Pharo by Example

Andrew P. Black   Stéphane Ducasse
Oscar Nierstrasz   Damien Pollet

with Damien Cassou and Marcus Denker

Version of 2009-09-18

Preface

What is Pharo?

Pharo is a modern, open source, fully-featured implementation of the Smalltalk programming language and environment. Pharo is derived from Squeak[1], a re-implementation of the classic Smalltalk-80 system. Whereas Squeak was developed mainly as a platform for developing experimental educational software, Pharo strives to offer a lean, open-source platform for professional software development, and a robust and stable platform for research and development into dynamic languages and environments. Pharo serves as the reference implementation for the Seaside web development framework.

Pharo resolves some licensing issues with Squeak. Unlike previous versions of Squeak, the Pharo core contains only code that has been contributed under the MIT license. The Pharo project started in March 2008 as a fork of Squeak 3.9, and the first 1.0 beta version was released on July 31, 2009.

Although Pharo removes many packages from Squeak, it also includes numerous features that are optional in Squeak. For example, true type fonts are bundled into Pharo. Pharo also includes support for true block closures. The user interfaces has been simplified and revised.

Pharo is highly portable — even its virtual machine is written entirely in Smalltalk, making it easy to debug, analyze, and change. Pharo is the vehicle for a wide range of innovative projects from multimedia applications and educational platforms to commercial web development environments.

There is an important aspect behind Pharo: Pharo should not just be a copy of the past but really *reinvent* Smalltalk. Big-bang approaches rarely succeed. Pharo will really favor evolutionary and incremental changes. We want to

[1]Dan Ingalls et al., Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97). ACM Press, November 1997 ⟨URL: http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html⟩.

be able to experiment with important new features or libraries. Evolution means that Pharo accepts mistakes and is not aiming for the next perfect solution in one big step — even if we would love it. Pharo will favor small incremental changes but a multitude of them. The success of Pharo depends on the contributions of its community.

Who should read this book?

This book is based on *Squeak by Example*[2], an open-source introduction to Squeak. The book has been liberally adapted and revised to reflect the differences between Pharo and Squeak. This book presents the various aspects of Pharo, starting with the basics, and proceeding to more advanced topics.

This book will not teach you how to program. The reader should have some familiarity with programming languages. Some background with object-oriented programming would be helpful.

This book will introduce the Pharo programming environment, the language and the associated tools. You will be exposed to common idioms and practices, but the focus is on the technology, not on object-oriented design. Wherever possible, we will show you lots of examples. (We have been inspired by Alec Sharp's excellent book on Smalltalk[3].)

There are numerous other books on Smalltalk freely available on the web but none of these focuses specifically on Pharo. See for example: http://stephane.ducasse.free.fr/FreeBooks.html

A word of advice

Do not be frustrated by parts of Smalltalk that you do not immediately understand. You do not have to know everything! Alan Knight expresses this principle as follows[4]:

[2] http://SqueakByExample.org

[3] Alec Sharp, *Smalltalk by Example*. McGraw-Hill, 1997 ⟨URL: http://stephane.ducasse.free.fr/FreeBooks/ByExample/⟩.

[4] http://www.surfscranton.com/architecture/KnightsPrinciples.htm

> **Try not to care.** Beginning Smalltalk programmers often have trouble because they think they need to understand all the details of how a thing works before they can use it. This means it takes quite a while before they can master `Transcript show: 'Hello World'`. One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".

An open book

This book is an open book in the following senses:

- The content of this book is released under the Creative Commons Attribution-ShareAlike (by-sa) license. In short, you are allowed to freely share and adapt this book, as long as you respect the conditions of the license available at the following URL: http://creativecommons.org/licenses/by-sa/3.0/.
- This book just describes the core of Pharo. Ideally we would like to encourage others to contribute chapters on the parts of Pharo that we have not described. If you would like to participate in this effort, please contact us. We would like to see this book grow!

For more details, visit http://pharo-project.org/PharoByExample.

The Pharo community

The Pharo community is friendly and active. Here is a short list of resources that you may find useful:

- http://www.pharo-project.org is the main web site of Pharo.
- http://www.squeaksource.com is the equivalent of SourceForge for Pharo projects. Many optional packages for Pharo live here.

Examples and exercises

We make use of two special conventions in this book.

We have tried to provide as many examples as possible. In particular, there are many examples that show a fragment of code which can be evaluated. We use the symbol $\longrightarrow$ to indicate the result that you obtain when you select an expression and print it:

```
3 + 4   ⟶   7   "if you select 3+4 and 'print it', you will see 7"
```

In case you want to play in Pharo with these code snippets, you can download a plain text file with all the example code from the book's web site: http://pharo-project.org/PharoByExample.

The second convention that we use is to display the icon to indicate when there is something for you to do:

Go ahead and read the next chapter!

Acknowledgments

We would first like to thank the original developers of Squeak for making this amazing Smalltalk development environment available as an open source project.

We would also like to thank Hilaire Fernandes and Serge Stinckwich who allowed us to translate parts of their columns on Smalltalk, and Damien Cassou for contributing the chapter on streams.

We especially thank Alexandre Bergel, Orla Greevy, Fabrizio Perin, Lukas Renggli, Jorge Ressia and Erwann Wernli for their detailed reviews.

We thank the University of Bern, Switzerland, for graciously supporting this open-source project and for hosting the web site of this book.

We also thank the Squeak community for their enthusiastic support of this book project, and for informing us of the errors found in the first edition of this book.

Chapter 1

A quick tour of Pharo

In this chapter we will give you a high-level tour of Pharo to help you get comfortable with the environment. There will be plenty of opportunities to try things out, so it would be a good idea if you have a computer handy when you read this chapter.

We will use this icon: ⓘ to mark places in the text where you should try something out in Pharo. In particular, you will fire up Pharo, learn about the different ways of interacting with the system, and discover some of the basic tools. You will also learn how to define a new method, create an object and send it messages.

1.1 Getting started

Pharo is available as a free download from http://pharo-project.org. There are three parts that you will need to download, consisting of four files (see Figure 1.1).

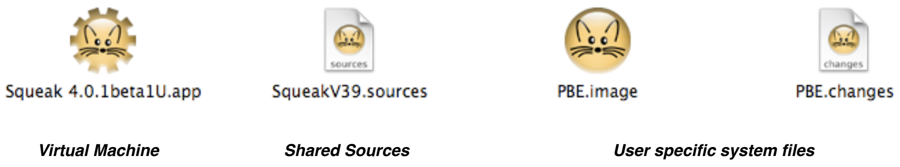


Figure 1.1: The Pharo download files.

1. The *virtual machine* (VM) is the only part of the system that is different for each operating system and processor. Pre-compiled virtual machines

are available for all the major computing environments. In Figure 1.1 we see the VM for the Mac is called *Squeak 4.0.1beta1U.app*.[1]

2. The *sources* file contains the source code for all of the parts of Pharo that don't change very frequently. In Figure 1.1 it is called *SqueakV39.sources*.

3. The current *system image* is a snapshot of a running Pharo system, frozen in time. It consists of two files: an *.image* file, which contains the state of all of the objects in the system (including classes and methods, since they are objects too), and a *.changes* file, which contains a log of all of the changes to the source code of the system. In Figure 1.1, we see that we are using the *PBE* image and changes files. Actually, we will use a slightly different image in this book.

Download and install Pharo on your computer.

We recommend that you use the image provided on the Pharo by Example web page.[2]

Most of the introductory material in this book will work with any version, so if you already have one installed, you may as well continue to use it. However, if you notice differences between the appearance or behaviour of your system and what is described here, do not be surprised.

As you work in Pharo, the image and changes files are modified, so you need to make sure that they are writable. Always keep these two files together. Never edit them directly with a text editor, as Pharo uses them to store the objects you work with and to log the changes you make to the source code. It is a good idea to keep a backup copy of the downloaded image and changes files so you can always start from a fresh image and reload your code.

The *sources* file and the VM can be read-only — they can be shared between different users. All of these files can be placed in the same directory, but it is also possible to put the Virtual Machine and sources file in separate directory where everyone has access to them. Do whatever works best for your style of working and your operating system.

Launching. To start Pharo, do whatever your operating system expects: drag the *.image* file onto the icon of the virtual machine, or double-click the *.image* file, or at the command line type the name of the virtual machine followed by the path to the *.image* file. (When you have multiple VMs installed on your machine the operating system may not automatically pick the right one; in this case it is safer to drag and drop the image onto the virtual machine, or to use the command line.)

[1] Pharo is derived from Squeak 3.9, and presently shares the VM with Squeak.
[2] http://pharo-project.org/PharoByExample

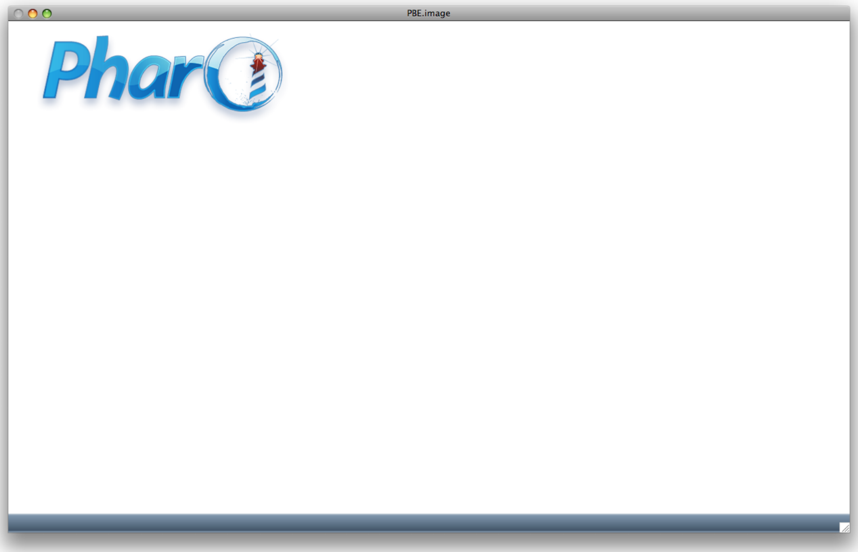


Figure 1.2: A fresh http://pharo-project.org/PharoByExample image.

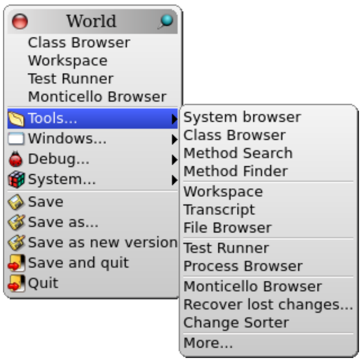
Once Pharo is running, you should see a single large window, possibly containing some open workspace windows (see Figure 1.2), and it's not obvious how to proceed! You might notice a menu bar, but Pharo mainly makes use of context-dependent pop-up menus.

Start Pharo. You can dismiss any open workspaces by clicking on the X *in the top left corner of the workspace window. You can collapse the windows (so that they can be expanded again later) by clicking on the* ○ *in the top-right corner.*

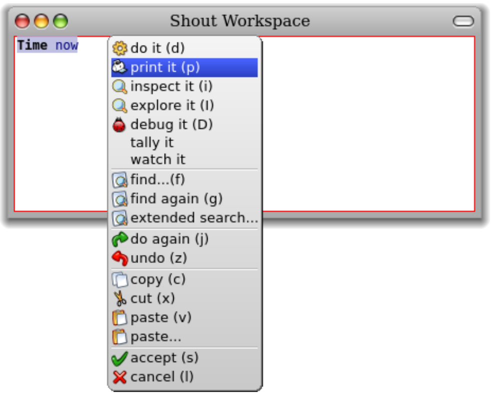
First Interaction. A good place to get started is the world menu shown in Figure 1.3 (a).

Click with the mouse on the background of the main window to show the world menu, then choose Workspace *to create a new workspace.*

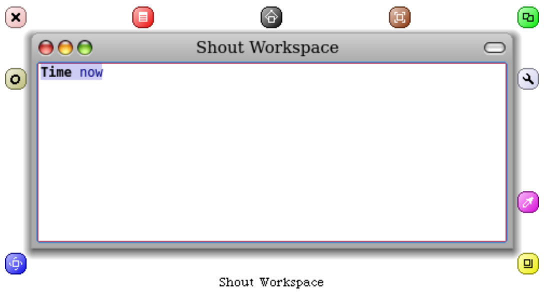
Smalltalk was originally designed for a computer with a three button mouse. If your mouse has fewer than three buttons, you will have to press extra keys while clicking the mouse to simulate the extra buttons. A two-button mouse works quite well with Pharo, but if you have only a single-button mouse, you should seriously consider buying a two-button mouse with a clickable scroll wheel: it will make working with Pharo much more pleasant.



(a) The world menu



(b) The contextual menu



(c) The morphic halo

Figure 1.3: The world menu (brought up by clicking), a contextual menu (action-clicking), and a morphic halo (meta-clicking).

Pharo avoids terms like "left mouse click" because different computers, mice, keyboards and personal configurations mean that different users will need to press different physical buttons to achieve the same effect. Originally Smalltalk introduced colours to stand for the different mouse buttons.[3] Since many users will use various modifiers keys (*control*, *ALT*, *meta* etc.) to achieve the same effect, we will instead use the following terms:

click: this is the most often used mouse button, and is normally equivalent to clicking a single-mouse button without any modifier key; click on the image to bring up the "World" menu (Figure 1.3 (a)).

action-click: this is the next most used button; it is used to bring up a contextual menu, that is, a menu that offers different sets of actions depending

[3]The button colours were *red*, *yellow* and *blue*. The authors of this book could never remember which colour referred to which button.

on where the mouse is pointing; see Figure 1.3 (b). If you do not have a multi-button mouse, then normally you will configure the *control* modifier key to action-click with the mouse button.

meta-click: Finally, you may meta-click on any object displayed in the image to activate the "morphic halo", an array of handles that are used to perform operations on the on-screen objects themselves, such as rotating them or resizing them; see Figure 1.3 (c).[4] If you let the mouse linger over a handle, a help balloon will explain its function. In Pharo, how you meta-click depends on your operating system: either you must hold SHIFT *ctrl* or SHIFT *alt* while clicking.

