Squeak For Non-Native Speakers

Noel Rappin

Chief Technical Architect

Echobridge

Introduction

Welcome to Squeak! If you are reading this book, it means that you are interested in learning more about the exciting range of applications and activities that are being built in Squeak. If you are reading this chapter, it means that you'd like a quick tour of the basics of using Squeak before learning about its internal implementation, or seeing how it can be applied as a web server, 3D graphics engine, sound synthesizer, handwriting recognizer, cheese grater, and so on.[1] As you'll see demonstrated in the later chapters of this book, it's very hard to start talking about all the features of Squeak without sounding like the host of a late-night infomercial (It's an object-oriented programming languages, and not one, but two interface packages, and a programming environment, and a web server! Now how much would you pay? Did we mention it's completely cross-platform?)

If you have never programmed in Squeak before, a short introduction is in order before you move on to the advanced topics. The goal of this chapter is to provide you with enough information about Squeak to understand the code samples and design principles in the later chapters, and allow you to experiment with it on your own – experimentation is a major part of the Squeak world view. You won't leave this chapter knowing every keyboard shortcut and message in Squeak – there's just too much. You will come out of this chapter with a place to start your own Squeak experience, and an understanding of the design principles behind Squeak, and how they differ from other languages you may be familiar with.

You are a member of the target audience for this chapter if you are an intermediate to advanced programmer, completely comfortable with basic concepts like: function, conditional, loop, variable, type. I assume that you've done at least enough object-oriented programming to be familiar with concepts such as class, instance, method, and inheritance. It will be helpful if at least some of that object-oriented experience is in C++ or Java

---

[1] Okay, just kidding about the cheese grater. All the other ones, though, are real.

– I'll be using those as comparisons. (previous Smalltalk experience is a plus, but not a requirement). It's entirely possible that even though you've done some OO programming, you've never really gotten what all the fuss was about. Hopefully we'll show you the reason here.

If that describes you, than this chapter should be at exactly the right place to help you jump to Squeak. If you've never done any object-oriented programming before, you might be better of starting with a general introduction to OOP before diving in here.

Feeling Right At Home

The first thing to show you in the tour of Squeak is how to get it and how to feel at home in the environment. Squeak's programming environment is a direct ancestor of the first Smalltalk programming environments written at Xerox PARC. While it is still in many ways more powerful than most programming environment created since, it also takes a little getting used to if you are more comfortable with say, CodeWarrior or Visual C++.

Getting, Having, Playing

The first step to using Squeak is to download it off the internet, and start it on your system. The initial implementation of Squeak was written for the MacOS. Within weeks of the first release, it was ported to Windows. Since then, Squeak has been ported to several varieties of UNIX, BeOS, DOS, a number of flavors of Windows CE, a Motorola chip running without an OS, and so on. The process of starting Squeak is broadly similar across platforms – any operating system quirks not discussed here should be available from the main Squeak site at http://www.squeak.org.

To download Squeak, first head to http://www.squeak.org and click on the link for downloading. In most cases, you will find an archive (ZIP, StuffIt, or tar) matching your OS of choice[2]. Download that archive in **binary** mode, and expand it into a directory on your machine. You must perform the download in binary mode, or some of the files may be corrupted by your download program reinterpreting the line endings incorrectly. The CD accompanying this book also contains a Squeak package for several operating systems, which includes the code discussed throughout this book.

---

[2] Some of the less common UNIX and other OS versions distribute VM source code, rather than binary. Compiling the VM is beyond this chapter, but the resources linked from Squeak.org should get you started.

The archive contains at least four files, three of which are identical no matter which platform you are running. Much of the cross-platform power of Squeak comes from the fact that over 90% of the data you download in your archive is identical no matter what platform you are using.

As of this writing, the current release of Squeak is 2.8, and your archive will contain the following files (file names may vary slightly from operating system to operating system, or from Squeak version to Squeak version).

- An image file: **squeak2.8.image**. The image file contains the entire binary state of your Squeak system. This includes the complete state of all objects known to the system, as well as all compiled code, and various global preferences that have been set. The image file can be thought of as a complete memory dump of the entire squeak system. In many respects, the image **is** the system. Images are completely compatible across platform – you can save an image in one platform and load it in another. You can also have more than one image saved under different names – if, for example, you have two different Squeak projects with different needs.

- A sources file: **squeakV2.sources**.   The sources file contains all of the text source code that was part of the Squeak system at the time of the Squeak 2.0 release. Don't change the name of this file – Squeak will look for it when it's time to display source code.

- A changes file: **squeak2.8.changes**. The changes file contains all the text source code that has been added or changed in Squeak since the 2.0 release. All the changes you make in Squeak by adding or modifying code will also be stored in the changes file. The changes file belongs to the image file of the same name – if you save your image under a new name, you will get a new changes file. The implication of this for you is that all the changes you make to your Squeak system can be viewed, undone, and shared.

- An executable virtual machine file. The exact name of this file changes from platform to platform. The VM file, which for most purposes you won't have to look at, does the background grunt work of converting platform independent Squeak code into platform dependent machine code. If this sounds familiar to you Java folks, well, where do you think they got the idea from? As compared to the Java VM, the Squeak one is much smaller (in fact, much of it is

written in Squeak code translated by Squeak to C) – which is one of the reasons why Squeak has been so easy to port.[3]

In addition to the above files, your Squeak package may include optional, platform dependent, plug-ins. These files provide support for other OS dependent features such as 3D graphics and sound.
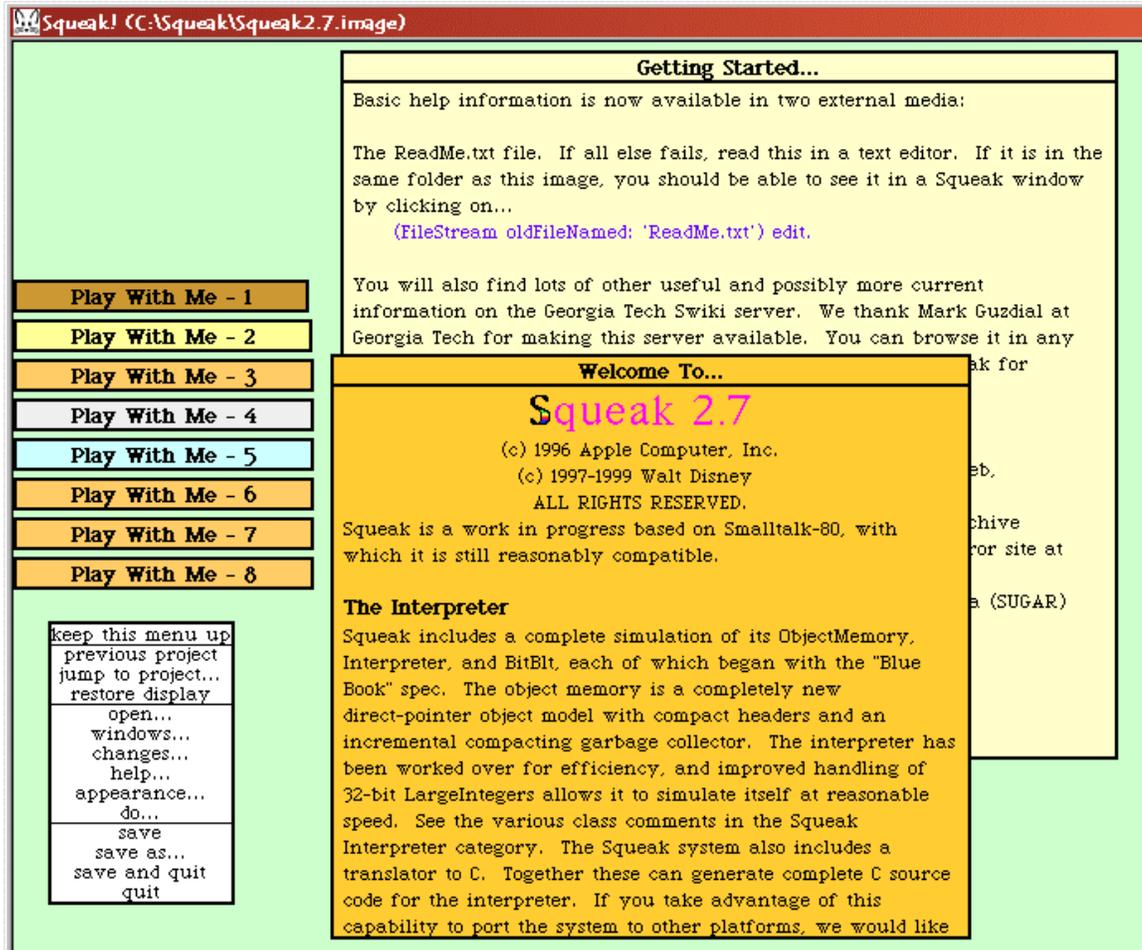
Once you've downloaded the files and dropped them all into a directory, starting up Squeak is a simple matter of running the VM executable with an image file as an argument. In the MacOS, simply double-clicking on an image file is enough. On Win32 machines, dragging the image file over the VM file works, as does using the Explorer to bind .image files to the VM executable file. On UNIX systems, the VM file can be invoked from the command line, with the image file as an argument.

Once you've launched Squeak in whatever OS, you will be treated to a main window in a particularly memorable shade of light green. There will be two open windows on top of that – one labeled "Welcome to Squeak 2.7" and the other labeled "Getting Started". Along the left side are a series of minimized windows labeled "Play With Me" numbered 1 through 8.

Figure 1: Squeak Welcome Screen

---

[3] Squeak's small VM increases flexibility at some cost in speed. Performance is increased by including native versions of the most commonly called Squeak code in the VM, called a *primitive*. If the native primitive fails, the system executes the Squeak version instead.

