Squeak

Learn Programming with Robots

STÉPHANE DUCASSE

Apress®

Squeak: Learn Programming with Robots

Copyright © 2005 by Stéphane Ducasse

Lead Editor: Jonathan Hassell
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Assistant Publisher: Grace Wong
Project Manager: Kylie Johnston
Copy Manager: Nicole LeClerc
Copy Editor: David Kramer
Production Manager: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Diana Van Winkle, Van Winkle Design Group
Proofreader: Elizabeth Berry
Indexer: Valerie Perry
Artist: Diana Van Winkle, Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Library of Congress Cataloging-in-Publication Data

Ducasse, Stéphane.
Squeak : learn programming with robots / Stéphane Ducasse.
p. cm.
ISBN 1-59059-491-6
1. Robots--Programming. I. Title.

TJ211.45.D83 2005
629.8'925117--dc22
2005013248

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Contents at a Glance

Contents

4916_Ch00_FINAL 5/18/05 3:52 PM Page viii

Foreword

By Alan Kay
President, Viewpoints Research Institute, Inc. & Sr. Fellow,
The Hewlett-Packard Company

The Future of Programming As Seen from the 1960s

I started graduate school (at the University of Utah ARPA Project) in November 1966, and it is interesting to look back on the world of programming as I surveyed it at that time.

The amazing Jean Sammit (who was an inventor of programming languages and their first historian, as well as being the first woman president of the ACM) was able to count about 3,000 programming languages that were extant by the late 1960s. Much was going on, and some of it was of great import and interest.

Algol 60, as Tony Hoare pointed out, "was a great improvement, especially on its successors!" It had many surface virtues, including a stronger feeling for contexts and environments for meaning in a programming language, and one remarkable feature for its day—call by name—which allowed its programmers a range of expression very similar to the language designers themselves. For example, one could write procedures that would have the same meanings and actions as the control statements in the language:

```
for (i, 1, 10, print(a[i])
```

where the first and fourth parameters would be marked `name` and thus bundled into an expression that correctly remembered the hierarchical namespace context of its variables, but could be manipulated and executed from inside the body of the `for` procedure. Not even the original LISP did this correctly at first!

And there was a little-known syntactic variant in the Algol 60 official syntax that encouraged a more readable form for made-up procedures. This allowed a comma in a procedure call to be replaced by the following construct:

```
): <some comment> (
```

and this would allow the preceding call to be written as follows:

```
for (i): from (1): to (10): do ( print( a[i] ))
```

Do this with a nice display or IBM Executive typewriter made into a terminal (as JOSS had), and you would get

for `(i):` **from** `(1):` **to** `(10):` **do** `( print( a[i] ))`

which looks a lot like the Algol base language but done as a meta-extension by the programmer for the benefit of other programmers.

Perhaps the single most profound set of language ideas and representations happened earlier than Algol, but took much longer for most computer people to understand (and many never did), in part because of the different and difficult-to-read notation (for outsiders at least), and because many of LISP's greatest contributions were "really meta." One of the great contributions of LISP was its evaluator written in itself in a half page of code. This was a kind of "Maxwell's Equations" for programming, and it allowed many things to be thought about that were essentially unthinkable in more normal approaches.

LISP itself was driven into existence to be the programming system for an interactive commonsense agent—The Advice Taker—that could take the wishes of a human user given in normal vernacular and turn them into computer processes that would carry out those wishes. Some very interesting intermediate languages, such as FLIP, and attempts at doing some of the Advice Taker properties, such as PILOT, were created in the mid 1960s.

Sketchpad was perhaps the most radical of the early systems because it tried to leap all the way to a reasonable interactive framework for people who wanted to use the computer for what it was best suited: interactive simulations of all kinds. The three cosmic contributions of Sketchpad were

- The first usable approach to interactive computer graphics
- A real object structure for all of its entities
- A nonprocedural way to program in terms of the desired end results, where the system could employ various automatic problem-solving processes to come up with the desired results

This was helped tremendously by a "tolerance approach" to solving constraints, which instead of trying for perfect logic/symbolic solutions of the sets of constraints instead tried to solve the constraints within global tolerances. This approach allowed many important problems to be dealt with that are still difficult or intractable symbolically today.

JOSS was a very different cup of tea: it did "almost nothing" (basically numerical calculations using numbers and array structures), but what it did do was done perfectly and in the form of what is still one of the great user interface designs in history.