Type Time now *in the workspace. Now action-click in the workspace. Select* print it.

We recommend that right-handed people configure their mouse to click with the left button, action-click with the right button, and meta-click with the clickable scroll wheel, if one is available. If you are using a Macintosh without a second mouse button, you can simulate one by holding down the ⌘ key while clicking the mouse. However, if you are going to be using Pharo at all often, we recommend investing in a mouse with at least two buttons.

You can configure your mouse to work the way you want by using the preferences of your operating system and mouse driver. Pharo has some preferences for customising the mouse and the meta keys on your keyboard. In the preference browser (System . . . ▷ Preferences . . . ▷ Preference Browser. . .), the keyboard category contains an option swapControlAndAltKeys that switches the action-click and meta-click functions. There are also options to duplicate the various command keys.

1.2 The World menu

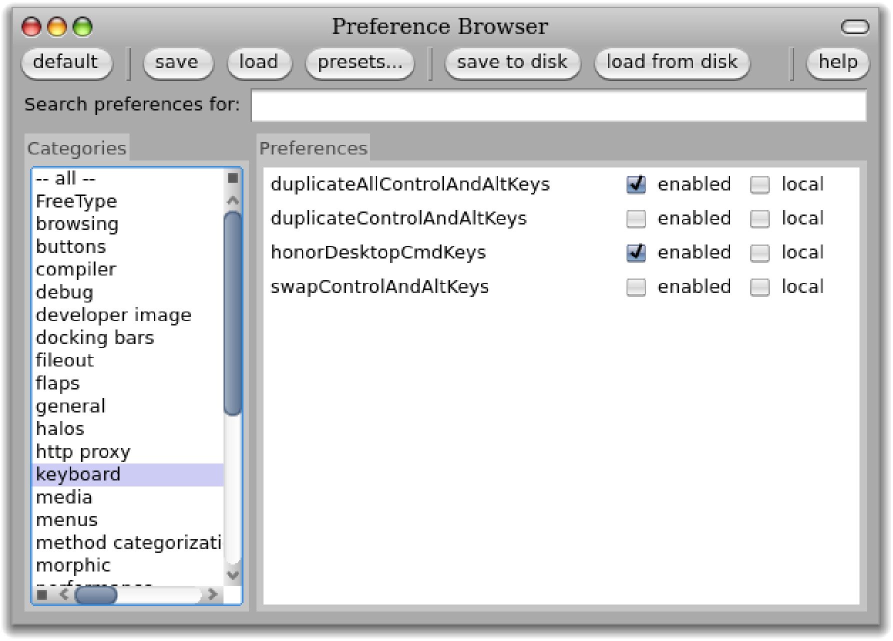
Click again on the Pharo background.

You will see the World menu again. Most Pharo menus are not modal; you can leave them on the screen for as long as you wish by clicking the push pin icon in the top-right corner. Do this.

The world menu provides you a simple means to access many of the tools that Pharo offers.

Have a closer look at the World *and* Tools . . . *menus. (Figure 1.3 (a))*

[4]Note that the morphic handles are usually inactive in Pharo, but you can turn them on using the Preferences Browser, which we will see shortly.



duplicateAllControlAndAltKeys	✓ enabled	☐ local
duplicateControlAndAltKeys	☐ enabled	☐ local
honorDesktopCmdKeys	✓ enabled	☐ local
swapControlAndAltKeys	☐ enabled	☐ local

Figure 1.4: The Preference Browser.

You will see a list of several of the core tools in Pharo, including the browser and the workspace. We will encounter most of them in the coming chapters.

1.3 Sending messages

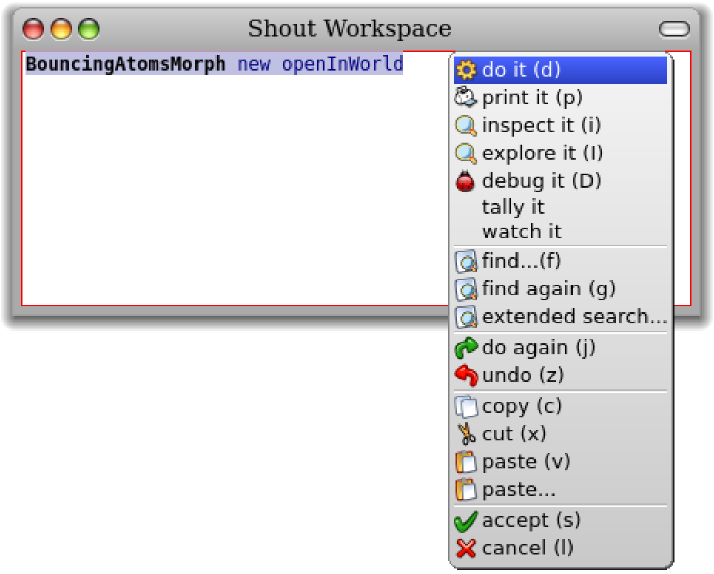
Open a workspace. Type in the following text:

```
BouncingAtomsMorph new openInWorld
```

Now action-click. A menu should appear. Select do it (d)*. (See Figure 1.5.)*

A window containing a large number of bouncing atoms should open in the top left of the Pharo image.

You have just evaluated your first Smalltalk expression! You just sent the message new to the BouncingAtomsMorph class, resulting in a new BouncingAtomsMorph instance, followed by the message openInWorld to this instance. The BouncingAtomsMorph class decided what to do with the new message, that



```
BouncingAtomsMorph new openInWorld
```

Figure 1.5: “Doing” an expression

is, it looked up its *methods* for handling new message and reacted appropriately. Similarly the BouncingAtomsMorph instance looked up its method for responding to openInWorld and took appropriate action.

If you talk to Smalltalkers for a while, you will quickly notice that they generally do not use expressions like “call an operation” or “invoke a method”, but instead they will say “send a message”. This reflects the idea that objects are responsible for their own actions. You never *tell* an object what to do — instead you politely *ask* it to do something by sending it a message. The object, not you, selects the appropriate method for responding to your message.

1.4 Saving, quitting and restarting a Pharo session

Now click on the bouncing atoms window and drag it anywhere you like. You now have the demo “in hand”. Put it down by clicking anywhere.

Select World ▷ Save as ... *, enter the name “myPharo”, and click on the* OK *button. Now select* World ▷ Save and quit.

Now if you go to the location where the original image and changes files were, you will find two new files called “myPharo.image” and

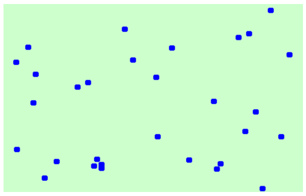


Figure 1.6: A BouncingAtomsMorph.

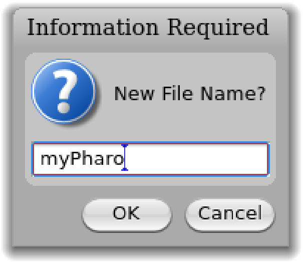


Figure 1.7: The save as ... dialogue.

"myPharo.changes" that represent the working state of the Pharo image at the moment before you told Pharo to Save and quit. If you wish, you can move these two files anywhere that you like on your disk, but if you do so you may (depending on your operating system) need to also move, copy or link to the virtual machine and the *.source* file.

Start up Pharo from the newly created "myPharo.image" file.

Now you should find yourself in precisely the state you were when you quit Pharo. The BouncingAtomsMorph is there again and it continues to move from where it was when you left it.

When you start Pharo for the first time, the Pharo virtual machine loads the image file that you provide. This file contains a snapshot of a large number of objects, including a vast amount of pre-existing code and a large number of programming tools (all of which are objects). As you work with Pharo, you will send messages to these objects, you will create new objects, and some of these objects will die and their memory will be reclaimed (*i.e.*, garbage-collected).

When you quit Pharo, you will normally save a snapshot that contains all of your objects. If you save normally, you will overwrite your old image file with the new snapshot. Alternatively you may save the image under a new name, as we just did.

In addition to the *.image* file, there is also a *.changes* file. This file contains a log of all the changes to the source code that you have made using the standard tools. Most of the time you do not need to worry about this file at all. As we shall see, however, the *.changes* file can be very useful for recovering from errors, or replaying lost changes. More about this later!

The image that you have been working with is a descendant of the original Smalltalk-80 image created in the late 1970s. Some of these objects have been around for decades!

You might think that the image is the key mechanism for storing and managing software projects, but you would be wrong. As we shall see very soon, there are much better tools for managing code and sharing software developed by teams. Images are very useful, but you should learn to be very cavalier about creating and throwing away images, since tools like Monticello offer much better ways to manage versions and share code amongst developers.

Blue-click on the BouncingAtomsMorph.[5]

You will see a collection of colored dots that are collectively called the BouncingAtomsMorph's morphic halo. Each dot is called a *handle*. Click in the pink handle containing the cross; the BouncingAtomsMorph should go away.

1.5 Workspaces and Transcripts

Close all open windows. Open a transcript and a workspace. (The transcript can be opened from the World ▷ Tools ... *submenu.)*

Position and resize the transcript and workspace windows so that the workspace just overlaps the transcript.

You can resize windows either by dragging one of the corners, or by meta-clicking the window to bring up the morphic handles, and dragging the yellow, bottom right handle.

At any time only one window is active; it is in front and has its border highlighted.

The transcript is an object that is often used for logging system messages. It is a kind of "system console".

Workspaces are useful for typing snippets of Smalltalk code that you would like to experiment with. You can also use workspaces simply for typing arbitrarily text that you would like to remember, such as to-do lists or instructions for anyone who will use your image. Workspaces are often used to hold documentation about a captured image, as is the case with the standard image that we downloaded earlier (see Figure 1.2).

Type the following text into the workspace:

```
Transcript show: 'hello world'; cr.
```

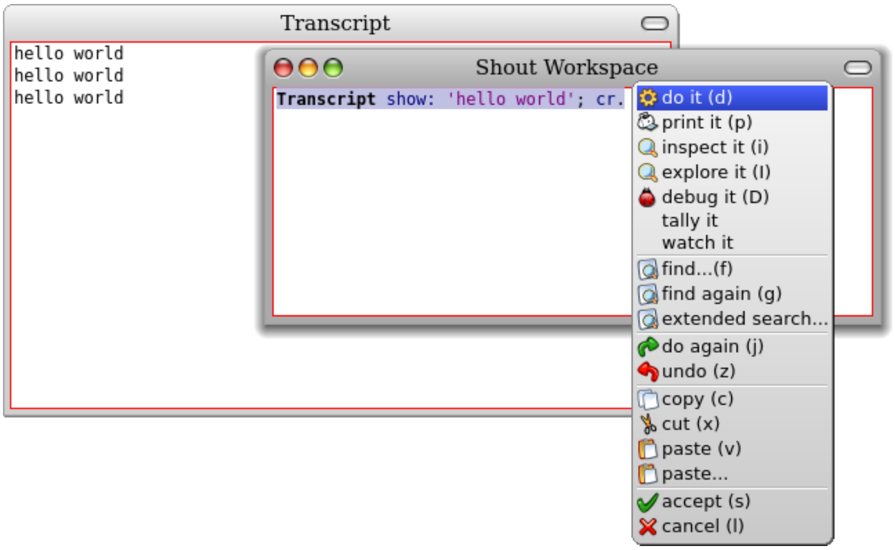
Try double-clicking in the workspace at various points in the text you have just typed. Notice how an entire word, entire string, or the whole text is

[5]Remember, you may have to set the halosEnabled option in the Preferences Browser.

selected, depending on whether you click within a word, at the end of the string, or at the end of the entire expression.

Select the text you have typed and action-click. Select do it (d) .

Notice how the text "hello world" appears in the transcript window (Figure 1.8). Do it again. (The (d) in the menu item do it (d) tells you that the keyboard shortcut to *do it* is CMD–d. More on this in the next section!)



```
hello world
hello world
hello world
```

```
Transcript show: 'hello world'; cr.
```

Figure 1.8: Overlapping windows. The workspace is active.

1.6 Keyboard shortcuts

If you want to evaluate an expression, you do not always have to action-click. Instead, you can use keyboard shortcuts. These are the parenthesized expressions in the menu. Depending on your platform, you may have to press one of the modifier keys (control, alt, command, or meta). (We will indicate these generically as CMD–*key*.)

Evaluate the expression in the workspace again, but using the keyboard shortcut: CMD–*d*.

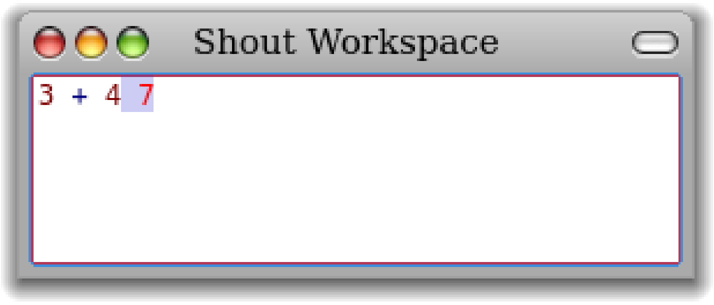
In addition to do it , you will have noticed print it , inspect it and explore it . Let's have a quick look at each of these.

Type the expression 3 + 4 *into the workspace. Now* do it *with the keyboard shortcut.*

Do not be surprised if you saw nothing happen! What you just did is send the message + with argument 4 to the number 3. Normally the result 7 will have been computed and returned to you, but since the workspace did not know what to do with this answer, it simply threw the answer away. If you want to see the result, you should print it instead. print it actually compiles the expression, executes it, sends the message printString to the result, and displays the resulting string.

Select 3+4 *and* print it *(CMD–p).*

This time we see the result we expect (Figure 1.9).



```
3 + 4 7
```

Figure 1.9: "Print it" rather than "do it".