Now, having launched Squeak successfully, you're probably wondering how you actually do anything. Time for the next section…

Environmental Concerns

Squeak's interface and environment are a direct descendent of the original Smalltalk-80 system, itself one of the progenitors of all the window and mouse based interfaces you see on your desktop today. That said, Squeak is likely to seem slightly different from your standard Windows or Mac program – you'll immediately notice that Squeak doesn't use menu bars, for one thing. In fact, it's more useful to think of Squeak as it's own operating system piggy-backing on your existing system rather than merely a program running on whatever OS is controlling the rest of your desktop – many Squeakers stay within Squeak for nearly all their daily computing tasks.

The mouse and keyboard are still the basic modes of interaction with Squeak. The keyboard works pretty much the way keyboards work. As for the mouse, it will almost work the way you expect. Squeak assumes

you have a three-button mouse, regardless of how many buttons your mouse actually has. The left button is used for pointing and selecting. The right button brings up a context-specific menu for whatever window you are in, and the center button brings up a menu of window activities (close, resize, change label) for the currently active window. In practice, you'll rarely use the center button – most of its features are available separately in the title bar of the window. If you are stuck with a Windows two-button mouse, center clicks are simulated with alt-left click.[4] On a Mac, right clicks are represented with cmd-click, and center clicks are opt-click. Occasionally, you'll see references to the three buttons as "red, yellow, and blue", a reference to the names used for the mouse buttons on the original Xerox Alto, but those names are not common anymore. Table 1 has a quick-reference chart of the buttons and their common meanings.

Button:	Left	Center	Right
Also Known As:	Red	Yellow	Blue
Generally Used For:	Pointing, selecting	Window manipulation	Context sensitive menu
Windows 2-Button Equivalent:	Left Click	Alt-Left Click	Right Click
Mac 1-Button Equivalent:	Click	Opt-Click	Cmd-Click

Table 1: Squeak Mouse Buttons

Playing around with the Squeak opening window, you'll notice that clicking on either of the large windows makes the window active, and also causes a scroll-bar to be displayed to the left of the window. Clicking on the desktop brings up a menu of common system functions including saving and quitting, that I'll refer to as the World Menu (also known as the Desktop menu, depending on which GUI you are using, but World Menu is so much cooler sounding).

Clicking on the smaller windows on the side will make them active, and display a close box on their left, and a maximize box on the right. (Don't close them yet, we're not done with them – if you do accidentally

---

[4] If you do have a three-button mouse in Windows, you can get the third button to work by right-clicking on the Squeak window toolbar (or it's button in the system tray), going to VM Preferences, and selecting "Use 3 Button Mouse Mapping"

close them, click anywhere on the desktop not covered by a window, and select quit. When prompted to save before quitting, select no. Then restart Squeak – your image will appear just as before.[5]
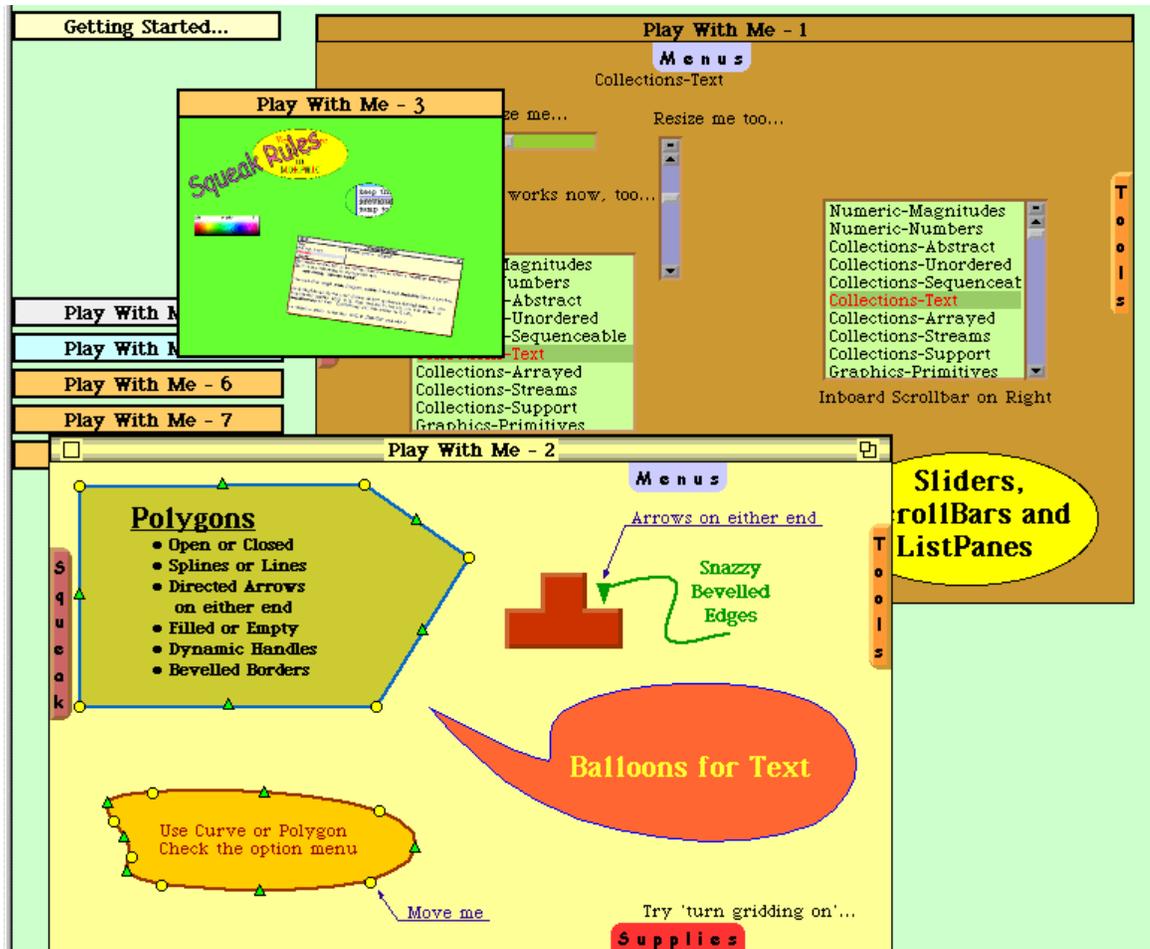

Figure 2: Some of the Play With Me Windows

Now, click on the window labeled "Play With Me 1", and maximize it. It turns out to display a window with some scroll bars and lists. Notice that clicking on this desktop causes a menu to display which looks slightly different than the World Menu on the background desktop. This is because Squeak actually has two completely distinct graphics/interface packages. The initial desktop uses MVC, which is based on Smalltalk-80's original Model, View, Controller framework. However, the Play With Me's and most of the newer Squeak interface work are in Morphic,

---

[5] Which is an important point that will be discussed later – ordinarily you do save your image when quitting, otherwise changes you make are not saved.

an interface package based on a system originally created for the Self language. From the user perspective of viewing and displaying code, the two packages are similar (Morphic is somewhat jazzier graphically and more fully featured), and most of the topics here will be applicable to both unless otherwise noted.

Now open "Play With Me 3". It opens into what looks like a thumbnail sketch of a Squeak desktop. It is, in fact, a thumbnail sketch of a Squeak desktop – a fact you can confirm by clicking on it, and choosing "enter project." Play With Me 3 is an example of a Squeak *project*. Projects are separate environments within squeak allow you to maintain a separate desktop and a separate set of code changes for each one. Being able to specify different screen preferences for different projects is nice (I usually give different projects different background colors, so I can tell what I'm working on at a glance). The real benefit to projects, however, is that the changes made to the image in each project are stored separately, allowing code in a project to be transferred more easily. We'll see more about this in a few pages when we discuss change sets.

You get out of a project by clicking on the desktop to get the World Menu, and selecting either "previous project", which takes you back to the project you just left, or "jump to project", which gives you a list of all projects in the system and lets you choose which one to go to.

Saving and quitting are also tasks that you'd like to do once in a while from your programming environment (even if the following chapters make a compelling case for never quitting Squeak, saving frequently is recommended). In Squeak, saving is an action you perform on the image as a whole – not on individual methods or classes of code. Individual methods or classes of code can be *filed out* as text, for storage or as a way to package software for distribution. New code methods are accepted into the image, and the image is then saved to disk. Storing the image to disk saves not only the code, but also the state of any global variables in the system. Not saving the image to disk means that any changes you've made in the image are gone (code changes are usually accessible in the .changes file, but must be reaccepted). To avoid this, you should get in the habit of saving the image frequently.

Given the above discussion, the bottom items on the World Menu – Save, Save As…, Save and Quit, and Quit – behave as you'd expect. Save As… allows you to create a clone of your image file. Quit will prompt you for a save before actually quitting. Windows users should take care not to exit Squeak by using the upper-right close box – that box will not give you the save prompt, rather, it will ask you if you want to quit Squeak *without* saving. Generally, you should resist this temptation.

Open Some Windows, Feel The Breeze

The Squeak user interface has a variety of specialized windows for code browsing, evaluation, and inspection. However, you'll likely spend the majority of your Squeak time in three windows, the System Browser, the Workspace, and the Transcript.

To start your exploration of these windows, access the World Menu from whatever project you happen to be in. Select "Open", and from the submenu, first select a Workspace, then select a Transcript. You are rewarded with two blank windows – one a pale yellow, and the other a burnt orange. It seems modest, but this is the beginning of Squeaking.

Select the Workspace (but make sure the Transcript is visible), and type the following (capitalization and parentheses are important):

Transcript show: (2 + 2) asString

With the cursor at the end of the line, right click the mouse, and select "do it" from the menu. If everything went correctly, you'll see a four show up in the Transcript window. If things don't go correctly you'll see an error window. For now, just close it and try again.

