluate value.
  ]
ifFalse: [(self allButFirst) switchOn: argument]

9. In file ch2-9.st

24
Object subclass: #Tree
  instanceVariableNames: 'info left right '
  classVariableNames: "
  poolDictionaries: 
  category: 'TreeProject'

isLeftTree
  "Is my left side a tree?"
  ↑(left isKindOf: Tree)

isRightTree
  "Is my right side a tree?"
  ↑(right isKindOf: Tree)

inorder
  "If leftside is a tree, traverse it.
  Traverse yourself.
  If rightside is a tree, traverse it."
  | result |
  result := OrderedCollection new.
  self isLeftTree ifTrue: [result := result, left inorder].
  result add: info.
  self isRightTree ifTrue: [result := result, right inorder].
  ↑result

info
  info isNil ifTrue: [↑] ifFalse: [↑info].

info: something
  info ← something.
10. It’s only a single line, in ScorePlayerMorph.

**makeControls**

```smalltalk
| b r reverbSwitch repeatSwitch |
b ← SimpleButtonMorph new
    target: self;
    borderColor: #raised;
    borderWidth: 2;
    color: color.
r ← AlignmentMorph newRow.
r color: b color; borderWidth: 0; inset: 0.
r hResizing: #shrinkWrap; vResizing: #shrinkWrap; extent: 5@5.
r addMorphBack: (b fullCopy label: 'x'; actionSelector: #delete). "<--- ADD THIS LINE"
r addMorphBack: (b fullCopy label: '<>'); actWhen: #buttonDown;
    actionSelector: #invokeMenu).
r addMorphBack: (b fullCopy label: 'Piano Roll'; actionSelector: #makePianoRoll).
r addMorphBack: (b fullCopy label: 'Rewind'; actionSelector: #rewind).
b target: scorePlayer.
r addMorphBack: (b fullCopy label: 'Play'; actionSelector: #resumePlaying).
r addMorphBack: (b fullCopy label: 'Pause'; actionSelector: #pause).
reverbSwitch ← SimpleSwitchMorph new
    offColor: color;
    onColor: (Color r: 1.0 g: 0.6 b: 0.6);
    borderWidth: 2;
```
label: 'Reverb Disable';
actionSelector: #disableReverb;
target: scorePlayer;
setSwitchState: SoundPlayer isReverbOn not.
r addMorphBack: reverbSwitch.
scorePlayer ifNotNil:
  [repeatSwitch ← SimpleSwitchMorph new
   offColor: color;
   onColor: (Color r: 1.0 g: 0.6 b: 0.6);
   borderWidth: 2;
   label: 'Repeat';
   actionSelector: #repeat;;
   target: scorePlayer;
   setSwitchState: scorePlayer repeat.
   r addMorphBack: repeatSwitch].
b target: self.

11. BlockContexts implement fork. FillInTheBlank>>request:initialAnswer: pops up a dialog box and returns the user's answer. Form>>fromFileNamed: will create an image from the specified filename.

4.3 Chapter 3

1. Yes, having super new initialize in two consecutive subclasses will work, however it is inefficient. initialize will actually get called twice. Recall that super new always returns an instance of the class that originally got sent the message. initialize will be sent to that same instance twice: Once in Box and once in NamedBox.

2. Having an object send new to itself inside a method named new is perhaps the tightest infinite loop available in Squeak.

3. Adding a Delay makes it easier to see:

30 timesRepeat: [jane turn: 12. joe turn: 10.
(Delay forSeconds: 0.5) wait].

4. In the first case, super new doesn't change who self is. Box new would return an instance of Box. In the case of draw and undraw, Box draw calls a class method draw, which doesn't exist.

5. This will be easier in Section 6 of Chapter 3, but the below method works. TextStyle default returns a bunch of things associated with the current text style, including an array of fonts.
draw
   | p |
   super draw.
   p ← Pen new.
   p turn: 90. "Always horizontal."
   p place: position. "Go to the position without drawing a line"
   p print: (name isNil ifTrue: ['Unnamed'] ifFalse: [name]) "Deal with nil names"
   withFont: (TextStyle default fontArray at: 1).
   "self drawNameColor: (Color black)."

6. If initialize does not draw the box, then an additional message send must be used to draw the box, e.g., joe := Box new. joe draw. Other than removing the draw message send, initialize remains the same.

4.4 Chapter 4

1. The design wouldn’t change if we went to an analog clock. That’s the point of MVC: The model stays the same.

2. You simply change the Ticker from seconds to milliseconds resolution, then change the Clock to respond to the ticker as a millisecond instead of a second.

3. This is a pretty easy problem – available as halfDayFormat1.1.cs.


continuousDayFormat
   displayFormat ← '24'
!!

halfDayFormat
   displayFormat ← '12'
!!

isContinuousDayFormat
   ⌈displayFormat = '24'
!!

isHalfDayFormat
   ⌈displayFormat = '12'

'From Squeak2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 11:52:24 am'

!Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 11:51'
isTicking
↑{(timer process isNil not) "If process isn't created yet, it's not ticking" and:
[(timer process suspendedContext isNil not)] "If the suspendedContext is nil, it's terminated"
!!

!Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 11:49'
start
timer isNil ifTrue: ["Lazy initialization -- should do this in initialize."
  timer ← SecondsTicker new.
  timer clock: self.].
self isTicking ifFalse: ["if already ticking, leave it alone"
  timer startTicking.]
!!

!Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 11:42'
stop
timer stopTicking.
!!

!SecondsTicker methodsFor: 'access' stamp: 'mjg 5/25/2001 11:40'
process
↑process
!!

5. See above


'From Squeak2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 12:02:52 pm'

!Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 12:02'
addendumPrint
(displayFormat = '12')
ifTrue: [(self hours > 12)
  ifTrue: ['"' pm'].
  (self hours < 12)
  ifTrue: ['"' am']
  ifFalse: ['"Exactly 12 must be printed as pm"
            '"' pm']]
ifFalse: ['"24—hour time is the default if no displayFormat is set"
          '"]'．

Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 11:57'!
display
  "Display the time in a given format"
  ↑ (self hoursPrint), ' ', (self minutesPrint), ' ', (self secondsPrint), (self addendumPrint)

Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 12:00'!
hoursPrint
  (displayFormat = '12') ifTrue:
    [(self hours) = 12 ifTrue: ['12'].
     (self hours) > 12 ifTrue: [↑(self hours − 12) printString].
     ↑self hours printString]
  ifFalse: [↑self hours printString].

Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 12:01'!
minutesPrint
  minutes ← time minutes printString.
  (minutes size < 2) ifTrue: [minutes ← '0', minutes]. "Must be two digits"
  ↑minutes

Clock methodsFor: 'time management' stamp: 'mjg 5/25/2001 12:01'!
secondsPrint
  seconds ← time seconds printString.
  (seconds size < 2) ifTrue: [seconds ← '0', seconds]. "Must be two digits"
  ↑seconds
7. This design is fairly weak for handling different times. The only way to do it, really, is to add the snooze time to the stop time – which then changes the stop time on the alarm which isn’t what you want to happen.

8. One good way to do it, which fixes several other design flaws, is to implement a queue of stop times.

9. If the one AlarmClock implemented a queue of stop times, then you wouldn’t need two AlarmClocks, and you’d only have a single SecondsTicker.

10. First, you’ll need a couple of scenarios. I suggest “I have an appointment on Tuesday at 11 am for lunch. I’ll add it” and “Today is Tuesday, and the appointment needs to alert the user.” I’ll leave the CRC Card analysis to the reader.

11. It’s pretty simple – just delegate the initialization. In file appointmentCreation.1.cs.

```csharp
'From Squeak2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 12:11:04 pm’!

makeAppointment: aDescription for: aDate at: aTime
  self description: aDescription.
  self date: (Date readFrom: (ReadStream on: aDate)).
  self alarm: aTime.

!!

makeAppointment: aDescription for: aDate at: aTime
| a |
  a ← Appointment new.
  a makeAppointment: aDescription for: aDate at: aTime.
  appointments add: a.
!!
```

12. This is a project idea. I recommend, though, looking at the PDAMorph that Dan Ingalls added in Squeak 3.0 — that project as the Calendar and Alarm Clock functionality in it.

13. It’s probably better to be more specific. better-on-off.1.cs.

```csharp
'From Squeak2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 12:14:05 pm’!
```

31
off
   alarm stop! !

offToday
   (appointments select: [:each | each date = Date today]) do: [:each | each off].

14. This is a single method added to Appointment.

    time
       ↑alarm alarmTime

15. Not setting the alarmBlock at the time of Appointment creation provides
    more flexibility for changing the alarm until the time the alarm is set.

4.5 Chapter 5

1. The below code (Clock-catchEvents.st) fixes the event loop to make
   buttons respond on mouseUp. It also stops the clock after the event loop
   ends.

'From Squeak2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 12:25:08 pm'!

catchEvents
   | hourPlus hourMinus minutePlus minuteMinus wentDown click |
   "Define the regions where we care about mouse clicks"
   hourPlus ← (position x) @ ((position y)+100) extent: 100@50.
   hourMinus ← ((position x)+100) @ ((position y)+100) extent: 100@50.
   minutePlus ← (position x) @ ((position y)+150) extent: 100@50.
   minuteMinus ← ((position x)+100) @ ((position y)+150) extent: 100@50.

   "Enter into an event loop"
   wentDown ← false.
   [Sensor yellowButtonPressed] whileFalse: "Yellow button press ends the clock"
      ["Give other processes a chance, and give user a chance to pick up."
       (Delay forMilliseconds: 500) wait.
       "self timeDisplay."

32
(Sensor redButtonPressed) ifTrue: [wentDown ← true].
((Sensor redButtonPressed not) and: [wentDown]) ifTrue: ["Now it's a release"
  wentDown ← false.
  click ← Sensor mousePoint.
  (hourPlus containsPoint: click) ifTrue: [self addHour].
  (hourMinus containsPoint: click) ifTrue: [self subtractHour].
  (minutePlus containsPoint: click) ifTrue: [self addMinute].
  (minuteMinus containsPoint: click) ifTrue: [self subtractMinute].]]
  self stop. "Stops the clock."
!!