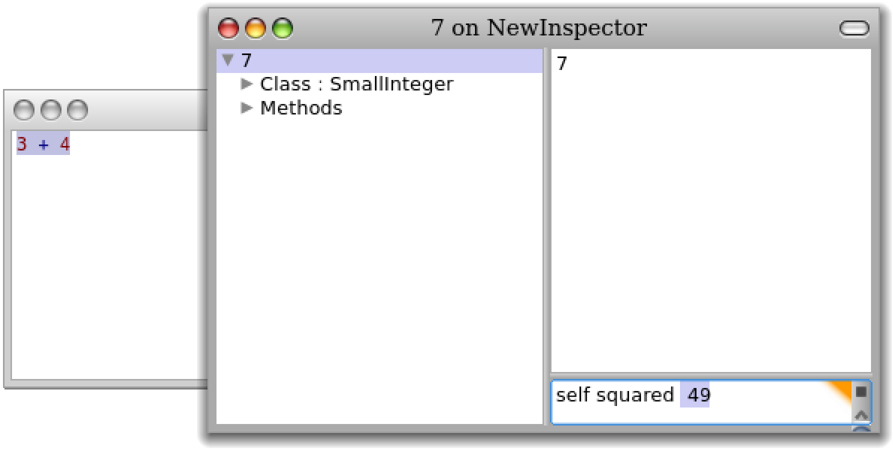
```
3 + 4   ⟶   7
```

We use the notation ⟶ as a convention in this book to indicate that a particular Pharo expression yields a given result when you print it.

Delete the highlighted text "7" (Pharo should have selected it for you, so you can just press the delete key). Select 3+4 *again and this time* inspect it *(CMD–i).*

Now you should see a new window, called an *inspector*, with the heading SmallInteger: 7 (Figure 1.10). The inspector is an extremely useful tool that will allow you to browse and interact with any object in the system. The title tells us that 7 is an instance of the class SmallInteger. The left panel allows us to browse the instance variables of an object, the values of which are shown in the right panel. The bottom panel can be used to write expressions to send messages to the object.

Type self squared *in the bottom panel of the inspector on* 7 *and* print it.



```
3 + 4
```

```
▼ 7
  ▶ Class : SmallInteger
  ▶ Methods
```

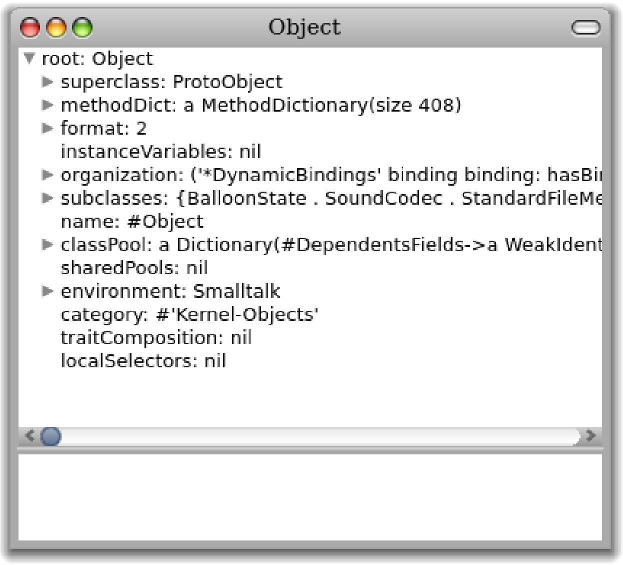
```
7
```

```
self squared 49
```

Figure 1.10: Inspecting an object.

Close the inspector. Type the expression Object *in a workspace and this time* explore it *(*CMD–I*, uppercased i).*

This time you should see a window labelled Object containing the text ▷ root: Object. Click on the triangle to open it up (Figure 1.11).



```
▼ root: Object
  ▶ superclass: ProtoObject
  ▶ methodDict: a MethodDictionary(size 408)
  ▶ format: 2
    instanceVariables: nil
  ▶ organization: ('*DynamicBindings' binding binding: hasBi
  ▶ subclasses: {BalloonState . SoundCodec . StandardFileMe
    name: #Object
  ▶ classPool: a Dictionary(#DependentsFields->a WeakIdent
    sharedPools: nil
  ▶ environment: Smalltalk
    category: #'Kernel-Objects'
    traitComposition: nil
    localSelectors: nil
```

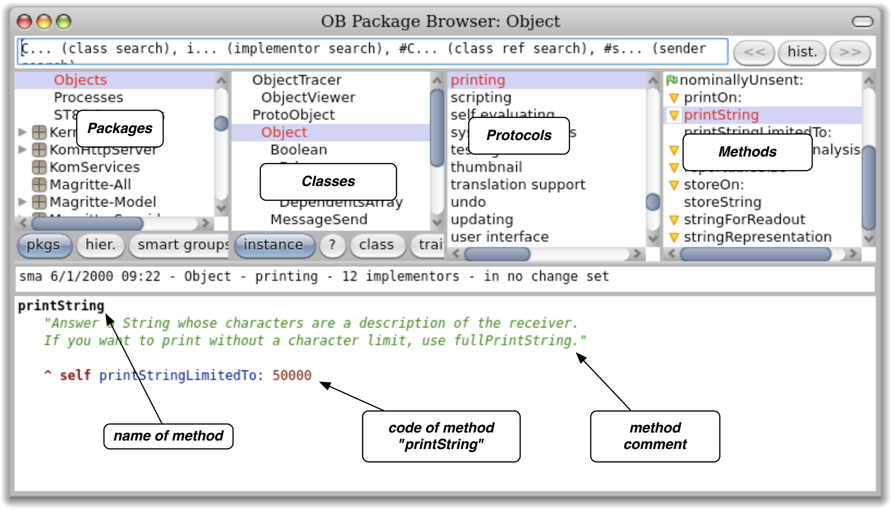
Figure 1.11: Exploring Object.

The explorer is similar to the inspector, but it offers a tree view of a complex object. In this case the object we are looking at is the Object class. We can see directly all the information stored in this class, and we can easily navigate to all its parts.

1.7 The Class Browser

The class browser[6] is one of the key tools used for programming. As we shall see, there are several interesting browsers available for Pharo, but this is the basic one you will find in any image.

Open a browser by selecting World ▷ Class browser.[7]



```
sma 6/1/2000 09:22 - Object - printing - 12 implementors - in no change set
```

```
printString
	"Answer a String whose characters are a description of the receiver.
	If you want to print without a character limit, use fullPrintString."

	^ self printStringLimitedTo: 50000
```

Figure 1.12: The browser showing the printString method of class object.

We can see a browser in Figure 1.12. The title bar indicates that we are browsing the class Object.

When the browser first opens, all panes are empty but the leftmost one. This first pane lists all known *packages*, which contain groups of related classes.

Click on the Kernel *package.*

[6]Confusingly, this is variously referred to as the "system browser" or the "code browser". Pharo uses the OmniBrowser implementation of the browser, which may also be variously known as "OB" or the "Package browser". In this book we will simply use the term "browser", or, in case of ambiguity, the "class browser".

[7]If the browser you get does not look like the one shown in Figure 1.12, then you may need to change the default browser. See FAQ 5, p. 318.

This causes the second pane to show a list of all of the classes in the selected package.

Select the class Object.

Now the remaining two panes will be filled with text. The third pane displays the *protocols* of the currently selected class. These are convenient groupings of related methods. If no protocol is selected you should see all methods in the fourth pane.

Select the printing *protocol.*

You may have to scroll down to find it. Now you will see in the fourth pane only methods related to printing.

Select the printString *method.*

Now we see in the bottom pane the source code of the printString method, shared by all objects in the system (except those that override it).

1.8 Finding classes

There are several ways to find a class in Pharo. The first, as we have just seen above, is to know (or guess) what category it is in, and to navigate to it using the browser.

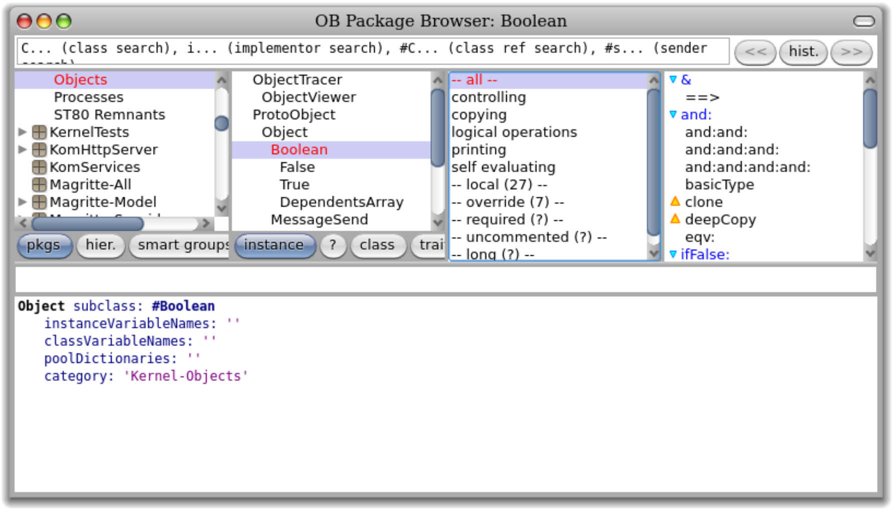
A second way is to send the browse message to the class, asking it to open a browser on itself. Suppose we want to browse the class Boolean.

Type Boolean browse *into a workspace and* do it.

A browser will open on the Boolean class (Figure 1.13). There is also a keyboard shortcut CMD–b (browse) that you can use in any tool where you find a class name; select the name and type CMD–b.

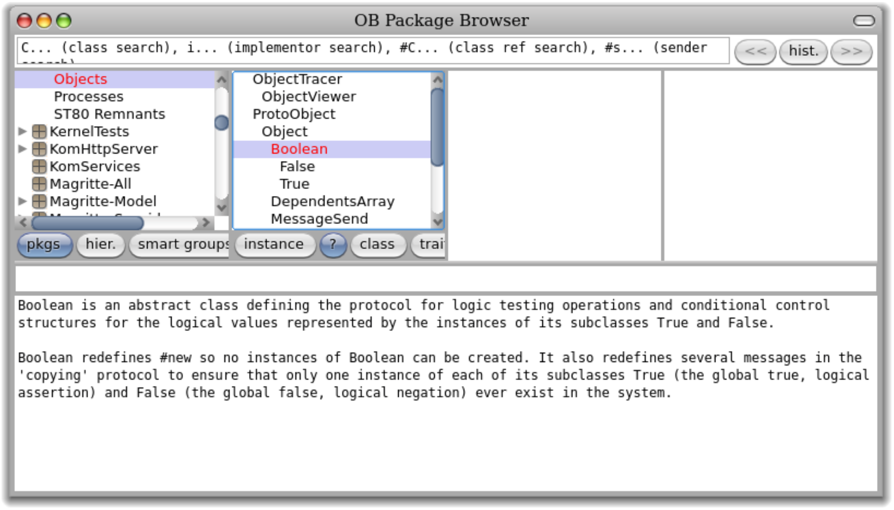
Use the keyboard shortcut to browse the class Boolean.

Notice that when the Boolean class is selected but no protocol or method is selected, instead of the source code of a method, we see a *class definition* (Figure 1.13). This is nothing more than an ordinary Smalltalk message that is sent to the parent class, asking it to create a subclass. Here we see that the class Object is being asked to create a subclass named Boolean with no instance variables, class variables or "pool dictionaries", and to put the class Boolean in the *Kernel-Objects* category. If you click on the ? at the bottom of the class pane, you can see the class comment in a dedicated pane (see Figure 1.14).



```
Object subclass: #Boolean
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Objects'
```

Figure 1.13: The browser showing the definition of class Boolean.



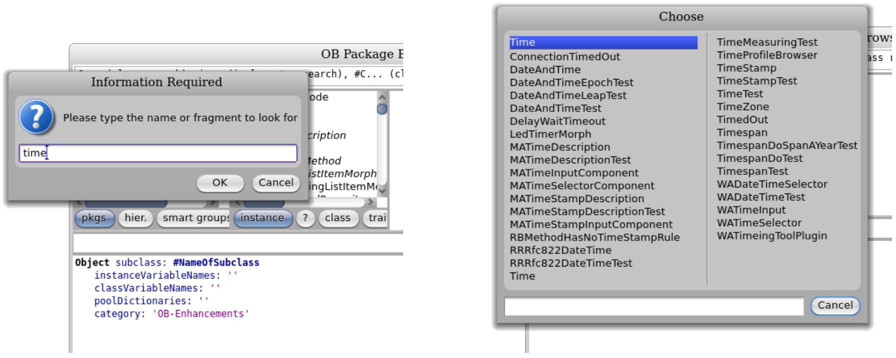
```
Boolean is an abstract class defining the protocol for logic testing operations and conditional control
structures for the logical values represented by the instances of its subclasses True and False.

Boolean redefines #new so no instances of Boolean can be created. It also redefines several messages in the
'copying' protocol to ensure that only one instance of each of its subclasses True (the global true, logical
assertion) and False (the global false, logical negation) ever exist in the system.
```

Figure 1.14: The class comment for Boolean.

Often, the fastest way to find a class is to search for it by name. For example, suppose that you are looking for some unknown class that represents dates and times.

Put the mouse in the package pane of the browser and type CMD–*f, or select* find class . . . (f) *by action-clicking. Type "time" in the dialog box and accept it.*

You will be presented with a list of classes whose names contain "time" (see Figure 1.15). Choose one, say, Time, and the browser will show it, along with a class comment that suggests other classes that might be useful. If you want to browse one of the others, select its name (in any text pane), and type CMD–b.