What's happening? The Workspace window is a text editor that allows you to evaluate Squeak code. The Transcript Window is a global space that is always available, and is used for returning values and for debugging purposes. The "do it" command sends text to the Squeak interpreter for evaluation (either the line the cursor is on, or a multi-line group of selected text). This particular line of code is adding $2 + 2$, converting the result to a string, and asking the Transcript to display the result (4, I hope).

Actually, you don't need the Transcript to show results. On a fresh workspace line, type the following:

2 squared.

Again, right-click with the cursor at the end of that line, but this time select "print it." The result (who didn't get 4 again?) will appear in the workspace. The "print it" command causes the code to be sent to the interpreter, and then the result returned by that code is written to the window at the end of the selection. Notice that the result is selected, so that it can be easily overwritten by the next words you type.
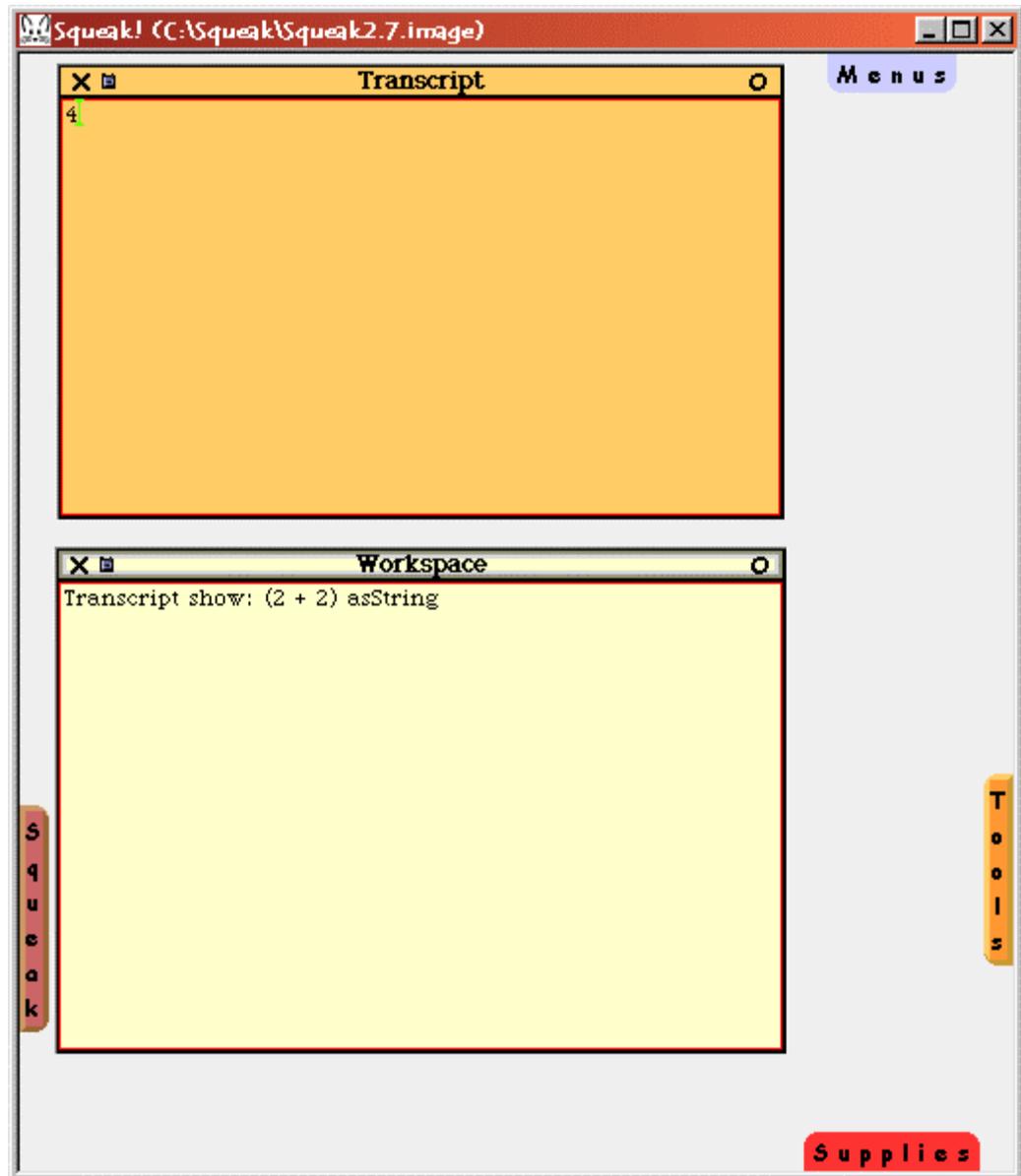
Squeak For Non-Native Speakers

Figure 3: Transcript and Workspace

Go to another fresh line and type:

'abc' reversed.

Be sure and use single quotes, and not double quotes. This time, right-click and select 'inspect it'. A new window opens, titled "String", with "self, all inst vars, 1, 2, 3" along the left side. "Inspect it" causes Squeak to evaluate code and open an *inspector window* on the result. Inspector windows show the values of all the component variables of an object, and can be opened on any Squeak object. In this case, "self" shows the object as a whole, in this case, the string 'cba', while the "1,2,3"

listings show the ASCII values for the individual characters that make up the string. Inspector windows are very helpful in debugging large and complex Squeak objects.
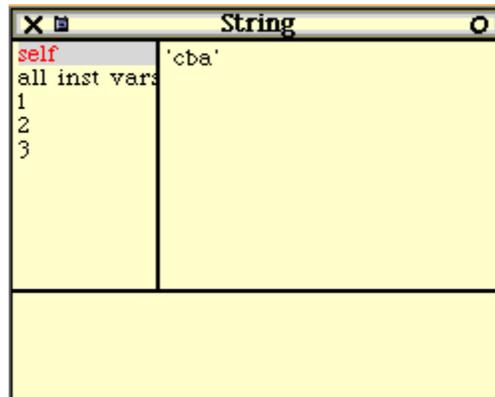

Figure 4: An Inspector Window

Now that you've seen individual lines of code, it's time to take a peek at the mother lode. Go to the World Menu, select "open", and then select "browser". A window will open, titled "System Browser". It has four panes across the top half, and a single pane taking up the bottom half.
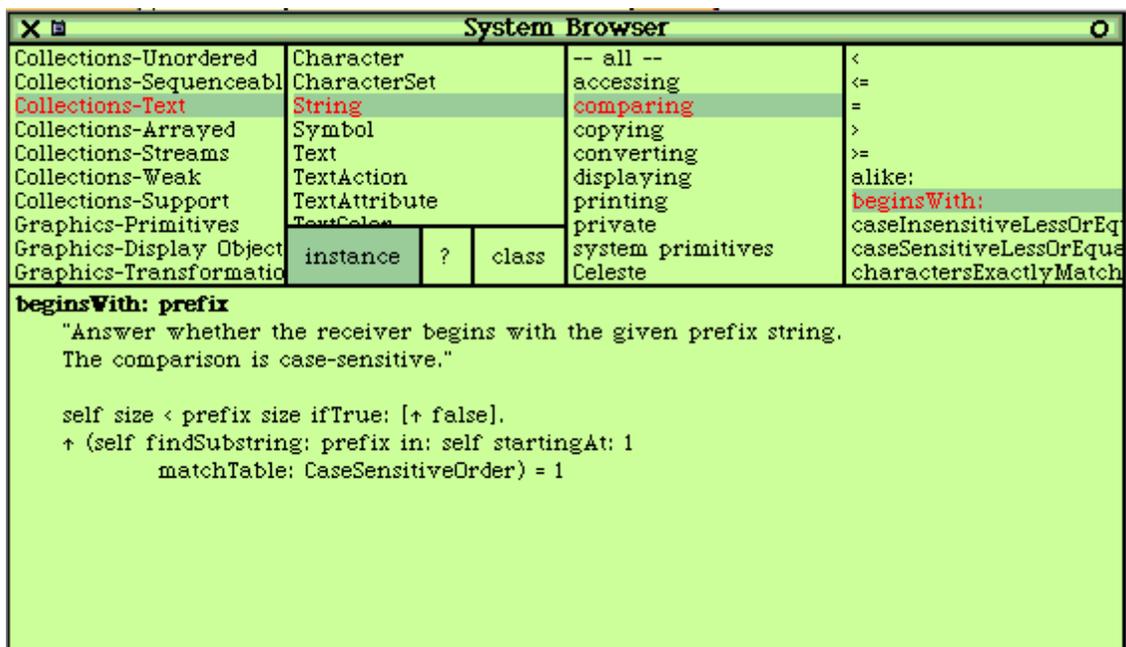

Figure 5: The System Browser

The System Browser allows you to view every single line of code that makes up Squeak, as well as being the main window for adding and editing

your own code. It is therefore a bit of an understatement to say that being able to use this browser effectively is critical to efficient Squeaking.

The top four panes allow you to browse and narrow your search to specific methods of Squeak code. From left to right:

- Category Pane: This pane presents *categories*, groups of Squeak classes arranged by functionality (the arranging is for human categorization only). Browsing up and down his column will give you some idea of the breadth of Squeak. Right-clicking in this pane will allow you to do things like search for a specific class or add a new category. Selecting a category brings up a list of classes in the…

- Class Pane: This pane presents all the classes that are members of a particular category. These are the classes that make up Squeak's object-oriented hierarchy. Right-clicking in this pane will bring up a menu of options for more information about the selected class. Pressing the "?" button will bring up a comment about the class. The "class" and "instance" buttons control what kind of information about the class is displayed. Selecting a class brings up a list of method categories in the…

- Method Category Pane: This pane presents a list of method categories for a particular class. The categories, which are for human readability only, divide the functionality of a class into more easily browsed chunks. Right-clicking on this menu allows you to add, rename, or organize categories. Selecting a category brings up a list of methods in the…

- Method Pane: This pane gives you a list of methods in the selected category of the selected class. Right-clicking here gives you a menu of more information about the selected method. Selecting a method allows you to see the code in the code pane.

