Introduction

This book is the book that I wanted when I started programming in Smalltalk. None of the books on the market seemed quite right; they all lacked some of what I needed to get my job done. Some were more like reference books, making sense if one already understood the material, but not very good if one lacked the basic knowledge. Others simply went over material that I could easily get by looking at the system classes using a Browser. The bottom line was that none of them seemed to provide answers to the questions I had: how do I use these capabilities when writing my application, how can I do the specific things I want to do, which is the best approach to take, how can I modify my development environment to make it more productive? In short, I wanted information that would steer me in the right direction and none of the books seemed to give that. This book is written for people who have similar questions and needs, who want to become productive in Smalltalk programming in as short a time as possible.

A lot of the material describes standard Smalltalk, but some material is specific to VisualWorks from ParcPlace-Digitalk. The book assumes that the reader is using VisualWorks 2.0 or VisualWorks 2.5, and where appropriate it explains the differences between these two releases. All the examples have been verified on both VisualWorks 2.0 and VisualWorks 2.5. Note that in most examples, to save space I have not shown the temporary variable declarations, so you will be prompted when you try to run or accept the code.

Approach

"Example is the school of mankind, and they will learn at no other". Like Edmund Burke, and like most people, I learn best from example. People generally find it difficult to take a theory and apply it to create a practical example. They usually find it far easier to take a concrete example, work with it, and extrapolate out the general behavior. Because of this, I've tried to include plenty of examples in the book, practical examples that the reader can use when developing applications. My goal has been to make this book different by focusing on practicality, on helping developers solve the real programming problems they face. The book has an attached diskette which contains the code for most of the examples.

This book presents the information that the reader needs in order to understand the concepts and the capabilities of Smalltalk. It then goes a step beyond and gives practical examples of how readers would use the capabilities when developing products. The book starts with the basic concepts of object-oriented development, specifically in the context of Smalltalk programming. It builds on this and talks about control structures, Collections, Streams, and other useful building blocks. Then it extends the basics and shows how the reader can use the Smalltalk classes when writing application programs. For example, Chapter 20, Error Handling, looks at the Signal and Exception mechanism that Smalltalk provides, then shows how you might use this mechanism when writing your application. Chapter 30, Testing, shows how you might test your application. Chapter, 33,

Managing Source Code, shows how you can track your changes, make sure that your image has the latest changes, and build a production image from the latest changes.

Layout of the book

The book is split into five sections: Basics, the Basic System Classes, Skills and Techniques, User Interface, and Advanced.

Section One, Basics, talks about the basics of Smalltalk. It talks about *objects* and *classes*, about *messages* and *methods*, and about variables and language syntax. It covers creating instances of classes and controlling program flow with the various conditional and looping structures. It ends with a chapter on thinking in terms of objects.

Section Two, the Basic System Classes, goes over some of the fundamental classes that you will use. Because everything in Smalltalk is an object, and all objects are instances of a class, we will talk about classes in every section. However, Section Two is more information oriented and less technique oriented than some of the other sections.

Section Three, Skills and Techniques, also talks a lot about the system classes, but covers techniques in using the classes. It shows how you work with Processes, how you can implement error handling code, and how to debug in Smalltalk. The section covers a lot of skills and techniques that you will use when developing your application.

Section Four, User Interface, focuses on user interface issues. It gives a lot of attention to the Model-View-Controller paradigm, then talks about how you can modify user interfaces at run-time. Since the user interface mechanisms differs widely between the different flavors of Smalltalk, this is the section most specific to VisualWorks.

Section Five, Advanced, is an advanced section, not because the topics are esoteric, but because the chapters tend to use advanced techniques or knowledge. In fact, some of the material in this section may be very useful, covering such topics as testing, adding methods to system classes, and customizing your development environment.

The book includes a comprehensive index. I've always been frustrated when I know a book contains the information I want, but I can't find it. So I hope this index proves to be less frustrating than some. However, some chapters, such as those on Collections, Streams, Object, and Meta-Programming, make passing references to a lot of messages, and I will not duplicate those references in the index.

Examples

The book contains many examples of Smalltalk code to illustrate points made in the chapters. There are two flavors of code: code that you can type into a workspace, highlight, and evaluate or inspect; and code that consists of classes and methods. The accompanying diskette contains all the code shown in the book. The two types of code are separated into two types of file. Files with an extension of `.ex` contain example code to be selected and evaluated. Files with an extension of of `.st` contain classes and methods that can be filed into the image. Appendix A, Source Files, describes which files relate to which chapters.