Please type the name or fragment to look for

time

```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'OB-Enhancements'
```

Time
ConnectionTimedOut
DateAndTime
DateAndTimeEpochTest
DateAndTimeLeapTest
DateAndTimeTest
DelayWaitTimeout
LedTimerMorph
MATimeDescription
MATimeDescriptionTest
MATimeInputComponent
MATimeSelectorComponent
MATimeStampDescription
MATimeStampDescriptionTest
MATimeStampInputComponent
RBMethodHasNoTimeStampRule
RRRfc822DateTime
RRRfc822DateTimeTest
Time
TimeMeasuringTest
TimeProfileBrowser
TimeStamp
TimeStampTest
TimeTest
TimeZone
TimedOut
Timespan
TimespanDoSpanAYearTest
TimespanDoTest
TimespanTest
WADateTimeSelector
WADateTimeTest
WATimeInput
WATimeSelector
WATimeingToolPlugin

Figure 1.15: Searching for a class by name.

Note that if you type the complete (and correctly capitalized) name of a class in the find dialog, the browser will go directly to that class without showing you the list of options.

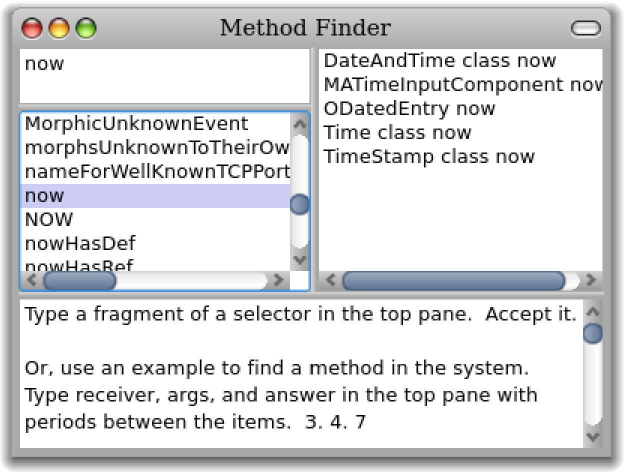
1.9 Finding methods

Sometimes you can guess the name of a method, or at least part of the name of a method, more easily than the name of a class. For example, if you are interested in the current time, you might expect that there would be a method called "now", or containing "now" as a substring. But where might it be? The *method finder* can help you.

Select World ▷Tools ... ▷Method finder. *Type "now" in the top left pane, and* accept *it (or just press the* RETURN *key).*

The method finder will display a list of all the method names that contain the substring "now". To scroll to now itself, move the cursor to the list and type "n"; this trick works in all scrolling windows. Select "now" and the right-hand pane shows you the classes that define a method with this name, as shown in Figure 1.16. Selecting any one of them will open a browser on it.

At other times you may have a good idea that a method exists, but will have no idea what it might be called. The method finder can still help! For example, suppose that you would like to find a method that turns a string into upper case, for example, it would translate 'eureka' into 'EUREKA'.



```
now
```

```
MorphicUnknownEvent
morphsUnknownToTheirOw
nameForWellKnownTCPPort
now
NOW
nowHasDef
nowHasRef
```

```
DateAndTime class now
MATimeInputComponent nov
ODatedEntry now
Time class now
TimeStamp class now
```

Type a fragment of a selector in the top pane. Accept it.

Or, use an example to find a method in the system.
Type receiver, args, and answer in the top pane with
periods between the items. 3. 4. 7

Figure 1.16: The method finder showing all classes defining a method named now.

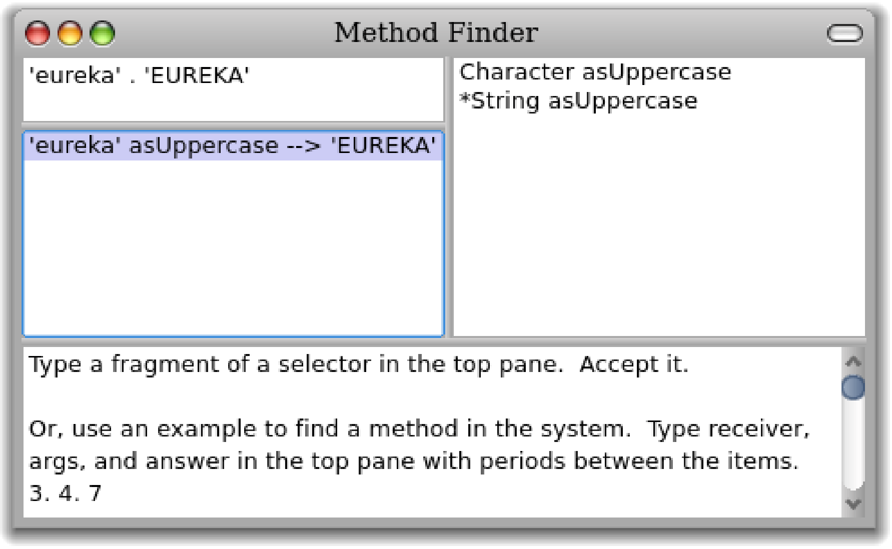
Type 'eureka' . 'EUREKA' *into the method finder and press the* RETURN *key, as shown in Figure 1.17.*

The method finder will suggest a method that does what you want.[8]

An asterisk at the beginning of a line in the right pane of the method finder indicates that this method is the one that was actually used to obtain the requested result. So, the asterisk in front of String asUppercase lets us know that the method asUppercase defined on the class String was executed and returned the result we wanted. The methods that do not have an asterisk are just the other methods that have the same name as the ones that returned the expected result. So Character»asUppercase was not executed on our example, because 'eureka' is not a Character object.

You can also use the method finder for methods with arguments; for example, if you are looking for a method that will find the greatest common factor of two integers, you might try 25. 35. 5 as an example. You can also give the method finder multiple examples to narrow the search space; the help text in the bottom pane explains how.

[8]If a window pops up with a warning about a deprecated method, don't panic — the method finder is simply trying out all likely candidates, including deprecated methods. Just click Proceed.



```
'eureka' . 'EUREKA'
'eureka' asUppercase --> 'EUREKA'
Character asUppercase
*String asUppercase
```

Type a fragment of a selector in the top pane. Accept it.

Or, use an example to find a method in the system. Type receiver, args, and answer in the top pane with periods between the items. 3. 4. 7

Figure 1.17: Finding a method by example.

1.10 Defining a new method

The advent of Test Driven Development[9] (TDD) has changed the way that we write code. The idea behind TDD is that we write a test that defines the desired behaviour of our code *before* we write the code itself. Only then do we write the code that satisfies the test.

Suppose that our assignment is to write a method that "says something loudly and with emphasis". What exactly could that mean? What would be a good name for such a method? How can we make sure that programmers who may have to maintain our method in the future have an unambiguous description of what it should do? We can answer all of these questions by giving an example:

> When we send the message shout to the string "Don't panic" the result should be "DON'T PANIC!".

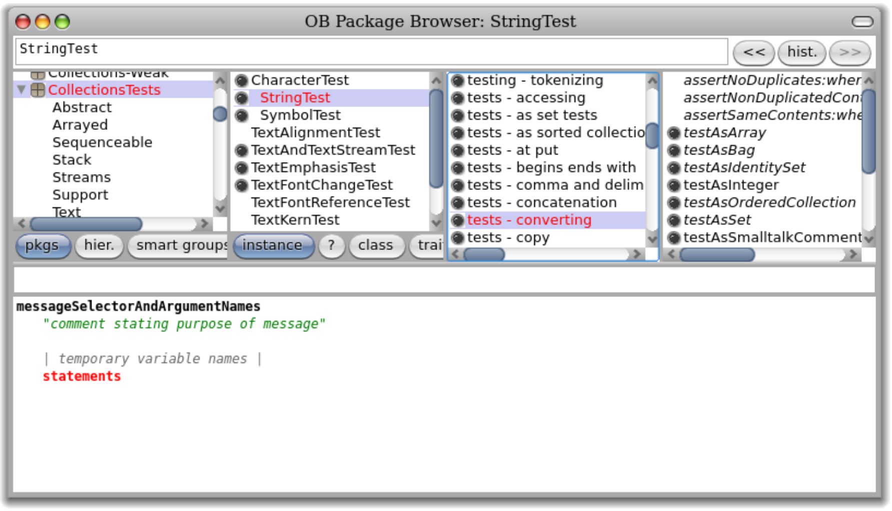
To make this example into something that the system can use, we turn it into a test method:

Method 1.1: *A test for a shout method*

```
testShout
   self assert: ('Don''t panic' shout = 'DON''T PANIC!')
```

[9]Kent Beck, *Test Driven Development: By Example*. Addison-Wesley, 2003, ISBN 0–321–14653–0.

How do we create a new method in Pharo? First, we have to decide which class the method should belong to. In this case, the shout method that we are testing will go in class String, so the corresponding test will, by convention, go in a class called StringTest.



```
messageSelectorAndArgumentNames
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

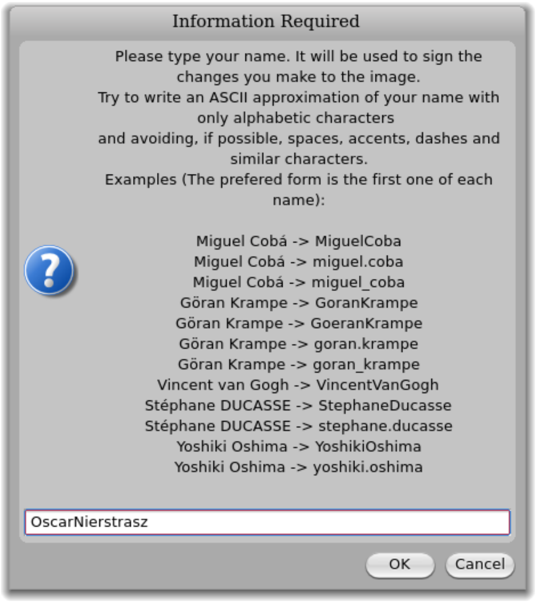
Figure 1.18: The new method template in class StringTest.

Open a browser on the class StringTest*, and select an appropriate protocol for our method, in this case* tests - converting*, as shown in Figure 1.18. The highlighted text in the bottom pane is a template that reminds you what a Smalltalk method looks like. Delete this and enter the code from method 1.1.*

Once you have typed the text into the browser, notice that the bottom pane is outlined in red. This is a reminder that the pane contains unsaved changes. So select accept (s) by action-clicking in the bottom pane, or just type CMD–s, to compile and save your method.

If this is the first time you have accepted any code in your image, you will likely be prompted to enter your name. Since many people have contributed code to the image, it is important to keep track of everyone who creates or modifies methods. Simply enter your first and last names, without any spaces, or separated by a dot.

Because there is as yet no method called shout, the browser will ask you to confirm that this is the name that you really want — and it will suggest some other names that you might have intended (Figure 1.20). This can be quite useful if you have merely made a typing mistake, but in this case, we really *do* mean shout, since that is the method we are about to create, so we have to confirm this by selecting the first option from the menu of choices, as shown in Figure 1.20.



Please type your name. It will be used to sign the changes you make to the image.
Try to write an ASCII approximation of your name with only alphabetic characters
and avoiding, if possible, spaces, accents, dashes and similar characters.
Examples (The prefered form is the first one of each name):

```
Miguel Cobá -> MiguelCoba
Miguel Cobá -> miguel.coba
Miguel Cobá -> miguel_coba
Göran Krampe -> GoranKrampe
Göran Krampe -> GoeranKrampe
Göran Krampe -> goran.krampe
Göran Krampe -> goran_krampe
Vincent van Gogh -> VincentVanGogh
Stéphane DUCASSE -> StephaneDucasse
Stéphane DUCASSE -> stephane.ducasse
Yoshiki Oshima -> YoshikiOshima
Yoshiki Oshima -> yoshiki.oshima
```

OscarNierstrasz

Figure 1.19: Entering your name.

Run your newly created test: open the SUnit TestRunner *from the* World *menu.*

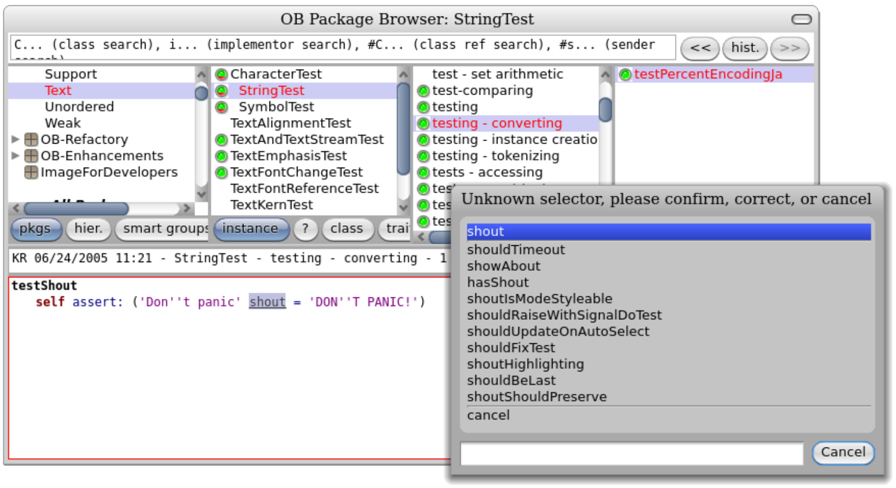
The leftmost two panes are a bit like the top panes in the browser. The left pane contains a list of categories, but it's restricted to those categories that contain test classes.

Select CollectionsTests-Text *and the pane to the right will show all of the test classes in that category, which includes the class* StringTest*. The names of the classes are already selected, so click* Run Selected *to run all these tests.*

You should see a message like that shown in Figure 1.21, which indicates that there was an error in running the tests. The list of tests that gave rise to errors is shown in the bottom right pane; as you can see, StringTest»#testShout is the culprit. (Note that StringTest>>#testShout is the Smalltalk way of identifying the testShout method of the StringTest class.) If you click on that line of text, the erroneous test will run again, this time in such a way that you see the error happen: "MessageNotUnderstood: ByteString»shout".

The window that opens with the error message is the Smalltalk debugger (see Figure 1.22). We will look at the debugger and how to use it in Chapter 6.

The error is, of course, exactly what we expected: running the test gen-