The code pane takes up the bottom half of the System Browser and allows you to view and edit Squeak code. From here you can also create new classes and methods.

As I mentioned, the System Browser contains all the code that runs Squeak, as well as the code that you will write. Since over 95% of Squeak is written in Squeak that includes just about everything. The code that displays the System Browser, interprets and evaluates text, the file system, the examples you'll see in this book. Some of the code covers the most current cutting-edge multimedia formats, like Flash. Some of the code is so old that the comments refer to the Xerox Alto. Feeling a little

overwhelmed is a common first reaction. Feeling empowered is a common second one.

This run through the basics of the Squeak interface is necessarily brief, and I've left out pages of things like keyboard shortcuts, and how to change the colors. The end of this chapter will tell you where to look for that information. As for us, it's time to look at Squeak, the programming language, in more detail.

Think Small

Smalltalk has a different structure than the object-oriented programming languages that you are most likely to be familiar with. Unlike C++ or Java, all variables in Smalltalk are objects, including integers, Booleans, floating point numbers, and characters. There are no "basic types" to clutter up the syntax. In addition, most of the structures that we traditionally think of as syntax, such as loops and conditionals, are implemented as part of the Smalltalk object library, and are not special syntactic forms. Smalltalk is designed around a very few syntax rules, applied consistently throughout the language, and providing for maximum flexibility for the programmer. Most of the language issues in this chapter also apply to other dialects of Smalltalk, but a few of them do not (and the interface details are Squeak specific). To minimize confusion, I will continue to refer to the environment and language as Squeak, rather than Smalltalk.

Squeak in a Note Card (Who needs a whole nutshell?)

The basic syntax of Squeak can be summarized easily on a 3x5 note card. Every line of Squeak code is evaluated in exactly the same way.

- Every variable is an object. Like C++ and Java, every object is an instance of a class, and has its instance variables and methods determined by that class. Unlike C++ and Java, there are no basic types that are not objects.

- All Squeak code is triggered by a message being sent to a specific object. The object replies to a message by evaluating a method of the same name. If the object does not have such a method, its parent object is checked for the method, and so on up until either the method is found, or an error is raised.

- All methods return a value.

- There are three types of messages.

- o unary messages, such as 3 negated. The syntax of a unary message is <object> <messagename>

- o binary messages, such as a + b. The syntax of a binary message is <object> <messagename> <object>.

- o keyword messages such as Transcript show: a. The syntax of a keyword message is <object> <messagepart>: <object>, where there can be multiple <messagepart>: <object> pairs that make up a single message.

- All code is evaluated from left to right, unary messages first, binary messages second, and keyword messages last. Parentheses are used as in other programming languages to force order of evaluation, and are frequently used to mark the boundaries of keyword messages, where that boundary might not be clear from the unparenthisized code.

- In an assignment statement, the right hand side is evaluated, and the symbol on the left hand side is assigned the resulting value.

And that's it. Every line of Squeak code follows exactly that pattern[6].

Applying the Rules

The next thing you need to understand about Squeak's syntax rules are that there are no exceptions for things such as, say, traditional operator precedence. Operator precedence in Squeak is strictly left to right. So, for example: 2 + 3 * 6 will evaluate to 30, not 20 as you would expect in most languages. If you want it to evaluate to 20, you need 2 + (3 * 6). Most Squeakers, consider this a small price to pay for the consistency and readability of Smalltalk code.

Here's an incorrect line of code for computing the length of the hypotenuse of a right triangle[7]:

---

[6] Okay, one exception – the syntax ^object for returning the result of a method. Alan Kay explains that this can be thought of as a message sent back to the calling object.

[7] Squeak currently uses two separate symbols for assignment – the more familiar := syntax, as well as the underscore character _, which is displayed as a left-pointing arrow. The two are functionally identical.

hypotenuse := 3 squared + 4 squared sqrt.

The sequence of messages evaluated in this example is as follows:

- On the right hand side of the equation. The interpreter evaluates the unary messages from left to right before it can evaluate the binary message +. The first unary message is 3 squared, which returns the integer object 9.

- The second unary message is 4 squared, which returns 16.

- At this point we have 9 + 16 sqrt, and the interpreter still has a unary message to evaluate before it can perform the addition. 16 sqrt returns 4.

- Now the addition is performed, returning 13.

- Hypotenuse is set to 13.

It's not uncommon to have this kind of message traffic jam in Squeak. Parentheses are the traffic cops of choice, turning the line of code into:

hypotenuse := (3 squared + 4 squared) sqrt.

The sequence of messages is the same until after message number 2, after which the sequence becomes:

- At this point we have (9 + 16) sqrt. The system can perform the addition, which returns 25.

- The interpreter now has 25 sqrt, which returns 5.

- Hypotenuse is set to 5

Object Orientation

The object-oriented semantics of Squeak are geared towards simplicity, elegance and flexibility, without the syntactic clutter of C++ and Java. To a programmer coming to Squeak from either of those languages, the Squeak object-oriented structures may seem to be missing features. After working with Squeak for a while, though, you are more likely to realize that the ease of working in Squeak more than makes up for the more complex and rarely used feature set of other object-oriented languages.

Squeak For Non-Native Speakers

Creating Objects

To start, Squeak's object-oriented hierarchy allows only single inheritance. Multiple inheritance is not supported[8], and there is no feature analogous to Java interfaces[9] or C++ templates (the dynamic nature of ordinary Squeak objects is what templates are trying to emulate).   All classes in Squeak inherit from the class Object, which has more functionality than its Java counterpart.

To create a new class of your own, open a System Browser, and select any category in the first pane. The code pane will show the following:

Object subclass: #NameOfClass

  instanceVariableNames: 'instVarName1 instVarName2'

  classVariableNames: 'ClassVarName1 ClassVarName2'

  poolDictionaries: ''

  category: 'Kernel-Methods'

This is a template for class creation. All you need to do is replace each slot with your needed values.   Replace Object with the expected superclass of the new class. Replace NameOfClass with the name of the new class (but leave the pound sign – that tells Squeak to consider the name a symbol literal rather than a variable). By complier-enforced convention, class names begin with capital letters. Replace 'instVarName1 instVarName2' with the list of instance variables in this class, separated by spaces. Replace 'ClassVarName1 ClassVarName2' with the list of class variables. We'll create a class called "person" for the rest of these examples. Change the code so that it reads:

Object subclass: #Person

  instanceVariableNames: 'firstName lastName'

  classVariableNames: 'Population'

  poolDictionaries: ''

  category: 'Tutorial'

When you are done, right click in the code pane, and select "accept". And just like that, a new class that you can select in the System Browser. You may need to right click in the category pane and select "update" to see the new category. There is nothing special about this code – it's just a Squeak message being sent to the class Object and creating a new subclass.

---

[8] Although it's actually not hard at all to hack it into the Squeak system by modifying the Object class – it's just rarely worth the trouble.

[9] Although some Smalltalk dialects do have features similar to interfaces, and it's not out of the question that Squeak will have one someday.

Each Squeak class defines instance variables and instance methods. Each instance, as you might expect, gets it's own copy of the instance variables, and can respond to the messages corresponding to the instance methods. By convention, instance variables start with lower case letters, and are mixed upper and lower case, i.e., "firstName". Any instance variable can be assigned any Squeak object as a value – there is no static typing in Squeak.

A Squeak instance variable can only be accessed from within that class or one of it's subclasses. In C++ terminology, all Squeak instance variables are *protected.* (Java's protected keyword also includes access within packages – there is no similar access path in Squeak). All access to the variable from outside the class must go through accessor methods. By convention, the getter method is just the name of the variable, and the setter method is a keyword message using the name of the variable and taking one parameter. For example, to get the firstName of an instance of a class Person, you would say:

aPerson firstName.

And to set the variable, you would use.

aPerson firstName: 'noel'.

Which shows another piece of Squeak syntax – the use of single quotes around string literals.

Note that you do have to explicitly write the getter and setter methods.[10] To do that, select Person in the class pane, and select one of the elements in the method category pane. The code pane should read like this:

message selector and argument names

    "comment stating purpose of message"

    | temporary variable names |

    statements

This is a template for method code. The name of the method and keywords goes in the first line. A comment is inserted into the double quotes. A list of temporary variables for the method goes between the pipe characters, separated by spaces, and the code itself goes after that. When you are done typing code into the code pane, right click in that pane and select "accept". If correct, the code is compiled into the Squeak image

---

[10] There is a Squeak preference that you can set that will automatically create them when called, but it's best to get in the habit of just writing them – it's easier to maintain your encapsulation that way.

– if not, error messages will appear in the code pane at the location of the error.

The code for the getter message looks like this:

firstName

    ^firstName

And the setter looks like this

firstName: aString

    firstName := aString

The ^ symbol indicates a returned value. The aString in the header and first line of the setter is the local name for the object passed as the argument. All Squeak methods return a value – if the message doesn't specify the value, it returns the object which received the message.

By the way, you don't need to go all the way back to the message template to write a new method – if you change the method name in the first line of the code pane, the method will be accepted under the new name without affecting the old method name. This can be a quick way to write a number of similar methods.

Instances and Classes

Classes, like everything else in Squeak, are objects – in this case, instances of the class Class. Object manipulation in Squeak is performed by sending messages to the class Object – for example, the creation of a subclass, as shown above.

A Squeak class creates a new instance by being passed the message new, as in:

    newPerson := Person new.

The new message acts very much like a default constructor in C++ or Java, it returns a new instance. By default, all the instance variables of the new instance are set to nil. The usual idiom is to create an instance method called initialize, and refer to it as follows:

    newPerson := Person new initialize.

Tracing the Squeak interpreter shows that the first message Person new returns a new instance of the Person class. That new instance is then sent the initialize message, and the initialized instance is then assigned to newPerson.