(When highlighting and selecting code in a Workspace, you will sometimes find that when you click to the right of the code, the following line is highlighted rather than the line to the left of where you click. By ending the statement with a period you can prevent this problem.)

Target audience

Based on a categorization of Smalltalk programmers into three categories — beginner, intermediate, and expert — this book is targeted at beginning to intermediate Smalltalk programmers. Expert programmers may get some useful new information from the book, but they will also know a great deal of what it contains. Beginning and intermediate programmers should find that the book answers a lot of questions about how to program in Smalltalk, how to use the system classes, and how to incorporate the Smalltalk mechanisms into application programs. It is a beginner book in the sense that it goes over the basic concepts, talking about classes, instances, messages and methods, then building on this to look at Collections, Streams, Processes, and so on. However, it doesn't spend as long on the basics as would a book geared totally toward beginners. My feeling is that it doesn't take too long to understand the basics, and I wanted this book to have greater use than just a few weeks. It is an intermediate book in that it gives suggestions on how the reader might write better applications, eliminate procedural code, implement more effective error handling mechanisms, and better test their software. It also contains many tips and hints that often take years to discover.

The book expects some amount of programming experience, whether in procedural or object-oriented programming. A complete novice will get some use from the book, but it will take more effort. While no significant knowledge of object-oriented programming or of Smalltalk is required, I do assume some basic knowledge of the VisualWorks tools, including how to use the Browsers to define classes and write code (when the book refers to a tool, it is referring to the VisualWorks functionality).

About Smalltalk

Smalltalk has been around a long time. It's origins date back to the early Seventies, when Alan Kay, then at Xerox PARC, wanted to create a language that children could program in. Kay's classic article, *The Early History of Smalltalk*, published in the ACM SIGPLAN Notices, Volume 28, No. 3, March 1993, talks about the evolution of early Smalltalk and about the influences of other languages on it.

Smalltalk was the original Object-Oriented (OO) language, and is still among the purest of OO languages. Unlike hybrid languages such as C++, Smalltalk forces you to think and program in OO terms. When it was originally developed it ran on its own hardware, and was basically the operating system for the hardware. Perhaps due to that heritage, it contains a lot of useful process control features that one would normally associate with an operating system, such as forked processes, semaphores, and mutual exclusion semaphores.

For most of its life, Smalltalk was associated with research and prototyping. However, it is now the fastest growing OO language. The advent of C++ has popularized and legitimized OO programming, but over time some people have become frustrated with the difficulties inherent in C++ development. Because of the new legitimacy of OO development and the difficulties of C++ development, a lot of attention has turned to Smalltalk. It has been further legitimized by IBM, which has made Smalltalk an important part of their development language strategy.

Smalltalk is often used for business applications, but it is also used for applications with a greater engineering orientation. Texas Instruments used Smalltalk to control an entire state-of-the-art wafer fabrication plant, adding extensions to allow distributed objects. Hewlett-Packard created Distributed Smalltalk, an extension to ParcPlace's VisualWorks that allows objects to communicate transparently over networks. Back in the Eighties, Tektronix began to use Smalltalk to run their logic analyzers and datascopes.

Features of Smalltalk

Smalltalk is a wonderful language to work with — in fact, it's hard to imagine a serious programming language being more fun than Smalltalk. Certainly, I've had more fun programming in Smalltalk than any other language I've worked with; so much fun that at times it's seemed incredible that I've also been paid to enjoy myself. If Smalltalk is so great, what makes it this way? I think it's a combination of things. In no particular order, here are some of those things.

Interpreted language

Smalltalk is conceptually an interpreted language. This means you can make changes to code and immediately execute them without having to go through a lengthy compile and link process. In fact, you can make code changes to an application that is actually running and have the changes take effect next time that piece of code is executed. You can also write short code segments (in any window) and execute them immediately[1].

Browsers

Smalltalk is a very well thought out environment, with a rich set of browsing tools. Because it was designed with children in mind, it is very easy to use. You can look at classes and their methods, and you can see just the local methods or also include inherited methods. From within a method you can ask to see all the methods that invoke it (the senders of the message), all the implementors of identically named methods (polymorphism), or you can browse the code of any of the methods this method invokes. From any of these displays you can modify code.