```
KR 06/24/2005 11:21 - StringTest - testing - converting - 1
```

```
testShout
    self assert: ('Don''t panic' shout = 'DON''T PANIC!')
```

Unknown selector, please confirm, correct, or cancel

```
shout
shouldTimeout
showAbout
hasShout
shoutIsModeStyleable
shouldRaiseWithSignalDoTest
shouldUpdateOnAutoSelect
shouldFixTest
shoutHighlighting
shouldBeLast
shoutShouldPreserve
cancel
```

Figure 1.20: Accepting the StringTest method testShout.

erates an error because we haven't yet written a method that tells strings how to shout. Nevertheless, it's good practice to make sure that the test fails because this confirms that we have set up the testing machinery correctly and that the new test is actually being run. Once you have seen the error, you can Abandon the running test, which will close the debugger window. Note that often with Smalltalk you can define the missing method using the Create button, edit the newly-created method in the debugger, and then Proceed with the test.

Now let's define the method that will make the test succeed!

Select class String *in the browser, select the* converting *protocol, type the text in method 1.2 over the method creation template, and* accept *it. (Note: to get a ↑, type ^).*

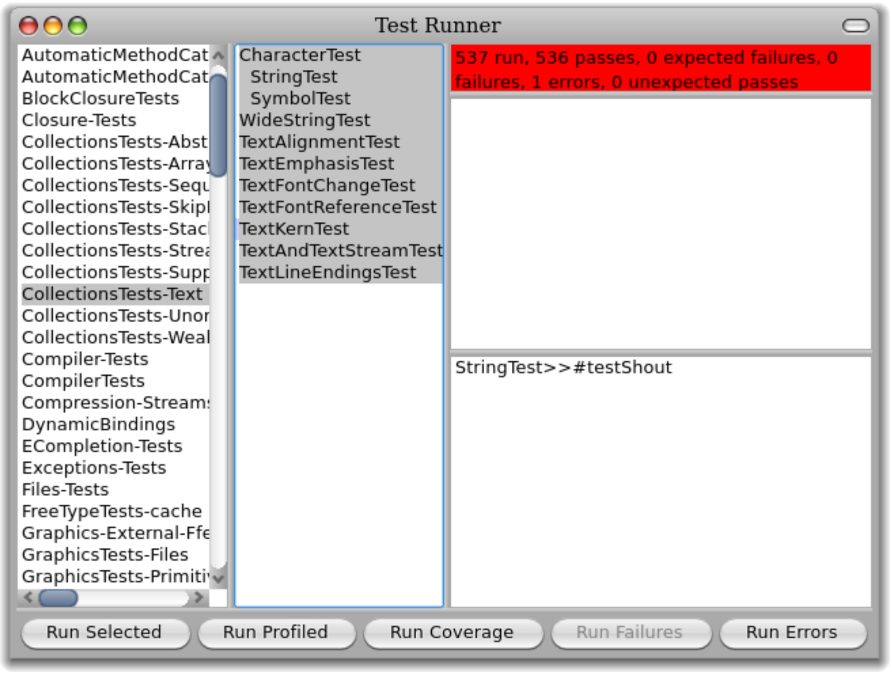
Method 1.2: *The shout method*

```
shout
  ↑ self asUppercase, '!'
```

The comma is the string concatenation operation, so the body of this method appends an exclamation mark to an upper-case version of whatever String object the shout message was sent to. The ↑ tells Pharo that the expression that follows is the answer to be returned from the method, in this case the new concatenated string.

Does this method work? Let's run the tests and see.

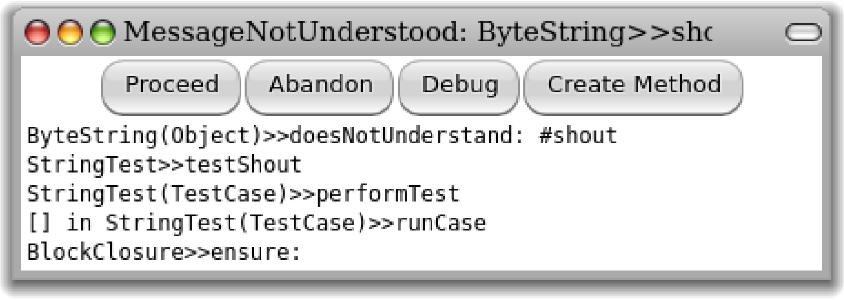
Click on Run Selected *again in the test runner, and this time you should see a*



```
537 run, 536 passes, 0 expected failures, 0 failures, 1 errors, 0 unexpected passes
```

```
StringTest>>#testShout
```

Figure 1.21: Running the String tests.

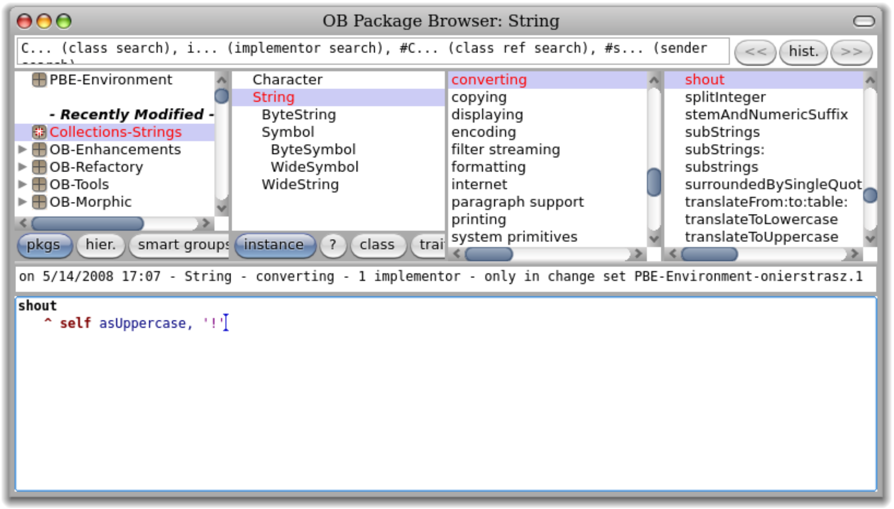


```
ByteString(Object)>>doesNotUnderstand: #shout
StringTest>>testShout
StringTest(TestCase)>>performTest
[] in StringTest(TestCase)>>runCase
BlockClosure>>ensure:
```

Figure 1.22: The (pre-)debugger.

green bar and text indicating that all of the tests ran with no failures and no errors.

When you get to a green bar[10], it's a good idea to save your work and take a break. So do that right now!



```
on 5/14/2008 17:07 - String - converting - 1 implementor - only in change set PBE-Environment-onierstrasz.1
```

```
shout
   ^ self asUppercase, '!'
```

Figure 1.23: The shout method defined on class String.

1.11 Chapter summary

This chapter has introduced you to the Pharo environment and shown you how to use some of the major tools, such as the browser, the method finder, and the test runner. You have also seen a little of Pharo's syntax, even though you may not understand it all yet.

- A running Pharo system consists of a *virtual machine*, a *sources* file, and *image* and *changes* files. Only these last two change, as they record a snapshot of the running system.
- When you restore a Pharo image, you will find yourself in exactly the same state — with the same running objects — that you had when you last saved that image.
- Pharo is designed to work with a three-button mouse to click, action-click or meta-click. If you don't have a three-button mouse, you can use modifier keys to obtain the same effect.
- You click on the Pharo background to bring up the *World menu* and launch various tools.
- A *workspace* is a tool for writing and evaluating snippets of code. You can also use it to store arbitrary text.
- You can use keyboard shortcuts on text in the workspace, or any other tool, to evaluate code. The most important of these are do it

(CMD–d), print it (CMD–p), inspect it (CMD–i), explore it (CMD–I) and browse it (CMD–b).

- The *browser* is the main tool for browsing Pharo code, and for developing new code.
- The *test runner* is a tool for running unit tests. It also supports Test Driven Development.

Chapter 2

A first application

In this chapter, we will develop a simple game: Lights Out.[1] Along the way we will demonstrate most of the tools that Pharo programmers use to construct and debug their programs, and show how programs are exchanged with other developers. We will see the browser, the object inspector, the debugger and the Monticello package browser. Development in Smalltalk is efficient: you will find that you spend far more time actually writing code and far less managing the development process. This is partly because the Smalltalk language is very simple, and partly because the tools that make up the programming environment are very well integrated with the language.

2.1 The Lights Out game

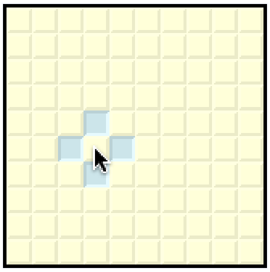


Figure 2.1: The Lights Out game board. The user has just clicked the mouse as shown by the cursor.

To show you how to use Pharo's programming tools, we will build a simple game called *Lights Out*. The game board is shown in Figure 2.1; it

[1] http://en.wikipedia.org/wiki/Lights_Out_(game)

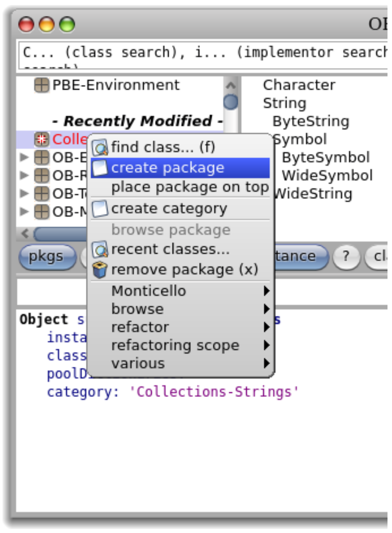
consists of rectangular array of light yellow *cells*. When you click on one of the cells with the mouse, the four surrounding cells turn blue. Click again, and they toggle back to light yellow. The object of the game is to turn blue as many cells as possible.

The Lights Out game shown in Figure 2.1 is made up of two kinds of objects: the game board itself, and 100 individual cell objects. The Pharo code to implement the game will contain two classes: one for the game and one for the cells. We will now show you how to define these classes using the Pharo programming tools.

2.2 Creating a new Package

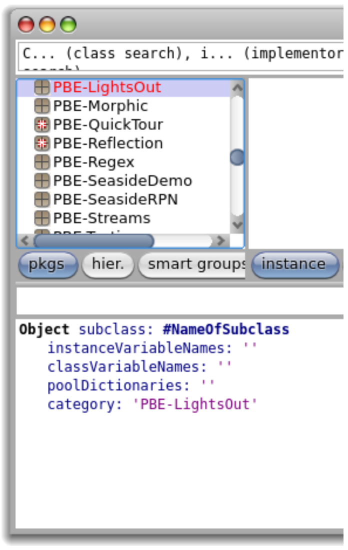
We have already seen the browser in Chapter 1, where we learned how to navigate to classes and methods, and saw how to define new methods. Now we will see how to create packages, categories and classes.

Open a browser and action-click in the package pane. Select create package.[2]



```
Object s
    insta
    class
    poolD
    category: 'Collections-Strings'
```

Figure 2.2: Adding a package.



```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE-LightsOut'
```

Figure 2.3: The class template.

Type the name of the new package (we will use *PBE-LightsOut*) in the dialog box and click accept (or just press the return key); the new package is created,

[2]We are assuming that the Package Browser is installed as the default browser, which should normally be the case. If the browser you get does not look like the one shown in Figure 2.2, then you may need to change the default browser. See FAQ 5, p. 318.

and positioned alphabetically in the list of packages.

2.3 Defining the class LOCell

As yet there are of course no classes in the new package. However, the main editing pane displays a template to make it easy to create a new class (see Figure 2.3).

This template shows us a Smalltalk expression that sends a message to a class called Object, asking it to create a subclass called NameOfSubClass. The new class has no variables, and should belong to the category *PBE-LightsOut*.

On Categories and Packages

Historically, Smalltalk only knows about *categories*, not packages. You may well ask, what is the difference? A category is simply a collection of related classes in a Smalltalk image. A *package* is a collection of related classes *and extension methods* that may be versioned using the Monticello versioning tool. By convention, package names and category names are the same. For most purposes we do not care about the difference, but we will be careful to use the correct terminology in this book since there are points where the difference is crucial. We will learn more when we start working with Monticello.

Creating a new class

We simply modify the template to create the class that we really want.

Modify the class creation template as follows:

- Replace Object by SimpleSwitchMorph.
- Replace NameOfSubClass by LOCell.
- Add mouseAction to the list of instance variables.

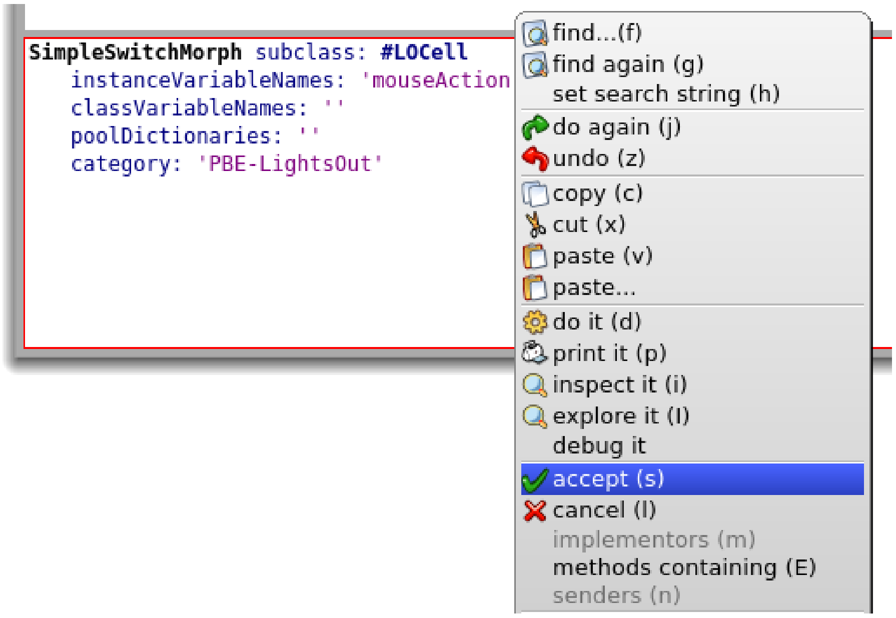
The result should look like class 2.1.

Class 2.1: *Defining the class* LOCell

```
SimpleSwitchMorph subclass: #LOCell
  instanceVariableNames: 'mouseAction'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−LightsOut'
```

This new definition consists of a Smalltalk expression that sends a message to the existing class SimpleSwitchMorph, asking it to create a subclass called LOCell. (Actually, since LOCell does not exist yet, we passed as an argument the *symbol* #LOCell which stands for the name of the class to create.) We also tell it that instances of the new class should have a mouseAction instance variable, which we will use to define what action the cell should take if the mouse should click over it.

At this point you still have not created anything. Note that the border of the class template pane has changed to red (Figure 2.4). This means that there are *unsaved changes*. To actually send this message, you must accept it.