You can create variables that belong to the class, rather to the instances, by putting class variable names in the subclass creation template. Class variable names are capitalized. Class variables behave

similarly to static variables in C++ and Java, however, like Squeak instance variables, they are all private, and can only be accessed within instances of that class.

Classes can also have methods. To view and create them, select the class button in the class pane of the system browser. One use of class methods is to create constructors that take parameters, for example, the message Array with: anObject which creates a new array and populates the first index of that array with anObject. Another use is to provide accessor methods for class variables.

Squeak has no explicit destructor methods. The Squeak garbage collector periodically removes all instances with no external references.

Inheriting The Wealth

Squeak objects inherit all the instance variables and methods of their ancestors all the way up to the class Object. Unlike C++ and Java, Squeak is completely late-binding, and performs no compile-time type checking to determine if the object actually answers the message being sent (although you will get an error if no object anywhere in the image defines that message). The method that is invoked from a message is based on the identity of the receiving object at run time. The flexibility of this is shown when you see a line of code that sends a message that is defined by a number of Objects across the hierarchy:

anObject asString.

Any Squeak object that defines (or has an ancestor which defines) the message asString can legally be assigned to anObject before this line is run. The specific asString that is run depends on the identity of anObject – Squeak first attempts to see if there is an asString instance method in anObject's class. If yes, that method is invoked. If not, Squeak works up the hierarchy until an asString method is encountered. If the method is not encountered, then Squeak will return an doesNotUnderstand error.

To override a method in a parent class, just define a method of the same name in the child class. If you need to call the parent method to extend it, Squeak provides the pseudo-variable super, which responds to the message sent as if starting one level up on the hierarchy. To refer to the current instance itself, Squeak uses the pseudo-variable self, which can be used either to send other messages to the same object, or to pass the object as a parameter in a message, analogous to C++ and Java's use of *this*. The variable self can also be used as an argument to a message, as in anotherObject doSomethingWith: self.

Squeak For Non-Native Speakers

Another important message for the Squeak object-oriented framework is subclassResponsibilty, which is used similarly to the C++ and Java keyword *abstract*. If you wish to declare a method in a superclass without a definition, with the intent that all subclasses will need to define it, then you create the method with the message body self subclassResponsibility. You can still create instances of the class with that method, but trying to invoke that method will result in an error.

Finding What You Need, Keeping What You Write

Most programmers coming to Squeak for the first time are used to programming environments where the code is stored in a series of text files, one for each class or module. The transition to Squeak's image based system can be jarring – it's an entirely different way of organizing and managing code. New Squeakers frequently struggle with finding existing code, maintaining their new code, and sharing code with other programmers. Squeak has a number of features that will assist with all of these needs.

The Art of Browsing

The System Browser is where you will most likely spend the majority of your time as a Squeaker. Each of the top panes in the browser window has menu items that allow you to better find or organize code at that level of detail. This section will cover most of them – there are other menu items that have little to do with organization and navigation, and will not be covered here. Many of these menu items call up windows that are cousins to the System Browser – the same idea, but organized around a different (usually smaller) structure. They all work essentially the same way, however, and code can be created or edited in any of them just as it is in the main System Browser.

Starting at the top left of the System Browser again, the category pane is by itself a structure for navigation and organization. As mentioned previously, the categories here are solely for human readability, the compiler pays no attention to them. There is no better way to get a feel for what Squeak provides you than to browse the category headings. To best use that power for yourself, you should give each of your projects a separate category. It's also common to maintain a category or two of common utility code that you would use in several projects. Large projects are often broken into more than one category, separating interface code from back end code is common.

Among the menu items available by right clicking on this pane are:

find class	Presents you with a dialog box where you can enter a class name or name fragment and get a list of all classes in the current image that match. Since most classes in the Squeak image have names that roughly correspond to their functionality, this can be extremely useful.
recent classes	Presents a list of the last 16 classes selected in a System Browser, and allows you to go directly to the one you choose.
browse all	Gives you a System Browser cousin that has no category pane. Instead, the class pane contains all classes in the Squeak image alphabetically.
browse	Gives you a System Browser cousin that only displays classes in the category selected.
reorganize	The code pane becomes a series of lists corresponding to the current relationship between categories and classes. With a little judicious editing, cutting, and pasting, you can move classes around to different categories. This method is easier than editing each class if you are going to change several classes at once.
update	Updates the System Browser to take notice of category additions or edits made in other System Browsers

The class pane has quite a few menu items to help you find related classes and methods.

browse class	Opens a new System Browser cousin with just the method categories and methods of the selected class.
browse full	Opens a new System Browser, with the selected class selected.
hierarchy	Puts, in the code pane, a text representation of the object hierarchy containing the selected class.
spawn hierarchy	Opens a new browser window. In the class pane of this window is the object hierarchy containing the selected class.

Squeak For Non-Native Speakers

spawn protocol	Opens a new browser window with only one pane on top that contains the methods of this class in alphabetical order.
inst var refs…	Gives you a menu of instance variables in this class. Selecting one returns a browser with all methods in the class that use that instance variable.
inst var defs…	As above, but returning all methods in the class that define (assign a value to) that instance variable.
class var refs…	As "inst var refs…" but with class variables.
class vars	Opens an inspector window which allows you to see the current values of all class variables of the selected class.
class refs	Opens a browser with a list of all methods that explicitly reference (by creating instances of, usually) the selected class.
unsent methods	Generates a menu of all methods in the current class that are never sent by any object anywhere in the current image.
Find method	Generates a menu of the methods of the current class. Selecting an item in that menu takes the current browser to that method.

The method category pane has relatively few menu items for navigation. Like the class category pane, it is not used by the compiler and interpreter. You should take advantage of it, however, to help organize classes that have multiple methods. Perusing the Squeak image, you will see that certain category names are used by convention, including initialize (or initialize-release), adding, accessing, testing, converting, and printing. Using category names that are already enshrined in the image will make it easier for others to read and use your code.

browse	Opens a code browser with all the methods in the selected category.
reorganize	Similar to the item in the class category pane, this item allows you to use text editing to organize the methods into categories.

alphabetize	Organizes the categories in the pain to alphabetical order.
remove empty	Removes categories in the pane that have no methods.

The code pane has a very large menu attached to it (the bottom entry "more…" opens a second page of the menu). Many of these items are duplicates of the category pane, and I'm not going to repeat those.

senders of…	Returns a code browser with a list of any method in the image that sends the selected message. However, not all of these methods are necessarily sending it to this class all the time – if you are searching on a common message such as add: it's likely that most of the messages are not aimed at the class you are in – but they potentially could be.
implementers of…	Returns a code browser with a list of every method in the image that implements the same message in other classes. This is where you would find all the other add: messages in the image.
method inheritance	Walks up the object hierarchy by returning a browser that shows you all the superclasses of this object that also define this message.
versions	The Squeak change facility keeps track of every accepted version of a method until specifically purged. Selecting this shows you a browser of the previous versions of this method.
implementers of sent messages	Otherwise known as "what code can I break" this item shows you every method of class that implements any message sent in the selected method. In other words, it's anyplace in the image that could be called from this code. Note that it does not take into account any information in the code about what classes are expected – it displays everything.
change sets with this method	A list of all change sets currently being maintained in the image that contain this method.
inspect instances	Inspect instances gives you an inspector window

inspect subinstances	with all instances of the selected class currently living in the image. Inspect subimages also includes instances of subclasses of the selected class.

Saving, Loading and Sharing

The easiest mechanism for saving Squeak code is merely to continuously save the image. Using the image as your code repository has many advantages. It stores the values of global and class variables, stores all previous versions, and requires no particular attention to detail beyond remembering to save every time you leave Squeak.

Unfortunately, there are a few drawbacks to relying on images. Although it doesn't happen often, it's possible that an image can become corrupted, and although the .changes file means you don't actually lose anything, recovering from an image crash can be a pain. Image files are rather large, and maintaining regular backups will eat up space more quickly than strictly necessary. Also, it's difficult to share code in an image file with another programmer who wants to install the code in their image.

Squeak provides two related mechanisms for saving code as text for easy backup and sharing. The simpler mechanism is called a *file out*. To create a file out, right click on any of the top panes of the System Browser or related code browser, and select the "file out" menu option. A file is created in your Squeak directory containing the contents of the selected category, class, method category, or method, depending on what pane you started in. You do not get a dialog to choose the name – the name of the selected item is used (for methods and method categories, it's the name of the class followed by the name of the method or category). The extensions on file outs is ".st". The file is text, and is more or less human readable – it includes some delimiter characters to allow the compiler to read the file back in.