The Debugger

The debugger gives you a stack trace of where you are, allowing you to look at any of the methods in the stack and to inspect any of the variables. The inspection capability is excellent, allowing you to easily follow through objects and their own instance variables using just the mouse. As well as inspecting object values, you can also change them. The debugger also has the browsing capabilities of being able to look at senders and implementors of methods. Because of the interpreted nature of Smalltalk, you can evaluate code segments in the

[1] While Smalltalk is conceptually interpreted, the actual implementation is rather more sophisticated. When you *accept* a method, the compiler checks it for correctness, complains about syntax errors, gives you warnings about semantic errors, then compiles your text into byte codes. Having pre-compiled byte codes to interpret at run time is a lot faster than interpreting the text each time. On top of that, VisualWorks caches the machine instructions. When the byte code are interpreted, machine instructions are generated and executed. VisualWorks caches the machine instructions so that it doesn't need to interpret the byte codes next time it executes the method. Of course, as in any cache, infrequently used methods may be swapped out.

debugger and also change the methods that are stopped in. Being able to change a method then continue execution is one of the best features of the debugger. In fact, I often consciously program using the debugger. If I don't know exactly how to program a method, I'll simply ask the method to bring up a debugger by writing `self halt`. Then I'll write the method in the debugger, using it to inspect variables and evaluate code snippets as I write.

The Class Library

One of the big productivity gains comes from reusable software. Smalltalk provides a class library of around one thousand classes, all of which make life a lot easier and a lot more productive for programmers. (In a new VisualWorks 2.5 image with no additional tools or classes loaded, there are 960 classes and 23,131 methods. With the Advanced Tools loaded, the counts are 1,014 and 24,529). It's nice to be able to use an already existing class to manage a queue. It's nice to be able to convert it to a sorted collection with a single message send. Most of the classes are conceptually simple — they have a well defined role and perform it well — which allows you to easily combine and reuse objects.

The system class code is all there for you to peruse, which gives you three benefits. First, you can figure out exactly how to use it instead of having to rely on usually incomplete or hard-to-understand documentation. Second, you can learn new techniques from reading the system classes. Third, it makes it a lot easier to subclass from the system classes. For example, some applications using forked processes may find that a simple SharedQueue will suffice for passing objects between processes. Other applications may need a priority based shared queue, and having a visible class library makes this very easy.

The bottom line is that having an extensive class library means that you have a huge amount of high quality code available, which means you can put together applications very quickly.

The Garbage Collector

One of the features of Smalltalk that makes writing code much easier is the Garbage Collector. When objects are no longer needed, you don't have to explicitly destroy them as in most other object-oriented languages. Instead, they are automatically garbage collected when they are no longer referenced by another object. This garbage collection makes it possible to concentrate on the application problem, not worrying about what will happen to your objects. When they are no longer needed they will simply disappear and the memory will be reclaimed.

Object-Oriented Thinking

Smalltalk is one of the pure OO languages. Unlike C++, in which it's easy to write procedural code, Smalltalk makes it difficult to write procedural code. It usually takes about 8-12 months for experienced procedural programmers to become fairly automatic in their Object-Oriented (OO) thinking. To use a language that doesn't help that process makes it much easier to remain in a half-way schizophrenic state.

OO thinking is fun and it is different. One difference is that OO development tends to be more iterative than procedural development, more spiral. In OO development we acknowledge that we don't fully understand our objects and their interactions and we acknowledge that they will be changed and transformed as we increasingly understand the problem.

For several reasons, Smalltalk makes it very easy to progress through the spiral of iterative development. We've talked about how easy it is to write, test, and change code. This ease of writing makes it possible to put off real work until later, so a lot of methods will do nothing but ask themselves, or another object, to do something else. This allows you to think in conceptual terms rather than worrying about the details. Smalltalk is wonderful to work with because it helps enforce OO thinking and because it makes it very easy to do iterative, spiral development.

Modifying and Extending the Environment

One of the great things about Smalltalk is that if you don't like something, you can change it. If you think it should be possible to convert characters to strings by sending them the message `asString`, you can extend Smalltalk to add the new method to the Character class (for example, `^String with: self`). If you'd like different menu options available from your Launcher window or you'd like different keyboard mappings, it's easy to change the code. We'll see some examples of this in Chapter 31, Customizing your Environment.

Smalltalk is reflective

The Smalltalk system can reflect on itself — it can look at itself. Because all the system code is available, you can use the system classes to examine how the system classes work. You can write methods that use knowledge of how methods work. Because you have the ability to find out about the nature of classes and objects, you can write code that makes use of this information. Not all applications need this knowledge, but sometimes knowing how classes and objects are put together can help you create a better solution.

Summary

In the end, no amount of intellectual discussion can really give you a feeling for the power and beauty of Smalltalk. Feelings have to be experienced, not rationalized. So, with this book in hand, get programming!