```
SimpleSwitchMorph subclass: #LOCell
    instanceVariableNames: 'mouseAction
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE-LightsOut'
```

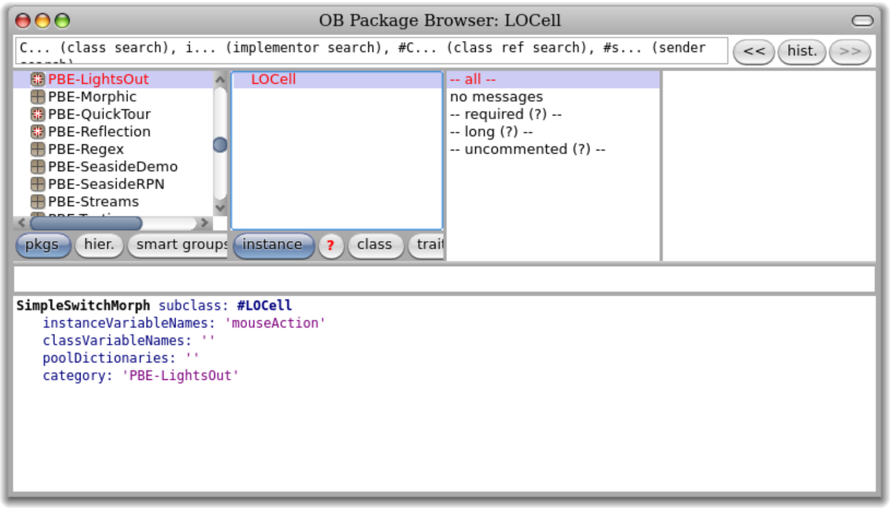
Figure 2.4: The class-creation Template.

Accept the new class definition.

Either action-click and select accept, or use the shortcut CMD–s (for "save"). The message will be sent to SimpleSwitchMorph, which will cause the new class to be compiled.

Once the class definition is accepted, the class will be created and appear in the classes pane of the browser (Figure 2.5). The editing pane now shows the class definition, and a small pane below it will remind you to write a few words describing the purpose of the class. This is called a *class comment*, and it is quite important to write one that will give other programmers a high-level overview of the purpose of this class. Smalltalkers put a very high value on the readability of their code, and detailed comments in methods are unusual: the philosophy is that the code should speak for itself. (If it doesn't, you should refactor it until it does!) A class comment need not contain a detailed description of the class, but a few words describing its overall purpose are vital if programmers who come after you are to know whether to spend time looking at this class.

Type a class comment for LOCell *and accept it; you can always improve it later.*



```
SimpleSwitchMorph subclass: #LOCell
    instanceVariableNames: 'mouseAction'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE-LightsOut'
```

Figure 2.5: The newly-created class LOCell

2.4 Adding methods to a class

Now let's add some methods to our class.

Select the protocol --all-- *in the protocol pane.*

You will see a template for method creation in the editing pane. Select it, and replace it by the text of method 2.2.

Method 2.2: *Initializing instances of* LOCell

```
1 initialize
2   super initialize.
3   self label: ''.
4   self borderWidth: 2.
5   bounds := 0@0 corner: 16@16.
6   offColor := Color paleYellow.
7   onColor := Color paleBlue darker.
8   self useSquareCorners.
9   self turnOff
```

Note that the characters '' on line 3 are two separate single quotes with nothing between them, not a double quote! '' denotes the empty string.

Accept *this method definition.*

What does the above code do? We won't go into all of the details here (that's what the rest of the book is for!), but we will give you a quick preview. Let's take it line by line.

Notice that the method is called initialize. The name is very significant! By convention, if a class defines a method named initialize, it will be called right after the object is created. So, when we evaluate LOCell new, the message initialize will be sent automatically to this newly created object. Initialize methods are used to set up the state of objects, typically to set their instance variables; this is exactly what we are doing here.

The first thing that this method does (line 2) is to execute the initialize method of its superclass, SimpleSwitchMorph. The idea here is that any inherited state will be properly initialized by the initialize method of the superclass. It is always a good idea to initialize inherited state by sending super initialize before doing anything else; we don't know exactly what SimpleSwitchMorph's initialize method will do, and we don't care, but it's a fair bet that it will set up some instance variables to hold reasonable default values, so we had better call it, or we risk starting in an unclean state.

The rest of the method sets up the state of this object. Sending self label: '', for example, sets the label of this object to the empty string.

The expression 0@0 corner: 16@16 probably needs some explanation. 0@0 represents a Point object with x and y coordinates both set to 0. In fact, 0 @0 sends the message @ to the number 0 with argument 0. The effect will be that the number 0 will ask the Point class to create a new instance with coordinates (0,0). Now we send this newly created point the message corner: 16@16, which causes it to create a Rectangle with corners 0@0 and 16@16. This newly created rectangle will be assigned to the bounds variable, inherited from the superclass.

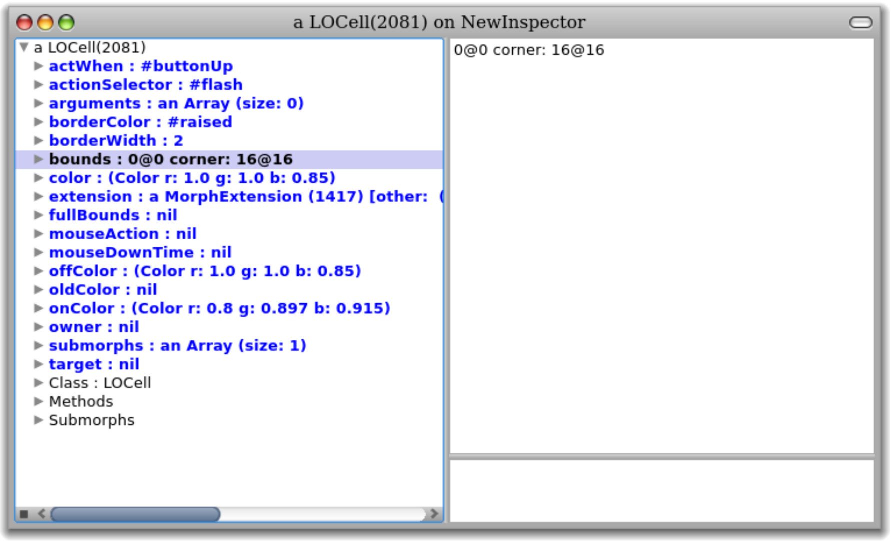
Note that the origin of the Pharo screen is the *top left*, and the y coordinate increases *downwards*.

The rest of the method should be self-explanatory. Part of the art of writing good Smalltalk code is to pick good method names so that Smalltalk code can be read like a kind of pidgin English. You should be able to imagine the object talking to itself and saying "Self use square corners!", "Self turn off!".

2.5 Inspecting an object

You can test the effect of the code you have written by creating a new LOCell object and inspecting it.

Open a workspace. Type the expression LOCell new *and* inspect it.



```
a LOCell(2081) on NewInspector

▼ a LOCell(2081)
  ▶ actWhen : #buttonUp
  ▶ actionSelector : #flash
  ▶ arguments : an Array (size: 0)
  ▶ borderColor : #raised
  ▶ borderWidth : 2
  ▶ bounds : 0@0 corner: 16@16
  ▶ color : (Color r: 1.0 g: 1.0 b: 0.85)
  ▶ extension : a MorphExtension (1417) [other:  (
  ▶ fullBounds : nil
  ▶ mouseAction : nil
  ▶ mouseDownTime : nil
  ▶ offColor : (Color r: 1.0 g: 1.0 b: 0.85)
  ▶ oldColor : nil
  ▶ onColor : (Color r: 0.8 g: 0.897 b: 0.915)
  ▶ owner : nil
  ▶ submorphs : an Array (size: 1)
  ▶ target : nil
  ▶ Class : LOCell
  ▶ Methods
  ▶ Submorphs

0@0 corner: 16@16
```

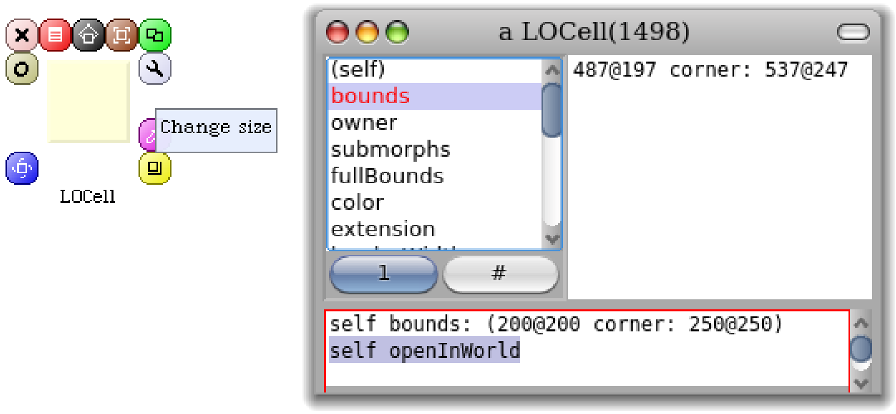
Figure 2.6: The inspector used to examine a LOCell object.

The left-hand pane of the inspector shows a list of instance variables; if you select one (try bounds), the value of the instance variable is shown in the right pane.

The bottom pane of the inspector is a mini-workspace. It's useful because in this workspace the pseudo-variable self is bound to the object being inspected.

Select the LOCell at the root of the inspector window. Type the text self bounds: (200@200 corner: 250@250) *in the bottom pane and* do it*. If you inspect the* bounds *variable you should now see that its value has changed. Now type the text* self openInWorld *in the mini-workspace and* do it*.*

The cell should appear near the top left-hand corner of the screen, indeed, exactly where its bounds say that it should appear. meta-click on the cell to bring up the morphic halo. Move the cell with the brown (next to top-right) handle and resize it with the yellow (bottom-right) handle. Notice how the bounds reported by the inspector also change.



```
(self)
bounds
owner
submorphs
fullBounds
color
extension
```

```
487@197 corner: 537@247
```

```
self bounds: (200@200 corner: 250@250)
self openInWorld
```

Figure 2.7: Resizing the cell.

Delete the cell by clicking on the x *in the pink handle.*

2.6 Defining the class LOGame

Now let's create the other class that we need for the game, which we will call LOGame.

Make the class definition template visible in the browser main window.

Do this by clicking on the package name. Edit the code so that it reads as follows, and accept it.

Class 2.3: *Defining the* LOGame *class*

```
BorderedMorph subclass: #LOGame
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−LightsOut'
```

Here we subclass BorderedMorph; Morph is the superclass of all of the graphical shapes in Pharo, and (surprise!) a BorderedMorph is a Morph with a border. We could also insert the names of the instance variables between the quotes on the second line, but for now, let's just leave that list empty.

Now let's define an initialize method for LOGame.

Type the following into the browser as a method for LOGame *and try to* accept *it:*

Method 2.4: *Initializing the game*

