# Introduction

The goal of this book is to help the reader to create multimedia projects in Squeak. The reader might be a student somewhere—that's probably the typical reader for this book, so "student" will get swapped for "reader" in many places. The reader might just be someone interested in Squeak.

In any case, there is an assumption that you're trying to do *projects*, serious efforts requiring pages of code. This book provides the information needed to get going with objects, user interface, and multimedia in Squeak. If you're working on a project, you are using Squeak, and you're actively trying to figure things out. This book gives you the tools to do that.

The structure of this book is aimed at the undergraduate computer science student, though the content is more generally on multimedia projects in Squeak. The below sections explain how the structure and content of the book was designed.

## 1 Approach of the Book

When American Universities were invented in the late 1800's, they were designed to be a mixture of the English College with its focus on undergraduate education and the German University with its focus on research. The goal was for the research to motivate, and even *inspire*, both students and faculty to be better learners and teachers. While this works in the best case, it has most often led to a higher priority on research than on teaching. (For a recent analysis of this tension, see Larry Cuban's *How the Scholar Trumped the Teacher*, Columbia Teacher's Press, 1999.)

The approach of this book is to be the reverse, hopefully, closer to the aims of the original inventors of the American University system. The pedagogy of this book is based on research in the learning sciences on how people learn. The content of this book is based on the research of me and my students developing collaborative multimedia in Squeak. The case studies in the latter half of the book are real projects that we designed, implemented, and then evaluated with real users to test the usability and effectiveness of our software.

While the case studies will probably not raise eyebrows, the structure and approach of the book may be uncomfortable to some. It may even seem intuitively wrong. Our intuitions are based on our experiences, in the classroom as well as in our daily life. But science is based on measurement. Science has come up with many ideas that seemed intuitively wrong, like disease being caused by small things too small to be seen by the naked eye, and that all objects fall at the same rate. As the methods of science have been applied to learning, similar non-intuitive lessons have been learned.

I attempted to apply the lessons about learning to the design of this book. I recognize, though, that research's lessons are not obvious—they require interpretation. My interpretations may be controversial, and even outright *wrong*. The responsibility for these interpretations is my own, not the original researchers.

## *1.1 Start from Where the Students Are*

There is a school of thought that says that students should be taught the abstractions necessary for proper execution in a domain before they are taught the actual execution. The argument is that the students' minds are then prepared to learn the "right" way to do things. This argument has been used to push for theory ahead of practice, design before implementation, and learning algorithms and development methodologies before actually doing any programming.

One of the unfortunate realities of our cognitive system is that we're very bad at transferring knowledge from one domain to another, even when they're tightly connected. We've known since the 1920's that students develop "brittle knowledge" (Alfred North Whitehead) that can be applied for a given exam or given course, but seems to disappear outside of the original class. The formal study of "brittle knowledge" arguably began in mathematics education research where students were found to become experts at one kind of equation, but adding in a single extra term totally confused them. The phenomenon was also noted in physics students, who could get A's on exams with tight explanations of acceleration and energy in a thrown ball, but who would explain outside of class that a ball falls because the atmosphere pushes on it. In my own research, I've been amazed to find that Engineering students seem to forget almost all of their Calculus when they get to the Junior and Senior years.

If students do not see the connections between areas of knowledge, then they won't transfer the knowledge. If they do not understand what they're learning, they can't see the connections. But if students do understand material, if they can see lots of relations between what they're learning and what they've known before, the knowledge is more likely to transfer and even be retained longer.

Case-based reasoning, one theory for how our cognitive systems work, has an explanation for all of this. As information comes into our mind, it becomes indexed. When a new event appears, we use our indices to figure out if we've ever seen anything like this before. If we do, then a connection is made. If our indices are developed well, we can match things as being similar. But if we learn things with indices that say "This is a boring fact for a course" as opposed to "This relates to design of programs," then we don't apply the information appropriately. It is possible to re-index things later, and it is possible to learn abstract things

with appropriate indices, but it's easier to learn new things as variations of known things, and then extend the indexing schemes.

The goal of connecting to what students already know is meeting the students where they are. While a student can memorize and even learn to reason with abstract material, this is a sign of the intellectual capabilities of the student, not the usefulness of the material. The real test is whether the students can *use* the knowledge later. The odds of having usable knowledge are improved if the material is presented when it makes sense to students and can easily be related to knowledge that the students already have.