A Programming Language was the name of a book by Kenneth Iverson that took a highly mathematical approach to programming via functions and metafunctions expressed as a kind of algebra. In those days the language was called "Iverson." An actual system in which you could program a computer was still just an IBM rumor at the time, but many paper programs were written using these ideas. The best thing about Iverson was that it really paid off if you thought of it as mathematical transforms and relationships, and didn't worry about how many operations would be required. Not worrying about number of operations was almost unthinkable in those days of 1 MHz clocks on multimillion-dollar building-sized computers, so Iverson and LISP were both very liberating vehicles for thinking ahead to the future, when machines would be smaller physically, and larger and faster logically.

The Simula designers wanted to model large, complex dynamic structures and realized that Algol blocks would do the job if you could cut them loose from Algol's hierarchical control structure. In the creation of Simula I in the mid 1960s, they were able to see that their ideas had great relevance to the language and its programming, and when they did Simula 67 they could replace many formerly built-in data types, such as `string`, with a Simula 67 class.

The idea of extending the syntax, semantics, and pragmatics of programming languages constituted an entire genre of investigation in the mid-to-late 1960s. One of the reasons for this is that it had become abundantly clear that programming was going to be difficult to scale, and that scalability in most dimensions was going to be critical to the health of computing. Where complexity is a central issue, architecture dominates materials. This realization started to make programming appear as something different from math, and it started to reveal itself as a new form of engineering. There were calls for the formation of a discipline to be called "software engineering" and to have a conference to try to figure out what this might mean (how to cope if you can't just do math?).

ARPA Information Processing Techniques Office (IPTO) was in full swing by the time I went to graduate school in 1966, and it had already made some great starts toward its collective dream of having interactive computing for everyone pervasively connected via an "intergalactic network." Just how to create this network (which had huge scaling requirements) generated some of the best systems thinking of the time, and was an important part of my own thinking about the future of programming.

The ARPA funders were wise and did not turn the vision of their dream into funding goals, but instead tried to find and fund talents that had their own ideas about what the dream meant and how it could be done. This resulted in about 17 sites in universities and companies, most of which had come up with very interesting and different designs and demos. This constituted a community of both "agreement and argument" that made everyone in it much smarter than they were before they joined the great dream.

Of course, given Jean Sammit's 3,000 languages, there is much I haven't mentioned, and much interesting design that happened from 1967 to the end of the decade that has to be omitted here. To pick just five developments of particular relevance to the readers of this book, I would choose the conception of objects that I came up with, and how they were supposed to be useful to end users of personal computers; Carl Hewitt's PLANNER system, which was the most cohesive system for doing "programming as reasoning"; Ned Irons's IMP system, which represents perhaps the first really useful completely extensible language; and Dave Fisher's Control Definition Language, which illuminated extensibility in general and with respect to control structures in particular.

My background was in mathematics and molecular biology (I worked my way through school as a journeyman programmer) and in the arts. Circumstances forced me to try to understand Sketchpad, Simula, and the proposed ARPA intergalactic network in my first week in graduate school, and the reaction I had was cataclysmic. They were similar in some ways and very different in others, but they were different species of the same genus if one took both a biological and mathematical perspective. Biologically, they were "almost cells" crying out to be cells. Mathematically, they were "almost algebras" crying out to be algebras. So my initial fusion of these metaphors with computing was that you could make everything from entities that were logical computers that could send messages (which would also have to be logical computers). The logical computers would act the part of cells, and the protocols devised could be very algebraic—what today is (incorrectly) called *polymorphism*. This would result in great simplicity and scalability at the "materials level," and would open the door for advancements in simplicity and scalability at the "expression level" where the programmer lived.

Several years later I found Hewitt's PLANNER, and realized that it was the basis of a way to get programs to be both more meaningful and more scalable. (Many of the ideas of PLANNER also turned up in the later language called Prolog.) It was pretty clear that trying to send messages that were goal-oriented could greatly help scalability, in part because there are far more ways to try to satisfy goals than goals (think of sorting as a goal versus all the ways to sort), and this separation could have great benefits in keeping programs more meaningful and less about optimizations mixed in with the meanings.

Meanwhile, the extensible language IMP had appeared, and there were several clever ideas that allowed it to be practical and not just wallow in its own meta-ness.