2. See previous.

3. This is a nice project at this stage of the book, to rewrite the digital clock as an analog one.

4. If your students know Java Swing, then they do already know an MVC-based toolkit, through the Listener class.

5. Stopping the clock on close is easy, but it’s harder to explain why you do it how you do it. Below is the code. ClockWindow is officially the model for the SystemWindow, so it has to catch the windowIsClosing method and do something with it. But it must also know the clock so that it can find it and stop it. Thus, ClockWindow goes back to having a clock instance variable. (StopClockOnClose.cs).

'From Squeak2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 12:32:25 pm'!
Object subclass: #ClockWindow
  instanceVariableNames: 'position buttons clock'
  classVariableNames: "
  poolDictionaries: "
  category: 'ClockWorks'!

!ClockWindow methodsFor: 'window processing' stamp: 'mjg 5/25/2001 12:32'!
openAsMorph2
  | win component filler |

  "Create the clock"
  clock := Clock new.
  clock setTime: (Time now printString).
  clock start.

  "Create a window for it"
  win := SystemWindow labelled: 'Clock'.
  win model: self.
"Set up the text view and the various pieces"
filler := AlignmentMorph newRow.
filler centering: #bottomRight.
win addMorph: filler frame: (0@0 extent: 1@0.6).
filler addMorph: component.

component := PluggableButtonMorph new
    model: clock;
    action: #addHour;
    label: 'Hours +';
    borderWidth: 1.
win addMorph: component frame: (0@0.6 extent: 0.5@0.2).
component := PluggableButtonMorph new
    model: clock;
    action: #subtractHour;
    label: 'Hours −';
    borderWidth: 1.
win addMorph: component frame: (0.5@0.6 extent: 0.5@0.2).
component := PluggableButtonMorph new
    model: clock;
    action: #addMinute;
    label: 'Minutes +';
    borderWidth: 1.
win addMorph: component frame: (0@0.8 extent: 0.5@0.1).
component := PluggableButtonMorph new
    model: clock;
    action: #subtractMinute;
    label: 'Minutes −';
    borderWidth: 1.
win addMorph: component frame: (0.5@0.8 extent: 0.5@0.1).

component := PluggableButtonMorph new
    model: clock;
    action: #stop;
    label: 'STOP';
    borderWidth: 1.
win addMorph: component frame: (0@0.9 extent: 1@0.1).

win openInWorld.
↑win! ！

!ClockWindow methodsFor: 'window processing' stamp: 'mjg 5/25/2001 12:32'!
windowIsClosing
    clock stop.
    super windowIsClosing.
6. With only the smallest of changes (basically, references to `clock` become references to `self`), `openAsMorph2` copies to class `Clock` easily. (`Clock-openAsMorph2.st`)

'From Squeak 2.7 of 5 January 2000 [latest update: #1762] on 25 May 2001 at 12:38:44 pm'!

openAsMorph2

| win component filler |

"Use the clock, self"
self set time: (Time now printString).
self start.

"Create a window for it"
win := SystemWindow labelled: ‘Clock’.
win model: self.

"Set up the text view and the various pieces"
filler := AlignmentMorph newRow.
filler centering: #bottomRight.
win addMorph: filler frame: (0@0 extent: 1@0.6).
filler addMorph: component.

component := PluggableButtonMorph new
model: self;
action: #addHour;
label: 'Hours +';
borderWidth: 1.
win addMorph: component frame: (0@0.6 extent: 0.5@0.2).
component := PluggableButtonMorph new
model: self;
action: #subtractHour;
label: 'Hours −';
borderWidth: 1.
win addMorph: component frame: (0.5@0.6 extent: 0.5@0.2).
component := PluggableButtonMorph new
model: self;
action: #addMinute;
label: 'Minutes +';
borderWidth: 1.
7. This is a project idea – see PDAmorph in Squeak 3.0 to see how Dan Ingalls solved it.

8. This is a project idea – see PDAmorph in Squeak 3.0 to see how Dan Ingalls solved it.

9. Note that in the Morphic code version of this, the kick script does go to the falling object. See how it’s structured there.

10. I’ll leave this as a kinetic exercise that doesn’t map well to a textual book.

11. Set the gravitational constant to zero (add zero rather than 1 to the Ellipse’s velocity), then kick it, to set the velocity to zero.

12. Simply change the ‘+’ sign to a ‘-’ where gravity is added to the velocity.

13. Prototype-based objects are probably faster to implement a new idea with, but they may be harder to maintain because there are many more places where a piece of inherited behavior might be coming from.

14. See the answer to 11.

15. This is a project idea, to create a Cannon and add horizontal velocity.

16. Global gravitational constants is another great project.

17. The top of Squeak 3.0 has eyeballs which do just this.

18. I’ll leave this one as a project, too. The JoystickMorph is pretty easy to access – there are a variety of accessors. The GraphMorph is harder to pump data into, but if you look at the method loadSoundData: you’ll see a good example of how to do it.
19. Creating new kinds of SliderMorph is a great project.