```
initialize
  | sampleCell width height n |
  super initialize.
  n := self cellsPerSide.
  sampleCell := LOCell new.
  width := sampleCell width.
  height := sampleCell height.
  self bounds: (5@5 extent: ((width*n) @(height*n)) + (2 * self borderWidth)).
  cells := Matrix new: n tabulate: [ :i :j | self newCellAt: i at: j ].
```

Pharo will complain that it doesn't know the meaning of some of the terms. Pharo tells you that it doesn't know of a message cellsPerSide, and suggests a number of corrections, in case it was a spelling mistake.

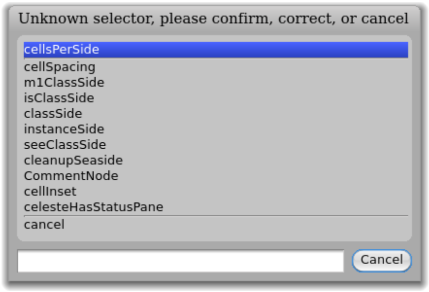


Figure 2.8: Pharo detecting an unknown selector.

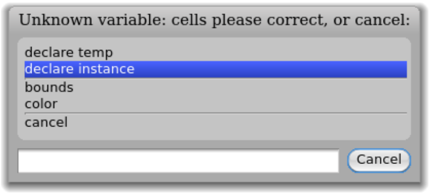


Figure 2.9: Declaring a new instance variable.

But cellsPerSide is not a mistake — it is just a method that we haven't yet defined — we will do so in a minute or two.

So just select the first item from the menu, which confirms that we really meant cellsPerSide.

Next, Pharo will complain that it doesn't know the meaning of cells. It offers you a number of ways of fixing this.

Choose declare instance *because we want* cells *to be an instance variable.*

Finally, Pharo will complain about the message newCellAt:at: sent on the last line; this is also not a mistake, so confirm that message too.

If you now look at the class definition once again (which you can do by clicking on the instance button), you will see that the browser has modified it to include the instance variable cells.

Let's look at this initialize method. The line | sampleCell width height n | declares 4 temporary variables. They are called temporary variables because their scope and lifetime are limited to this method. Temporary variables with explanatory names are helpful in making code more readable. Smalltalk has no special syntax to distinguish constants and variables, and in fact all four of these "variables" are really constants. Lines 4–7 define these constants.

How big should our game board be? Big enough to hold some integral number of cells, and big enough to draw a border around them. How many cells is the right number? 5? 10? 100? We don't know yet, and if we did, we would probably change our minds later. So we delegate the responsibility for knowing that number to another method, which we will call cellsPerSide, and which we will write in a minute or two. It's because we are sending the cellsPerSide message before we define a method with that name that Pharo asked us to "confirm, correct, or cancel" when we accepted the method body for initialize. Don't be put off by this: it is actually good practice to write in terms of other methods that we haven't yet defined. Why? Well, it wasn't until we started writing the initialize method that we realized that we needed it, and at that point, we can give it a meaningful name, and move on, without interrupting our flow.

The fourth line uses this method: the Smalltalk self cellsPerSide sends the message cellsPerSide to self, i.e., to this very object. The response, which will be the number of cells per side of the game board, is assigned to n.

The next three lines create a new LOCell object, and assign its width and height to the appropriate temporary variables.

Line 8 sets the bounds of the new object. Without worrying too much about the details just yet, just believe us that the expression in parentheses creates a square with its origin (*i.e.*, its top-left corner) at the point (5,5) and its bottom-right corner far enough away to allow space for the right number of cells.

The last line sets the LOGame object's instance variable cells to a newly created Matrix with the right number of rows and columns. We do this by sending the message new:tabulate: to the Matrix class (classes are objects too, so we can send them messages). We know that new:tabulate: takes two arguments because it has two colons (:) in its name. The arguments go right after the colons. If you are used to languages that put all of the arguments together inside parentheses, this may seem weird at first. Don't panic, it's only syntax!

It turns out to be a very good syntax because the name of the method can be used to explain the roles of the arguments. For example, it is pretty clear that Matrix rows: 5 columns: 2 has 5 rows and 2 columns, and not 2 rows and 5 columns.

Matrix new: n tabulate: [:i :j | self newCellAt: i at: j] creates a new n×n matrix and initializes its elements. The initial value of each element will depend on its coordinates. The (i,j)th element will be initialized to the result of evaluating self newCellAt: i at: j.

2.7 Organizing methods into protocols

Before we define any more methods, let's take a quick look at the third pane at the top of the browser. In the same way that the first pane of the browser lets us categorize classes into packages so we are not overwhelmed by a very long list of class names in the second pane, so the third pane lets us categorize methods so that we are not overwhelmed by a very long list of method names in the fourth pane. These categories of methods are called "protocols".

If there are only a few methods in a class, the extra level of hierarchy provided by protocols is not really necessary. This is why the browser also offers us the *--all--* virtual protocol, which, you will not be surprised to learn, contains all of the methods in the class.

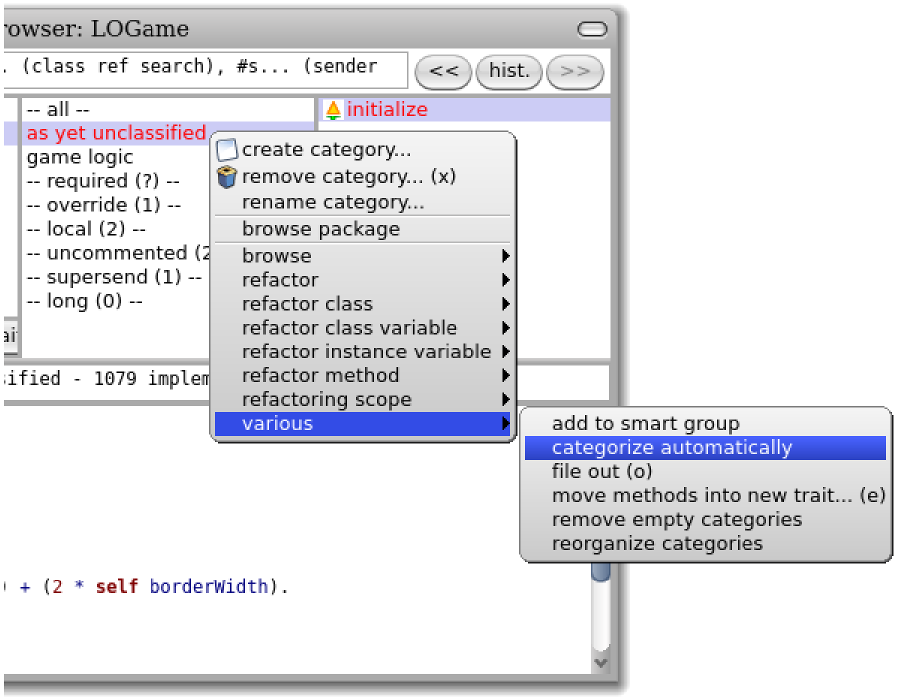
If you have followed along with this example, the third pane may well contain the protocol *as yet unclassified*.

Action-click in the protocol pane and select various ▷ categorize automatically *to fix this, and move the* initialize *methods to a new protocol called* initialization.

How does Pharo know that this is the right protocol? Well, in general Pharo can't know, but in this case there is also an initialize method in a superclass, and Pharo assumes that our initialize method should go in the same category as the one that it overrides.

A typographic convention. Smalltalkers frequently use the notation ">>" to identify the class to which a method belongs, so, for example, the cellsPerSide method in class LOGame would be referred to as LOGame>> cellsPerSide. To indicate that this is *not* Smalltalk syntax, we will use the special symbol » instead, so this method will appear in the text as LOGame» cellsPerSide

From now on, when we show a method in this book, we will write the name of the method in this form. Of course, when you actually type the code into the browser, you don't have to type the class name or the »; instead, you just make sure that the appropriate class is selected in the class pane.



```
+ (2 * self borderWidth).
```

Figure 2.10: Automatically categorize all uncategorized methods.

Now let's define the other two methods that are used by the LOGame» initialize method. Both of them can go in the *initialization* protocol.

Method 2.5: *A constant method.*

```
LOGame»cellsPerSide
  "The number of cells along each side of the game"
  ↑ 10
```

This method could hardly be simpler: it answers the constant 10. One advantage of representing constants as methods is that if the program evolves so that the constant then depends on some other features, the method can be changed to calculate this value.

Method 2.6: *An initialization helper method*

```
LOGame»newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on–screen
  representation at the appropriate screen position.  Answer the new cell"
  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i – 1) * c width) @ ((j – 1) * c height) + origin.
  c mouseAction: [self toggleNeighboursOfCellAt: i at: j]
```

Add the methods LOGame»cellsPerSide *and* LOGame»newCellAt:at:.

Confirm the spelling of the new selectors toggleNeighboursOfCellAt:at: and mouseAction:.

Method 2.6 answers a new LOCell, specialized to position (i, j) in the Matrix of cells. The last line defines the new cell's mouseAction to be the *block* [self toggleNeighboursOfCellAt: i at: j]. In effect, this defines the callback behaviour to perform when the mouse is clicked. The corresponding method also needs to be defined.

Method 2.7: *The callback method*

```
LOGame»toggleNeighboursOfCellAt: i at: j
  (i > 1) ifTrue: [ (cells at: i – 1 at: j ) toggleState].
  (i < self cellsPerSide) ifTrue: [ (cells at: i + 1 at: j) toggleState].
  (j > 1) ifTrue: [ (cells at: i  at: j – 1) toggleState].
  (j < self cellsPerSide) ifTrue: [ (cells at: i at: j + 1) toggleState].
```

Method 2.7 toggles the state of the four cells to the north, south, west and east of cell (i, j). The only complication is that the board is finite, so we have to make sure that a neighboring cell exists before we toggle its state.

Place this method in a new protocol called game logic. *(Action-click in the protocol pane to add a new protocol.)*

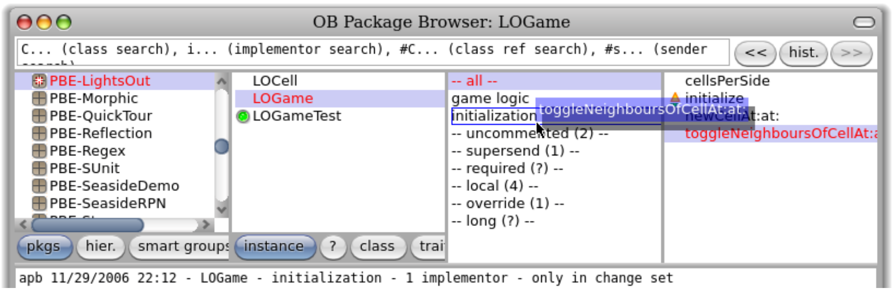
To move the method, you can simply click on its name and drag it to the newly-created protocol (Figure 2.11).

To complete the Lights Out game, we need to define two more methods in class LOCell to handle mouse events.

Method 2.8: *A typical setter method*

```
LOCell»mouseAction: aBlock
  ↑ mouseAction := aBlock
```

Method 2.8 does nothing more than set the cell's mouseAction variable to the argument, and then answers the new value. Any method that *changes* the



```
apb 11/29/2006 22:12 - LOGame - initialization - 1 implementor - only in change set
```

Figure 2.11: Drag a method to a protocol.

value of an instance variable in this way is called a *setter method*; a method that *answers* the current value of an instance variable is called a *getter method*.

If you are used to getters and setters in other programming languages, you might expect these methods to be called setmouseAction and getmouseAction. The Smalltalk convention is different. A getter always has the same name as the variable it gets, and a setter is named similarly, but with a trailing ":", hence mouseAction and mouseAction:.

Collectively, setters and getters are called *accessor* methods, and by convention they should be placed in the *accessing* protocol. In Smalltalk, *all* instance variables are private to the object that owns them, so the only way for another object to read or write those variables in the Smalltalk language is through accessor methods like this one[3].

Go to the class LOCell, *define* LOCell»mouseAction: *and put it in the* accessing *protocol.*

Finally, we need to define a method mouseUp:; this will be called automatically by the GUI framework if the mouse button is released while the mouse is over this cell on the screen.

Method 2.9: *An event handler*

```
LOCell»mouseUp: anEvent
  mouseAction value
```

Add the method LOCell»mouseUp: *and then* automatically categorize *methods.*

What this method does is to send the message value to the object stored in the instance variable mouseAction. Recall that in LOGame»newCellAt: i at: j we assigned the following code fragment to mouseAction:

[3]In fact, the instance variables can be accessed in subclasses too.

```
[self toggleNeighboursOfCellAt: i at: j ]
```

Sending the value message causes this code fragment to be evaluated, and consequently the state of the cells will toggle.

2.8 Let's try our code

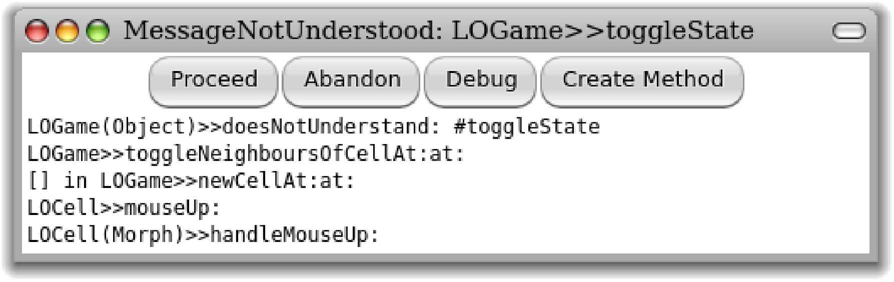
That's it: the Lights Out game is complete!

If you have followed all of the steps, you should be able to play the game, consisting of just 2 classes and 7 methods.

In a workspace, type LOGame new openInWorld *and* do it.

The game will open, and you should be able to click on the cells and see how it works.

Well, so much for theory... When you click on a cell, a *notifier* window called the PreDebugWindow window appears with an error message! As depicted in Figure 2.12, it says MessageNotUnderstood: LOGame»toggleState.



```
LOGame(Object)>>doesNotUnderstand: #toggleState
LOGame>>toggleNeighboursOfCellAt:at:
[] in LOGame>>newCellAt:at:
LOCell>>mouseUp:
LOCell(Morph)>>handleMouseUp:
```

Figure 2.12: There is a bug in our game when a cell is clicked!

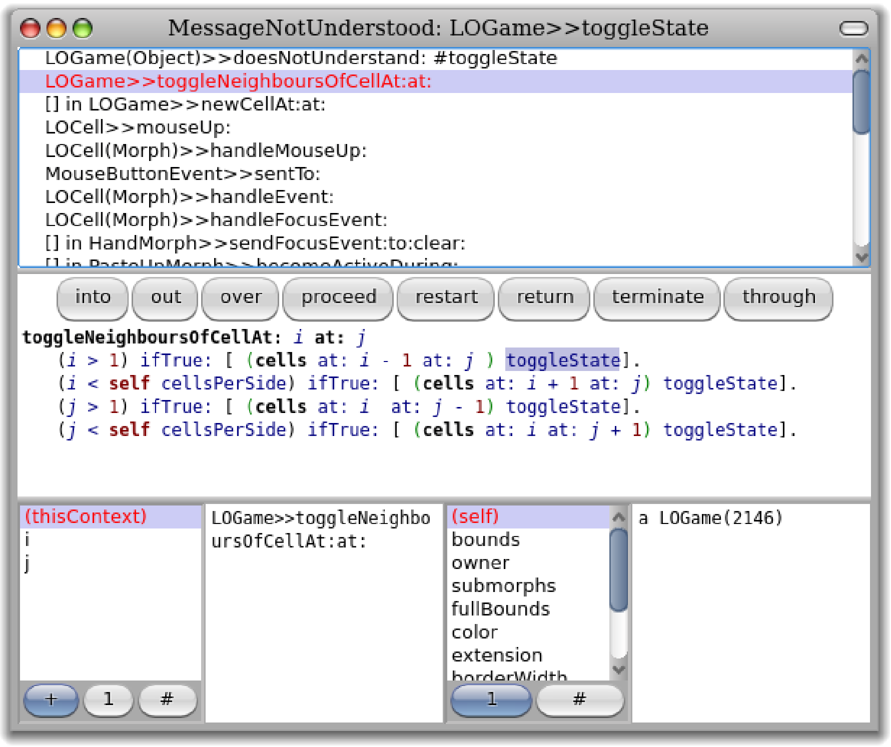
What happened? To find out, let's use one of Smalltalk's more powerful tools: the debugger.

Click on the debug *button in the notifer window.*

The debugger will appear. In the upper part of the debugger window you can see the execution stack, showing all the active methods; selecting any one of them will show, in the middle pane, the Smalltalk code being executed in that method, with the part that triggered the error highlighted.

Click on the line labelled LOGame»toggleNeighboursOfCellAt:at: *(near the top).*

The debugger will show you the execution context within this method where the error occurred (Figure 2.13).