And, in parallel to the thesis I was working on about personal computers and object-oriented systems for all levels of users, Dave Fisher was working on a very nice complementary set of ideas about how to make control structures extensible via being able to add new meanings dynamically to a LISP-style meta-interpreter.

LOGO, the first great programming language for children, was a happy combination of JOSS and LISP, by Papert, Feurzig, Bobrow, and others at BBN. This opened up the idea of children as very important end users of the powerful ideas of computing, and changed my idea of computing from a tool or vehicle to a *medium* of expression that had a similar cosmic destiny to that of the printing press.

These five systems and the invitation to help start up Xerox PARC were the impetus for Smalltalk, and are most noticeable in the first versions of Smalltalk.

Looking back from today, it is striking that

- The level of expression in today's programming is so low (really back around 1965 for most of it), and very few programmers today program even at the level of what was possible in LISP and/or Smalltalk in the 1970s.

- Smalltalk has not changed appreciably since it was released as Smalltalk-80 in the early 1980s, even though it contains its own metasystem and is thus very easy to improve.

- Moore's Law from 1965 turned out to be pretty much correct, and we can now build huge HW and SW systems, yet they are very fragile because the scalable concepts beyond simple objectness have not been added (i.e., we perhaps have cells, but no concept of even tissues, or how to build/grow multicelluar organisms).

- The Internet turned out to be a very successful expression of a radical approach to architecture and scaling, yet no software/programming system is set up to allow programmers to express Internet-like systems (what would the programs for the exemplary systems of Google and Amazon look like in such a new kind of programming system?).

What happened to progress in the last 25 years? And why is Squeak essentially just a free Smalltalk, if we desperately need progress?

In 1995 the Internet had gotten mature enough for us to try some experiments with media that we'd long wanted to do. And the Java (and other programming systems) of the time (and today) missed pretty badly in being flexible, meta, and portable enough to serve as a vehicle. Since we had done Smalltalk once before, and had written a book about how to do a complete such system, it made some sense to take a year to make a free, controllable Smalltalk and release it on the Internet (in fact, it took about nine months). The idea was that Squeak should not even be the vehicle so much as the factory for a much better twenty-first-century language.

However, programming systems in which programmers can program often take on a life of their own, and much of the Squeak open source movement and interest is in precisely a free Smalltalk with a media system that is highly portable. I think it is safe to say that most of the Squeak community is dedicated to making this Smalltalk more useful and accessible, and not devoted to making something so much better as to render Smalltalk obsolete (a fate I would dearly love to see happen).

So, I would like to encourage the readers of this excellent new book to not think of Smalltalk as a bunch of features from the vendor gods that must be adhered to, but as a system that is capable of great extension in all dimensions that will reward those who come up with better ways to program. At PARC we changed Smalltalk every few weeks, and in a major way every two years. Though it has hardly changed since then, please do and put those big changes out on the Internet for all of us to learn from and enjoy!

About the Author

STÉPHANE DUCASSE obtained his Ph.D. at the University of Nice-Sophia Antipolis and his habilitation at the University of Paris 6. He was recipient of the SNF 2002 Professeur Boursier Award. He is now Professor at the University of Berne and the Université of Savoie.

Stéphane's fields of interests are design of reflective systems, object-oriented languages design, composition of software components, design and implementation of applications, reengineering of object-oriented applications, and teaching novices. He is the main developer of the Moose reengineering environment. He loves programming in Smalltalk and is the president of the European Smalltalk User Group.

Stéphane has written several books in French and English: *La programmation: une approche fonctionnelle et recursive en Scheme* (Eyrolles 96), *Squeak* (Eyrolles 2001), and *Object-Oriented Reengineering Patterns* (MKP 2002).

If you want to discover why Stéphane is having fun with Squeak and actively participating in its development, check out `http://www.squeak.org/`. Check out `http://smallwiki.unibe.ch/botsinc/` for the web site of this book.

Acknowledgments

I would like to thank all of you who read parts and drafts of this book and provided feedback. It is not an easy task to read a work in progress, and I am grateful to all of you who made the effort. I will not attempt to list all your names here, because I am sure to forget some of you. However, I must mention Orla Greevy, Ian Prince, and Daniel Knierim, who read the entire manuscript. Thank you for your feedback and support. I would also like particularly to mention Daniel Villain, who read a draft of the French version.