Academics, researchers, and other smart people often disbelieve this point. They reflect on their own learning and note that they often *prefer* to get the abstractions first. There are several responses to this kind of reaction. First, self-introspection is not necessarily the best way to come up with lessons about learning. People do fool themselves. (For example, people often believe that shortcut keys are faster than menus, but all explicit measurements show that mousing over menus is always faster—see Bruce Tognazzini's *Tog on Interface* 1992 Addison-Wesley for a review of this research.) But perhaps the more significant response is that smart people are already smart. They've picked up all kinds of concrete and abstract knowledge already. They're excellent learners who know how to figure out connections to new knowledge. As my advisor, Elliot Soloway, likes to say, "20% of the people will learn whatever you do to them. It's the other 80% that you have to worry about."

For these reasons, the order of events in this book is concrete and easily understandable first, abstract and more general later.

- Squeak is first introduced as being like other languages that students might know, then the new and original features are introduced.
- Two chapters on Squeak programming appear before the chapter on design of object-oriented programs.
- Technical details of building user-interfaces are presented before interface design principles.

## *1.2 Learning involves Testing and Failure*

Noted cognitive scientist Roger Schank has promoted the importance of "failure-based learning." If you're always successful, you don't learn much. But after you've failed, you're in the perfect position to learn a lot. Someone who has just failed is now interested in reading, in exploring theory, and in engaging in inquiry in order to understand the failure and how to avoid it next time. In order to fail, you must face a "test" of some kind. The test doesn't have to be paper-and-pencil. It doesn't have to be formal at all. But if there isn't an event that tests your knowledge and

provides the *opportunity* to fail, then the failing and learning will never occur.

Computer scientists are well-familiar with this process of testing, failing, and learning from it. We even have invented a nice word for it: *debugging*. Debugging is such an important part of computer science that computer scientist and educator Gerald Sussman has been quoted as saying that "Programming is debugging a blank sheet of paper."

But when it comes to user interfaces, computer scientists tend to shy away from a real test. The buttons get pushed, and the menus get dragged, and success is declared. However, the *design* of the user interface is based on what the user wants, not whether the buttons can be pushed. To really test the *design*, one has to face the users. This is called *user interface evaluation*.

Students cannot learn design without evaluation. Otherwise, it's almost impossible for the design to fail, and thus learning cannot occur. The evaluation does not have to be all that sophisticated. The "discount usability methods" of Jakob Nielsen and others work because the real problems of user interface design tend to be right up front and pretty easy to see.

The chapter on user interface design includes sections on evaluation. But more importantly, each of the case studies in the back of the book includes an evaluation with real users. Some of the evaluations are survey-based, others are observational with interviews, and still others use recordings of user events to figure out what happened. In every case, there is *some* testing of the design assumptions.

## *1.3 Generation and Inquiry, not Transmission*

Even though people speak of "transmitting" or "delivering" material in a classroom, that isn't how learning works. Cognitive science has known for decades now that all real learning is *constructivist:* It's an active process of figuring things out and relating it to other knowledge. The goal of education is to motivate students to think about things, and thus, learn.

The goal of a book, then, is not to deliver facts, but to provide methods of generating knowledge and fuel for students' inquiry. A list of phrases to be memorized doesn't lead to learning. But if a book explains *how* to do something, and then provides some particularly interesting somethings to *do*, then the setting for learning is prepared.

There are sections of this book explicitly labeled "Generating" sections. These sections are meant to show how to dig into Squeak, how to study the exercises, how to build your own things in Squeak.

The rest of the book is meant to fuel inquiry, that is, students' exploration of things of interest. Inquiry can take lots of forms in lots of

different directions. The goal for this book is to provide starting places for many of these: From a historical or technical perspective; from object-oriented design or user interface design; from building objects to building interfaces.

## 2 Content of the Book

This is a book about using Squeak to build multimedia programs. It is not really a general book about object-oriented design or multimedia design. Rather, it's meant to be a book a particular programming language that has a rich history and is particularly well suited to use in computer science education. But in order to build multimedia programs in Squeak, students need to know:

- How to program in Squeak, from both a language and an environment perspective;
- How do design programs for Squeak;
- How to build user interfaces in Squeak;
- How to design and evaluate user interfaces in Squeak;
- How to do multimedia in Squeak;
- And lots of examples of how to do it.

In short, that's the content of the book.

The book is particularly aimed at the early-to-intermediate undergraduate student. There is an assumption here that students already know *some* programming language, but little else. The goal is to get students, as quickly as possible, producing multimedia applications in Squeak. It serves as an excellent lead-in to more advanced classes in any of these subject areas.

At Georgia Tech, we have been teaching a course on material like this for over five years. It has met with great success. This book is based on the notes for that course. I hope that this book will be as successful in other classes, and for you the reader, however you come to this book.