To load a file out into your Squeak image, you open a File List by selecting "open" from the main World Menu, and then selecting "file list". A file list lets you browse your system's hard drive and has three panes. The top right pane contains a list of the files and subdirectories in the directory being browsed. The top left pane has the file tree down to the current directory, and the bottom pane shows the contents of the current file. To file in a .st file, browse to it so that the contents of the file are showing in the bottom pane, right click, and select "file it in" from the menu. Ordinarily, you'll see a progress bar, and you may see some

warnings in a Transcript window, if classes are loaded with dependencies to classes that have not yet been loaded. Don't worry – assuming all the classes are in the file out at all, it will all be fine as soon as everything loads. After filing in the .st file, do an update on the category pane of a System Browser, and you will see that the new code has been loaded into the image.
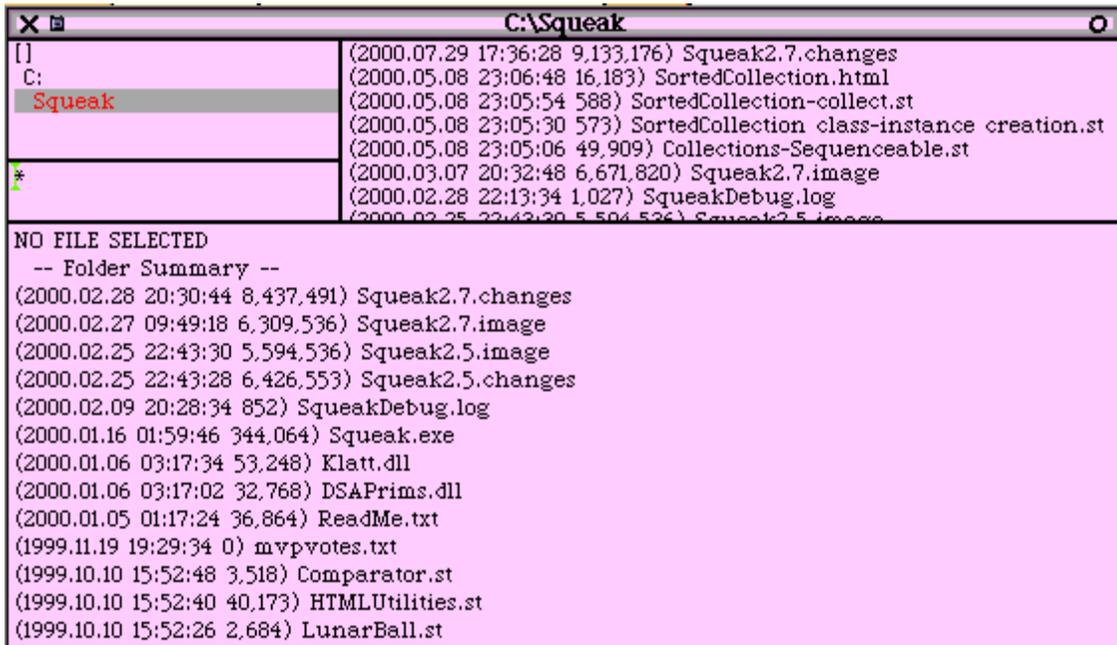


```
X ▣                        C:\Squeak                            O
[]              (2000.07.29 17:36:28 9,133,176) Squeak2.7.changes
 C:             (2000.05.08 23:06:48 16,183) SortedCollection.html
  Squeak        (2000.05.08 23:05:54 588) SortedCollection-collect.st
                (2000.05.08 23:05:30 573) SortedCollection class-instance creation.st
                (2000.05.08 23:05:06 49,909) Collections-Sequenceable.st
*               (2000.03.07 20:32:48 6,671,820) Squeak2.7.image
                (2000.02.28 22:13:34 1,027) SqueakDebug.log
                (2000.02.25 22:43:30 5,504,536) Squeak2.5.image
NO FILE SELECTED
 -- Folder Summary --
(2000.02.28 20:30:44 8,437,491) Squeak2.7.changes
(2000.02.27 09:49:18 6,309,536) Squeak2.7.image
(2000.02.25 22:43:30 5,594,536) Squeak2.5.image
(2000.02.25 22:43:28 6,426,553) Squeak2.5.changes
(2000.02.09 20:28:34 852) SqueakDebug.log
(2000.01.16 01:59:46 344,064) Squeak.exe
(2000.01.06 03:17:34 53,248) Klatt.dll
(2000.01.06 03:17:02 32,768) DSAPrims.dll
(2000.01.05 01:17:24 36,864) ReadMe.txt
(1999.11.19 19:29:34 0) mvpvotes.txt
(1999.10.10 15:52:48 3,518) Comparator.st
(1999.10.10 15:52:40 40,173) HTMLUtilities.st
(1999.10.10 15:52:26 2,684) LunarBall.st
```

Figure 6: File List Window

File outs are limited, however, to only saving entire categories or classes, and only one per file. To work around that limitation, you can use a change set. Every change you make to the Squeak image is stored in one or more change sets, and these sets can be saved as text and loaded just like file outs (only with a .cs extension). So, any combination of changes you have made at any time to the current image can be combined into a change set, saved, and shared.

Easily said, but you do have to manage the change sets a little bit to make sure the right changes are in the set you want. The easiest way to do this is to create a Squeak project. By default, all changes made inside a project are stored to a change set of the same name. So all the changes in a project can be saved together, which is often good enough to distribute code.

For a more targeted change set, you can use a Change Sorter. Change Sorters allow you to see the contents of change sets, and to modify them. You create a change sorter from the World Menu open command or from

the World Menu changes submenu, and they come in two flavors, Simple and Dual.

A simple change sorter shows you all the Change Sets in the system – that's the top left pane. The top right pane is a list of all classes that have changes in the selected change set. The middle pane contains a list of methods in the selected class that are in the change set (not necessarily a list of all methods that the class has). And the bottom pane shows the code of the selected method, and you can edit and accept code within it.

Figure 7: Simple Change Sorter

From the top left pane, you can right click and select "new change set…" to create a new change set, then select it, right click and select "make changes go to me". From that point on, all changes in that project will go to that change set. This is particularly nice if you know you are only going to have a few changes that you want saved or distributed.

For even more control over the change set, a Dual Change Sorter looks like two Simple Change Sorters glued together. You can create a new change set on either side, and direct changes from the selected set on one side to be copied to the selected set on the other side. This is the way to go when you only want to distribute some of your recent changes, or if you want to distribute a number of disparate changes across the image.

Change sets are filed out by right clicking on the change set pane and selecting "file out", and are loaded into the image by using the File List, exactly as ordinary file outs are loaded.

Squeak For Non-Native Speakers

Figure 8: Dual Change Sorter

Classes You Should Know

By now, you are probably wondering exactly how all this plays out in terms of things that you actually do during your day to day existence as a programmer. There are several Squeak classes that you should be acquainted with to support common programming structures.

Building With Blocks

Nearly all of Squeak's structured programming constructs depend on a class called BlockClosure, which has no direct analogue in C++ or Java. A BlockClosure (usually just called a block) is a lump of compiled Smalltalk code that can be passed to or returned from methods just like any other object. The block can then be executed at any time.

You create a block by enclosing the code to be blocked inside brackets. For example, type the following into a workspace:

[2 + 3]

Performing a printIt on that line of code gives you: [] in UndefinedObject>>DoIt which mostly means that you have an unevaluated BlockClosure. In order to get the block to do anything you have to pass it the value message:

[2 + 3] value

Performing a printIt on that line of code gives you five. You can also assign the block to an object:

a := [2 + 3].

a value.

Will also evaluate to five.

Blocks can also have variables with scope local to the block, which you create like so:[11]

[:a :b | a + b]

And evaluate by passing the message value: with parameters, for example:

[:a :b | a + b] value: 2 value: 3

Will also evaluate to five.

Currently in the Squeak image you can have up to four value: in a row. If you have more temporaries than that, you can either a) seriously consider refactoring, b) add the message you want or c) use valueWithArguments: anArray.

Big deal, right? I mean, we just came up with a really convoluted way of doing things that we could already do anyway. The block concept is very powerful, however, and allows Squeak objects to have much more flexible behavior than their C++ or Java counterparts.

Consider, for example the problem of maintaining a sorted list. Typically this is managed by comparing new objects in the list to the existing objects. In Java 1.1 and in most C++ implementation, you have to roll this functionality on your own. In Java 1.2, which partially replicates the Smalltalk collection classes, this can be accomplished by having your class implement the `Comparable` interface and adding a new method to every class you want sorted. Or you can mess with inner classes. Forget about easily sorting objects of different classes.

In Squeak, as we will see in more detail below, the same functionality is accomplished quite easily in the class SortedCollection by the simple application of a block. SortedCollection has as an instance variable called sortBlock which, you can see by looking at the class method SortedCollection>>new defaults to [:x :y | x <= y]. This deftly handles

---

[11] It's actually a long-standing issue in the Squeak implementation that the scope of block variable is actually larger than the block itself. However, for reasons of good programming practice and compatibility with future versions of Squeak, you should never depend on a block variable being available outside the block.

Squeak For Non-Native Speakers

any object that implements the <= method, and if you want to sort on some more complex set of values you just send a sortBlock: message to your collection and pass it any complicated block that you want (well, it does have to have the two temporary variables).

Blocks are commonly used to provide "pluggable" behavior to classes, where a class might be called upon to provide the same kind of behavior in many different contexts. As we'll see below, control structures such as loops and conditionals are prime examples – an if statement always does the same thing structurally, even though the specific code being executed changes each time. You can also easily create your own customized control structures that also take blocks as arguments.

Conditionals, Loops and Other Programmer Type Things

Structured constructs such as conditionals and loops show off the elegance of Squeak's pure object-oriented system.

A standard if/then statement is handled in Squeak using the ifTrue:ifFalse message, like this:

(a = b) ifTrue: [Transcript show: 'equal']
            ifFalse: [Transcript show: 'not equal'].

The parameters passed to the ifTrue: and ifFalse: keywords are blocks, and as discussed in the previous section can either be typed out as shown above or be variables previously set to block values. When the message is evaluated, one block or the other is evaluated based on the value of the Boolean expression.

You may have noticed some hand waving in the previous sentence. After all, I had made a point of saying that everything in Squeak is a message sent to an object, and there are no syntactic special forms. What object, then, receives the ifTrue:ifFalse: message?

Tracing out the receiver of the message shows that the = message is sent to the object a. The = message returns either true or false, which means that the message is actually being sent to either true or false. And, if you point your System Browser at the 'Kernel – Objects' category, you'll see a class called False and a class called True. Each of these classes is a Singleton, meaning that they each have only one instance object – false and true respectively. (If I haven't mentioned that Squeak is case-sensitive so far, now would be a good time to squeeze that in – capital 'T' True is the class, lowercase 't' true is the value.). Further browsing in the True class, shows that True>>ifTrue:ifFalse: is simply evaluated as follows:

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

^trueAlternativeBlock value

In other words, to evaluate if-then statements, Squeak depends on the principles of object and message dispatch to control which branch is chosen.[12] False>>ifTrue:ifFalse is the exact same one-liner, except that it is the falseAlternativeBlock that is sent the variable message. The messages ifFalse:ifTrue, ifTrue:, and ifFalse: are also available and are implemented analogously.