I want to thank the Squeak community for the help they have provided me during the development of the environments used in this book, and for developing the amazing Squeak environment in the first place. In particular, I would like to thank Nathanael Schärli and Ned Konz for their help. I offer special thanks to all the developers who helped Smalltalk to escape from the clouds of dreamland and become a reality. I would also like to thank all the "Smalltalkers" who made this language and community so exciting. May you continue to make your dreams come true.

Writing this book has been a long and difficult process, because teaching novices is difficult. Moreover, I am not an easy person to live with, and as a researcher, I become excited by too many topics. I want to thank Didier Besset particularly for many fruitful discussions at the beginning of this project.

I also want to thank my wife, Florence, and my sons Quentin and Thibaut, two small boys who loved to run noisily around my desk when I was trying to concentrate on my work. Thank you for accepting a husband and father who was not always present, enthusiastic, and accessible. But soon we will be programming together.

Preface

Knowledge is only one part of understanding. Genuine understanding comes from hands-on experience.

—S. Papert

Goals and Audience

The goal of this book is to explain elementary programming concepts (such as loops, abstraction, composition, and conditionals) to novices of all ages. I believe that learning by experimenting and solving problems is central to human knowledge acquisition. Therefore, I have presented programming concepts through simple problems such as drawing golden rectangles or simulating animal behavior.

My ultimate goal is to teach you object-oriented programming, because this particular paradigm provides an excellent metaphor for teaching programming. However, object-oriented programming requires some more elementary notions of programming and abstraction. Therefore, I wrote this book to present these basic programming concepts in an elementary programming environment with the special perspective that this book is the first in a series of two books. Nevertheless, this book is completely self-contained and does not require you to read the next one. The second book introduces another small programming environment. It focuses on intermediate-level topics such as finding a path through a maze and drawing fractals. It also acts as a companion book for readers who want to know more and who want to adapt the environment of this book to their own needs. Finally, it introduces object-oriented programming.

The ideal reader I have in mind is an individual who wants to have fun programming. This person may be a teenager or an adult, a schoolteacher, or somebody teaching programming to children in some other organization. Such an individual does not have to be fluent in programming in *any* language.

The material of this book was originally developed for my wife, who is a physics and mathematics teacher in a French school where the students are between eleven and fifteen years old. In late 1998, my wife was asked to teach computing science, and she was dismayed by the lack of appropriate material. She started out teaching HTML, Word, and other topics, and she remained dissatisfied, since these approaches failed to promote a scientific attitude toward computing *science*. Her goal was to teach computer science as a process of attacking problems and finding solutions.

As a computer scientist, I was aware of work on the programming language Logo, and I particularly liked the idea of experimentation as a basis for learning. I was also aware that the programming language Smalltalk had been influenced by the ideas of Seymour Papert and those behind Logo, and that it had originated from research on teaching programming to children. Moreover, Smalltalk has a simple syntax that mimics natural language. At about that time, the Squeak environment had arrived at a mature state, and books started to become available in late 1999. But these were for experienced programmers, so I started and wrote the present book.

The environments that I use in this book and its companion book are fully functional. They have gone through several iterations of improvements based on the feedback that I have received from teachers. A guiding rule in our work has been to modify the Squeak environment as little as possible, for our goal is for readers to be able to extend the ideas presented in this book and develop new ones of their own.

Object-Oriented Structure and Vocabulary

The chapters of this book are relatively small, so that each chapter can be turned into a one- or two-hour lab session. I do not advocate presenting the material directly to children for self-instruction, but each chapter in fact has all the material for such an approach.

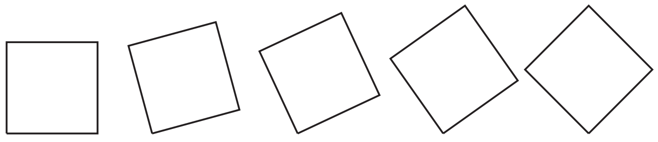
Although object-oriented programming is not developed in this book, I use its vocabulary. That is, we create objects from classes and send them messages. Object behavior is defined by methods. I made this choice because the metaphor offered by object-oriented programming is natural, and children have an intuitive understanding of the idea of objects and their behavior.

Those who are used to Logo may wonder why our robots do not have "pen up" and "pen down" methods, but instead "go" and "jump," where under the former, a robot moves leaving a trace, while the latter moves a robot forward without leaving a trace. I believe that the go and jump paradigm is better suited to the ideas of object-oriented programming and encapsulation of data than the traditional pen down and pen up design. An excellent analysis of these two approaches was made by Didier Besset, who collaborated with me on this project in its early stages.

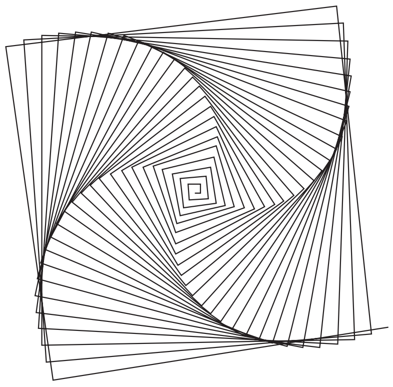
Organization

The book is divided into five parts, as described below.

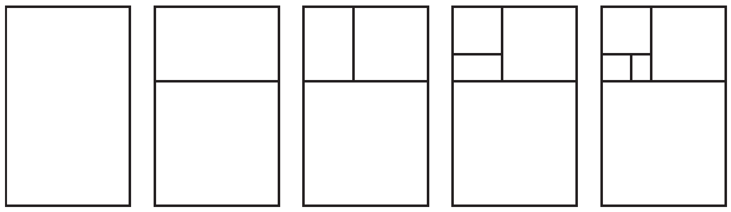
Getting Started. Part 1 shows how to get started with the Squeak environment. It explains the installation process and how to launch Squeak, and then presents robots and their behavior. A first simple program that draws some lines is presented.



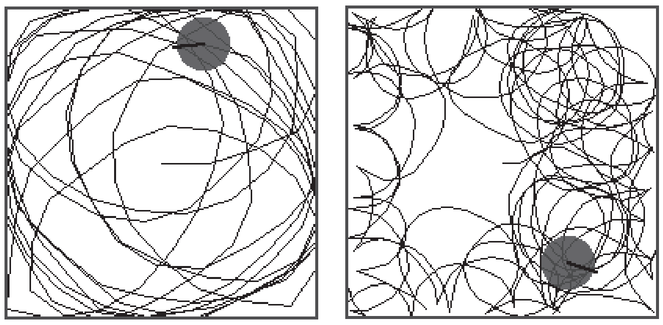
Elementary Programming Concepts. Part 2 introduces first programming concepts, such as loops and variables. It shows how messages sent to a robot are resolved.



Bringing Abstraction into Play. Part 3 introduces the necessity of abstraction, that is, methods or procedures that can be reused by different programs. The most difficult concept introduced is the idea of composing new methods from existing ones to solve more complex problems. Several nontrivial experiments are proposed, such as drawing golden rectangles. Techniques and tools for debugging programs are also introduced.



Conditionals. Part 4 introduces the notion of conditionals, conditional loops, and Boolean expressions, all of which are central to programming. This part also introduces the notion of references in a two-dimensional space and some other types of robot behavior. Finally, ways of using robots to simulate the behavior of simple animals are presented.



Other Squeak Worlds. Part 5 presents two entertaining programming environments that are available in Squeak: the eToy graphical scripting system and the 3D authoring environment Alice.

Why Squeak and Smalltalk?

You may be wondering why among the large number of programming languages available today I have chosen Smalltalk. Smalltalk and Squeak have been chosen for the following reasons:

- Smalltalk is a powerful language. You can build extremely complex systems within a language that is simple and uniform.
- Smalltalk was designed as a teaching language. It was influenced by Logo and LISP, and Smalltalk in turn heavily influenced languages such as Java and C#. However, those languages are much too complex for a first exposure to programming. They have lost the beauty of Smalltalk's simplicity.
- Smalltalk is dynamically typed, and this makes transparent a number of concerns related to types and type coercion that are tedious to explain and of little interest to the novice.
- With Smalltalk you need to learn only key, essential concepts, concepts that are to be found in all programming languages. Thus with Smalltalk I can focus on explaining the important concepts without having to deal with difficult or unattractive aspects of more complex languages.
- Squeak is a powerful multimedia environment, so after reading my books you will be able to build your own programs in a truly rich context.
- Squeak is available without charge and runs on all of today's principal computing platforms. And it should be easily portable to the platforms of the future.
- Squeak is popular. For example, in Spain, it is used in schools, where it runs on over 80,000 computers.