Looping constructs in Squeak start similarly, with the WhileTrue: and WhileFalse: messages. They are, as you would expect, Squeak's while loop messages. There is, however, one significant "gotcha" compared to the if messages discussed above. Unlike the if messages, which are sent to Boolean objects, the while messages are sent to blocks that return Booleans, such as:

[x < 3] whileTrue: [x := x + 1].

There is a reason for this oddity, only I had to stare at my Squeak screen for a few minutes before I remembered what it was. The block allows the receiver of the message to be re-evaluated before each potential loop iteration, whereas a static Boolean object would not be re-evaluated. The argument block of the whileTrue: in this case, [x := x + 1], is evaluated every time through the loop, for as long as the receiver block continues to evaluate to true.

Squeak has two mechanisms that give for loop functionality. The simplest is the timesRepeat: aBlock message which is sent to an Integer:

4 timesRepeat: [Transcript print: 'Hi'].

The block is repeated the value of the integer times. The times repeat construct does not give you access to the index variable within the block. To do that, you need the to:do: message, which looks like this.

1 to: 4 do: [:index | index + 1]

The to:do: message is sent to any integer[13] and is defined thusly in the Squeak image (although, usually compiled inline in practice).

to: stop do: aBlock

---

[12] In practice, actually, the Squeak compiler usually inlines Boolean calls into the method body for performance purposes.

[13]Technically the message can be sent to any number – it's defined in the Number superclass of all numbers. However, it's really only meaningful for integers. The general behavior of these kinds of intervals for non-integers was a raging topic of debate on the Squeak mailing list a few months back.

```
| nextValue |
        nextValue _ self.
        [nextValue <= stop]
                whileTrue:
                        [aBlock value: nextValue.
                        nextValue _ nextValue + 1]
```

We've already discussed all the Squeak constructs in this code. The first line creates a temporary value, which is then set to self – in this case the number that receives the message (1 in the above call). Line 3 defines the block that is the loop condition, while line 4 sets up the loop itself. Inside the loop, the argument block [:index | index + 1] is sent the value: message with the index of that pass through the loop, and then the counter is incremented. This means that the argument block must have exactly one temporary variable defined to accept the index value.

Squeak also offers the to:by:do: message, which allows you to increment (or decrement) by integers other than one (but not zero). Given the definition of to:do: it shouldn't be hard to figure out how to:by:do: is written – or you could just look it up in the Number class.

Collecting Objects

Just about every programming language created since about 1945 has included some mechanism for manipulating a group of variables, most commonly by using a numerically indexed array of values. Few languages, however, offer as rich and varied set of collection options as Smalltalk. Of all the features of Smalltalk, the collection classes are the ones that I most frequently miss when I'm programming in other languages – it's only in recent years that other languages have even come close to the depth of the Smalltalk collection hierarchy.

In Squeak, the collection classes are defined in the seven categories prefixed with "Collection", and yes, that's seven categories, not seven classes. In fact, it's 65 classes in Squeak 2.7, if I'm counting right (admittedly some of them are support classes, not actually collections). Luckily, we only need to go through a few of them in order to get the general idea. The majority of your collection class use in Squeak will probably be in one of the concrete classes discussed in this section.

Usually, I find it easier to discuss the Squeak collection classes by starting at the top of the hierarchy – the abstract class Collection – and discussing the common features of all Squeak collections, then moving to the concrete classes and discussing their specific functionality. If you find

the next few paragraphs a little too abstract, it might be helpful to skip down a little bit to the discussion of a specific collection, then come back.

A Squeak collection is… well, there's no getting around the word collection… a Squeak collection is a collection of Squeak objects. Collections are not restricted as to the type or class of Squeak objects that they can contain, and, unlike arrays in many other programming languages, are not restricted to holding only objects of a single class at a time, nor are you required to specify the class of objects to be contained when defining or using the collection.[14]

The class Collection, which is abstract, defines the general protocol for all collections. Collection defines many messages that can be sent to any of the collection subclasses. Some of the most common, important, or interesting include size, which returns the number of elements in the collection. The method isEmpty, returns true or false based on whether there are no elements in the collection. The message includes: anObject returns true or false based on whether the argument is included in the collection. Collections can be converted from one type to another by using a method of the form asSortedCollection or asSet.

The most useful features of the collection class are the enumeration classes that perform actions on each element of the collection. The general form is exemplified by the do: aBlock message, which takes as an argument a single variable block and sends each individual element in the collection to the block. do: returns the original collection, so it is usually called for its side effects. For example, size is defined as:

size

```
| tally |
tally := 0.
self do: [:each | tally := tally + 1].
^tally
```

This message simply runs a counter and increments it once for every element in the collection. Notice that in this case, although the block takes an argument, the argument is not used. It is needed, however, because the implementations of do: will send the one argument value: message to the block.

---

[14] And having made that bold statement, I admit in the fine print that, while it's true for most collections, certain collections do limit themselves to specific classes for performance reasons, or for other logical reasons. Strings, for example, are strictly collections of characters.

Squeak For Non-Native Speakers

If you want to return a changed list, then you can use the collect: aBlock message. Like do:, it takes a one argument block and applies it to each element. However, rather than returning the original collection, it returns the new collection formed by the result of each block application in order.

You can also filter a list using select: aBlock, which takes a one argument block and returns a collection containing all the elements for which the block evaluates to true.

And most fun of all, there is inject: thisValue into: binaryBlock. It's actually easier to display the code for inject: then it is to describe it, so here goes[15] (I'll give examples of inject: and the other enumeration methods in a moment).

```
inject: thisValue into: binaryBlock
        | nextValue |
        nextValue := thisValue.
        self do: [:each | nextValue := binaryBlock value: nextValue value: each].
        ^nextValue
```

It is useful, really, and you'll see how in just a moment.

The specific subclass of Collection that is most similar to what you have likely done before is Array. Arrays in Squeak work quite similarly to arrays in other programming languages. They have a fixed size, although in Squeak that size is set at run time and not at compile time. Like other Squeak collections, any array can have any time of objects as components.

Arrays are created in a variety of ways. Literal arrays are created with a hash sign and parentheses: #(1 2 3) is a three element array. They can also be created using the class method with: and it's variants. Array with: 1 with: 2 with: 3 creates the same three element array. So, as promised, some enumeration examples:

#(1 2 3) collect: [:each	each squared]	(1 4 9)
#(1 2 3) select: [:each	each even]	(2)
#(1 2 3) inject: 0 into: [:subTotal :next	subTotal + next].	6

---

[15] Okay, if you do want it explained, here's the method comment, "Accumulate a running value associated with evaluating the argument, binaryBlock, with the current value of the argument, thisValue, and the receiver as block arguments."

The inject one is the trickiest. Looking at the implementation of inject: above, we can see that on the first pass through the loop, the block is sent the arguments 0 and 1. It sums them and returns 1. Next time through, the arguments are the running total,1 and the next list element, 2, returning 3, and finally the arguments are 3 and 3, returning 6. It sums the collection.

Arrays are accessed using the at: message: #(1 2 3) at: 2 returns 2. This points out a large difference between Squeak arrays and C and Java – the first element of a Squeak array has the index 1, not the index 0. Array indices are set using the at: put: message: #(1 2 3) at: 2 put: 4 returns #(1 4 3). There are also handy messages like first and last, that return what they say they return.

Arrays are actually only one of the subclasses of Collection that also inherits from SequenceableCollection – essentially any collection where the elements have a sequence, and thus integer indices. Another useful subclass of SequenceableCollection is OrderedCollection, which is an array without the fixed size restriction. In addition to the array access messages, an OrderedCollection also responds to add:, which adds an object to the end of the collection, and add addFirst:, which adds one to the beginning. The combination makes OrderedCollection useful for mimicking stacks or queues. And, as mentioned above in the section on blocks, there is SortedCollection, which maintains the sequence in sorted order based on a comparison block. Unlike Array and OrderedCollection, SortedCollection does not respond to the message at: put:, since the sorted collection insists on controlling where in the sequence a new member belongs.

The sequenceable collection that you are likely to use most often, however, is String, which is defined as an ArrayedCollection of characters. String literals are enclosed in single quotes (double quotes are reserved for comments). Strings in Squeak are mutable in content, but not in length, and can be changed using the same basic methods discussed for arrays. Strings are concatenated using the comma (,) operator – a method that actually also works for all sequenceable collections, should you have need of it. In addition to the existing methods for arrays, the String class has quite a few utility methods, including, but hardly limited to, methods for finding substrings and characters, comparison methods including methods for comparing the beginning and end of strings, conversions to date, number, HTML, and postscript, correction against a dictionary, enumeration over the lines of the string, and so on. A complete listing is obviously beyond the scope of this chapter, but most of the methods are short, and well commented – digging through them is a good way to get a

handle on some simple, useful Squeak code. Before you try and write some fancy utility on strings, take a look here, since there's a good chance that somebody has beaten you to it.

The ordered collections are similar to what you are probably used to from other programming collections, however the unordered collections don't have related functionality in most languages. The simplest of these is Bag. Bag is about as simple as a collection gets. It stores anything you put in it, as many times as you put it in, in no particular order.[16] Items are added to the bag using the add: anObject message, and are retrieved using… well, items in a bag aren't usually retrieved the way that items in an array are – bags have no order, and thus no index to grab the item. However, you do have access to the full power of the enumeration methods such as do: and select:.

If you come from a C background, you are probably wondering why you would ever use a bag. Often, however, the enumeration routines are everything you need in a collection – think of how many times you create an array and only access it in the context of looping through it (and if you do need it sorted, there's always asSortedCollection). In addition, using a bag when you don't need the indexes leads to time and space efficiencies – owing to the implementation of bags, testing for inclusion is particularly fast, as compared to arrays. Perhaps more importantly, it provides a comment on the programmers intentions and the expected use of the collection.

More structured than the Bag is Set, which mimics a mathematical set – an unordered collection of objects, each of which appears exactly once. Set enforces this by overwriting the add: anObject message to only add the new item if it does not already exist in the set. Converting a collection to a set using asSet and then converting it back can be a useful mechanism for removing duplicates from a collection.

The most useful unordered collection is Dictionary. A dictionary is a collection of associations – a key and a value. Both the key and the value can be any Squeak object. The keys are unique within a specific dictionary, while the values need not be. Objects are placed in the dictionary using at: anObject put: anObject and are retrieved using at: anObject. Therefore:

sample := Dictionary new.

sample at: 'abc' put: 123.

---

[16] In practice, each element of a Bag is only stored once, with a dictionary keeping track of how many times it's in the Bag – it's more space efficient.

sample at: 'abc'

The last line will return 123. A subsequent at:put: call to sample with 'abc' as the key will replace the 123 value with the new value. You may be familiar with this kind of functionality through Java Hashtables (without, of course, having to typecast everything) or Perl associative arrays (which are limited to string keys only).

Dictionary objects have additional enumeration functions – the standard do: function enumerates over the values in the dictionary, but there are also keysDo:, which takes a one-element block and enumerates over the keys in the dictionary, and keysAndValuesDo:, which takes a two element block and enumerates over the set of keys and values.(the key is the first argument to the block, and the value is the second, and also associationsDo: which takes a one argument block and applies it to all the association objects in the dictionary. There are also functions for testing and removing keys that are similar to the existing ones for handling values in other collections.

You Also Should Know The Debugger…

Despite all the myriad tools Squeak has to help you in your programming, it is nevertheless inevitable that you will have the occasional bug or two in your code as you write it. In order to help you find your way through that bug, here is a brief introduction to the Squeak debugger.

The debugger, in addition to being invoked by errors in the code, can also be invoked at any point in the code by including the message self halt.[17]   In order to show a typical use of the debugger, set up an example using the person class defined above (this example assumes you have getter and setter messages set for both the firstName and lastName instance variables).

Let's define the following method to display a person's full name. This method should concatenate three strings, the firstName of the person, a space, and the lastName of the person.

fullName

    ^firstName, ' ' lastName

Unfortunately, I seem to have forgotten the commas between the space and the last name – a potential syntax error. Let's see what happens if the code is evaluated.

---

[17]Remember, though, that everything in Squeak is an Object, including errors. Both user defined halt messages (class Halt) and inadvertent MessageNotUnderstood errors share the same superclass, and invoke the debugger through the same entry point.

In the workspace, type:

A := Person new.

A firstName: 'Noel' .

A lastName: 'Rappin.

A fullName

Now, select and doIt: for the entire chunk. You should get a message like the one in Figure 9.

The error message box describes the error, and provides a stack trace of all the messages leading to the error. It offers three options:

- Proceed: just keep going as though the halt had never been called. Usually only useful after a programmer placed halt.

- Abandon: stops the message processing at that point.
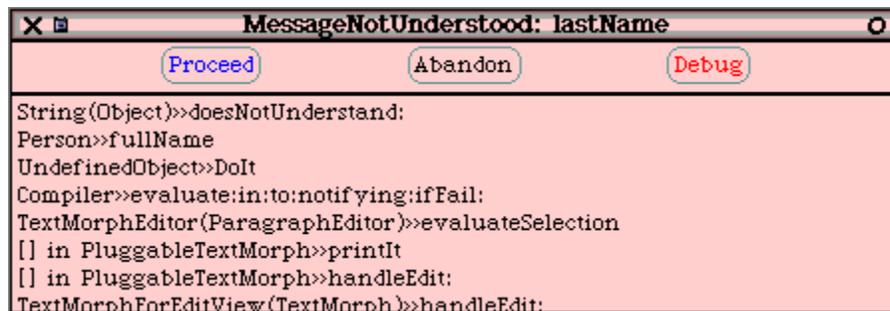
- Debug: opens the debugger window.

Figure 9: Error Message Box

The debugger window allows you to track the behavior of your errant program to a fine level of detail. The top pane of the window provides a stack trace, starting from the beginning of execution and ending in the message that actually invoked the error. Note, that as in this example, the top message Object>>doesNotUnderstand: is not the message with the actual error, rather, it's the message invoked by Squeak in response to the condition of asking an object to reply to a message that is not defined. It's not unusual to have to scroll down through one or two messages to reach the method that actually contains the defect. Right-clicking on the top pane gives you a menu that has both instructions for the debugger such as step forward one message, and also some of the method search functionality that is in the method pane of the system browser.
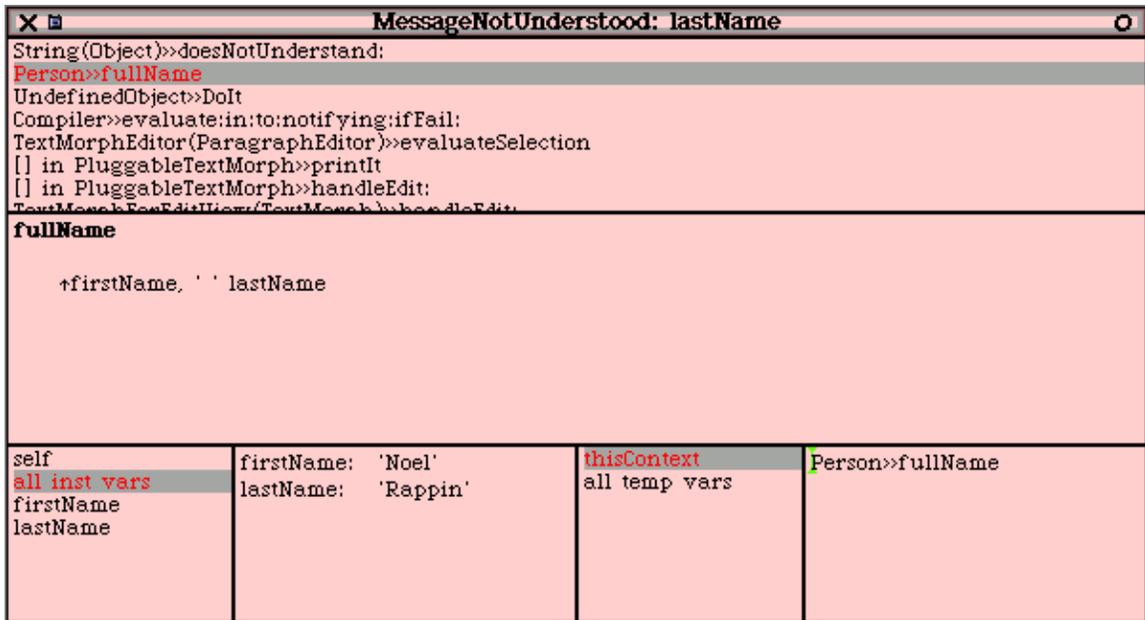
Figure 10: Squeak Debugger

The middle pane is an ordinary code editor. The last message to begin evaluation is highlighted. Right-clicking on this pane gives you everything you would expect in a code editor, including the ability to edit and accept code right in the debugger (and then, of course, continue onward as if nothing had ever gone wrong).

The bottom pane is split in half. The left half contains a mini inspector window for the object that is executing the message in the pane (in other words, the object returned by evaluating self inside that message). The top two entries are always "self" and "all inst vars" followed by a list of all instance variables in that object. Selecting one of the items in the list shows its value in the next pane, right clicking on one of them opens a menu with many of the search functions we have seen before, and the opportunity to open a full blown inspector window on the selected object.

The right half of the pane contains a similar list, but for the context of the message send itself, and containing the values of any variables temporary to the method. In this case, there are none, so all we see are the entries for "this context", and "all temp vars". Again, you have the opportunity to open full inspector windows on any of the objects listed.

Looking at the values in Object>>doesNotUnderstand, you can see that in that context self is ' ' – a string containing only a space, and the message is a Message with selector: lastName and arguments: (). In other words, without the comma, Squeak is evaluating the code: ' '

lastName as (what else) a message send. In this case the string ' '
doesn't have a message called lastName, which is what triggers the error.

For More Information…

The main Squeak web site at http://www.squeak.org is relatively small, but
contains pointers to all the current versions in all operating systems
currently supported, and information about the Squeak mailing list. Much
of the other information on the site is not updated frequently.

The Squeak Swiki at http://minnow.cc.gatech.edu/squeak is updated
frequently, however. The Swiki is a communal, editable web site,
maintained by the Squeak community, and powered by Squeak – you'll
read about it later in this book. The Swiki has all kinds of useful
information posted by Squeakers.

To get on the Squeak mailing list, send an email message to squeak-
request@cs.uiuc.edu with a subject of "Subscribe". The Squeak mailing
list runs at about 50 messages per day, and is archived at
http://macos.tuwien.ac.at:9009/Server.home and
http://www.egroups.com/list/squeak/. Topics of discussion on the list
vary widely from the newest of newbie questions, to the arcane details of
the virtual machine, to Squeak Central messages about the future of
Squeak.

A very useful quick reference of Squeak, containing full lists of menus,
keyboard shortcuts, and documentation of key classes, is available on-line
at: http://www.gate.net/~werdna/squeak-qref.html.

Several Squeak specialties, including the Personal Web Server and
Siren maintain their own mailing lists. More information about these is
available on the Swiki.

Updates to the main Squeak image are available from within Squeak,
assuming you have an internet connection. Select help from the World
Menu, then select "update code from server". A series of change sets will
be loaded onto your system (this will likely take several minutes). These
changes contain enhancements and bug fixes that have been added to the
core image by Squeak Central (many of them appear on the Squeak mailing
list first). Announcements of new updates are made to the Squeak mailing
list. Note that updates only affect the image itself, changes to the Virtual
Machine must be downloaded separately.

Happy Squeaking!