```
LOGame(Object)>>doesNotUnderstand: #toggleState
LOGame>>toggleNeighboursOfCellAt:at:
[] in LOGame>>newCellAt:at:
LOCell>>mouseUp:
LOCell(Morph)>>handleMouseUp:
MouseButtonEvent>>sentTo:
LOCell(Morph)>>handleEvent:
LOCell(Morph)>>handleFocusEvent:
[] in HandMorph>>sendFocusEvent:to:clear:
```

```
toggleNeighboursOfCellAt: i at: j
   (i > 1) ifTrue: [ (cells at: i - 1 at: j ) toggleState].
   (i < self cellsPerSide) ifTrue: [ (cells at: i + 1 at: j) toggleState].
   (j > 1) ifTrue: [ (cells at: i  at: j - 1) toggleState].
   (j < self cellsPerSide) ifTrue: [ (cells at: i at: j + 1) toggleState].
```

```
(thisContext)
i
j
```

```
LOGame>>toggleNeighboursOfCellAt:at:
```

```
(self)
bounds
owner
submorphs
fullBounds
color
extension
```

```
a LOGame(2146)
```

Figure 2.13: The debugger, with the method toggleNeighboursOfCell:at: selected.

At the bottom of the debugger are two small inspector windows. On the left, you can inspect the object that is the receiver of the message that caused the selected method to execute, so you can look here to see the values of the instance variables. On the right you can inspect an object that represents the currently executing method itself, so you can look here to see the values of the method's parameters and temporary variables.

Using the debugger, you can execute code step by step, inspect objects in parameters and local variables, evaluate code just as you can in a workspace, and, most surprisingly to those used to other debuggers, change the code while it is being debugged! Some Smalltalkers program in the debugger almost all the time, rather than in the browser. The advantage of this is that you see the method that you are writing as it will be executed, with real parameters in the actual execution context.

In this case we can see in the first line of the top panel that the toggleState message has been sent to an instance of LOGame, while it should clearly have been an instance of LOCell. The problem is most likely with the initialization of the cells matrix. Browsing the code of LOGame»initialize shows that cells is filled with the return values of newCellAt:at:, but when we look at that method, we see that there is no return statement there! By default, a method returns self, which in the case of newCellAt:at: is indeed an instance of LOGame.

Close the debugger window. Add the expression "↑ c" to the end of the method LOGame»newCellAt:at: *so that it returns* c. *(See method 2.10.)*

Method 2.10: *Fixing the bug.*

```
LOGame»newCellAt: i at: j
  "Create a cell for position (i,j) and add it to my on–screen
  representation at the appropriate screen position.  Answer the new cell"
  | c origin |
  c := LOCell new.
  origin := self innerBounds origin.
  self addMorph: c.
  c position: ((i – 1) * c width) @ ((j – 1) * c height) + origin.
  c mouseAction: [self toggleNeighboursOfCellAt: i at: j].
  ↑ c
```

Recall from Chapter 1 that the construct to return a value from a method in Smalltalk is ↑, which you obtain by typing ^.

Often, you can fix the code directly in the debugger window and click Proceed to continue running the application. In our case, because the bug was in the initialization of an object, rather than in the method that failed, the easiest thing to do is to close the debugger window, destroy the running instance of the game (with the halo), and create a new one.

Do: LOGame new openInWorld *again.*

Now the game should work properly.

2.9 Saving and sharing Smalltalk code

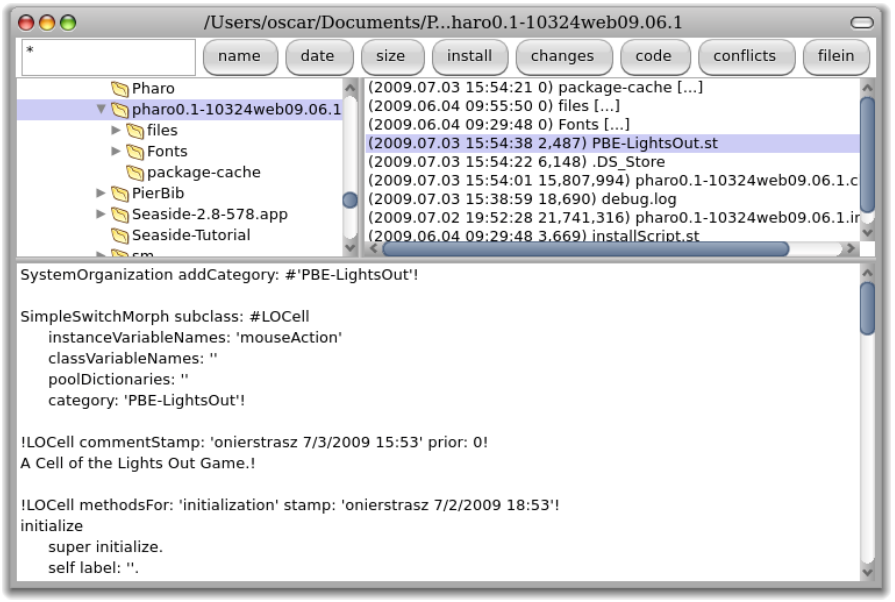
Now that you have the Lights Out game working, you probably want to save it somewhere so that you can share it with your friends. Of course, you can save your whole Pharo image, and show off your first program by running it, but your friends probably have their own code in their images, and don't want to give that up to use your image. What you need is a way of getting source code out of your Pharo image so that other programmers can bring it into theirs.

The simplest way of doing this is by *filing out* the code. The action-click menu in the System Categories pane will give you the option to file out the whole of category *PBE-LightsOut*. The resulting file is more or less human readable, but is really intended for computers, not humans. You can email this file to your friends, and they can file it into their own Pharo images using the file list browser.

Action-click on the PBE-LightsOut *package and* various ▷ file out *the contents.*

You should now find a file called "PBE-LightsOut.st" in the same folder on disk where your image is saved. Have a look at this file with a text editor.

*Open a fresh Pharo image and use the File Browser tool (*Tools . . . ▷ File Browser*) to* file in *the PBE-LightsOut.st fileout. Verify that the game now works in the new image.*



```
(2009.07.03 15:54:21 0) package-cache [...]
(2009.06.04 09:55:50 0) files [...]
(2009.06.04 09:29:48 0) Fonts [...]
(2009.07.03 15:54:38 2,487) PBE-LightsOut.st
(2009.07.03 15:54:22 6,148) .DS_Store
(2009.07.03 15:54:01 15,807,994) pharo0.1-10324web09.06.1.c
(2009.07.03 15:38:59 18,690) debug.log
(2009.07.02 19:52:28 21,741,316) pharo0.1-10324web09.06.1.ir
(2009.06.04 09:29:48 3,669) installScript.st
```

```
SystemOrganization addCategory: #'PBE-LightsOut'!

SimpleSwitchMorph subclass: #LOCell
    instanceVariableNames: 'mouseAction'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE-LightsOut'!

!LOCell commentStamp: 'onierstrasz 7/3/2009 15:53' prior: 0!
A Cell of the Lights Out Game.!

!LOCell methodsFor: 'initialization' stamp: 'onierstrasz 7/2/2009 18:53'!
initialize
    super initialize.
    self label: ''.
```

Figure 2.14: Filing in Pharo source code.

Monticello packages

Although fileouts are a convenient way of making a snapshot of the code you have written, they are decidedly "old school". Just as most open-source projects find it much more convenient to maintain their code in a repository using CVS[4] or Subversion[5], so Pharo programmers find it more convenient

[4] http://www.nongnu.org/cvs
[5] http://subversion.tigris.org

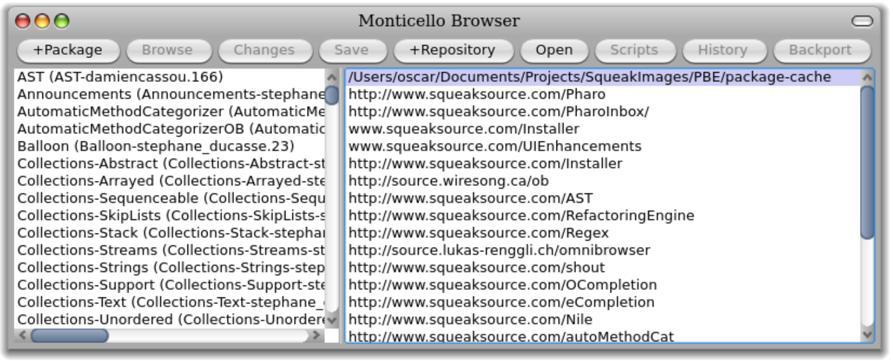
to manage their code using Monticello packages. These packages are represented as files with names ending in .mcz; they are actually zip-compressed bundles that contain the complete code of your package.

Using the Monticello package browser, you can save packages to repositories on various types of server, including FTP and HTTP servers; you can also just write the packages to a repository in a local file system directory. A copy of your package is also always cached on your local hard-disk in the *package-cache* folder. Monticello lets you save multiple versions of your program, merge versions, go back to an old version, and browse the differences between versions. In fact, Monticello is a distributed revision control system; this means it allows developers to save their work on different places, not on a single repository as it is the case with CVS or Subversion.

You can also send a .mcz file by email. The recipient will have to place it in her *package-cache* folder; she will then be able to use Monticello to browse and load it.

Open the Monticello browser from the World *menu.*

In the right-hand pane of the browser (see Figure 2.15) is a list of Monticello repositories, which will include all of the repositories from which code has been loaded into the image that you are using.



```
AST (AST-damiencassou.166)
Announcements (Announcements-stephane
AutomaticMethodCategorizer (AutomaticMe
AutomaticMethodCategorizerOB (Automatic
Balloon (Balloon-stephane_ducasse.23)
Collections-Abstract (Collections-Abstract-st
Collections-Arrayed (Collections-Arrayed-ste
Collections-Sequenceable (Collections-Sequ
Collections-SkipLists (Collections-SkipLists-s
Collections-Stack (Collections-Stack-stephai
Collections-Streams (Collections-Streams-st
Collections-Strings (Collections-Strings-step
Collections-Support (Collections-Support-ste
Collections-Text (Collections-Text-stephane_
Collections-Unordered (Collections-Unorder
```

```
/Users/oscar/Documents/Projects/SqueakImages/PBE/package-cache
http://www.squeaksource.com/Pharo
http://www.squeaksource.com/PharoInbox/
www.squeaksource.com/Installer
www.squeaksource.com/UIEnhancements
http://www.squeaksource.com/Installer
http://source.wiresong.ca/ob
http://www.squeaksource.com/AST
http://www.squeaksource.com/RefactoringEngine
http://www.squeaksource.com/Regex
http://source.lukas-renggli.ch/omnibrowser
http://www.squeaksource.com/shout
http://www.squeaksource.com/OCompletion
http://www.squeaksource.com/eCompletion
http://www.squeaksource.com/Nile
http://www.squeaksource.com/autoMethodCat
```

Figure 2.15: The Monticello browser.

At the top of the list in the Monticello browser is a repository in a local directory called the *package cache*, which caches copies of the packages that you have loaded or published over the network. This local cache is really handy because it lets you keep your own local history; it also allows you to work in places where you do not have internet access, or where access is slow enough that you do not want to save to a remote repository very frequently.

Saving and loading code with Monticello.

On the left-hand side of the Monticello browser is a list of packages that have a version loaded into the image; packages that have been modified since they were loaded are marked with an asterisk. (These are sometimes referred to as dirty packages.) If you select a package, the list of repositories is restricted to just those repositories that contain a copy of the selected package.

What is a package? For now, you can think of a package as a group of class and method categories that share the same prefix. Since we put all of the code for the Lights Out game into the category called *PBE-LightsOut*, we can refer to it as the PBE–LightsOut package.

Add the PBE–LightsOut *package to your Monticello browser using the* +Package *button and type* PBE–LightsOut.

SqueakSource : a SourceForge for Pharo.

We think that the best way to save your code and share it is to create an account for your project on a SqueakSource server. SqueakSource is like SourceForge[6]: it is a web front-end to a HTTP Monticello server that lets you manage your projects. There is a public SqueakSource server at http://www.squeaksource.com, and a copy of the code related to this book is stored there at http://www.squeaksource.com/PharoByExample.html. You can look at this project with a web browser, but it's a lot more productive to do so from inside Pharo, using the Monticello browser, which lets you manage your packages.

Open a web browser to http://www.squeaksource.com. *Create an account for yourself and then create* (i.e., *"register") a project for the Lights Out game.*

SqueakSource will show you the information that you should use when adding a repository using the Monticello browser.

Once your project has been created on SqueakSource, you have to tell your Pharo system to use it.

With the PBE–LightsOut *package selected, click the* +Repository *button in the Monticello browser.*

You will see a list of the different types of Repository that are available; to add a SqueakSource repository select HTTP. You will be presented with a dialog in which you can provide the necessary information about the server. You should copy the presented template to identify your SqueakSource project, paste it into Monticello and supply your initials and password:

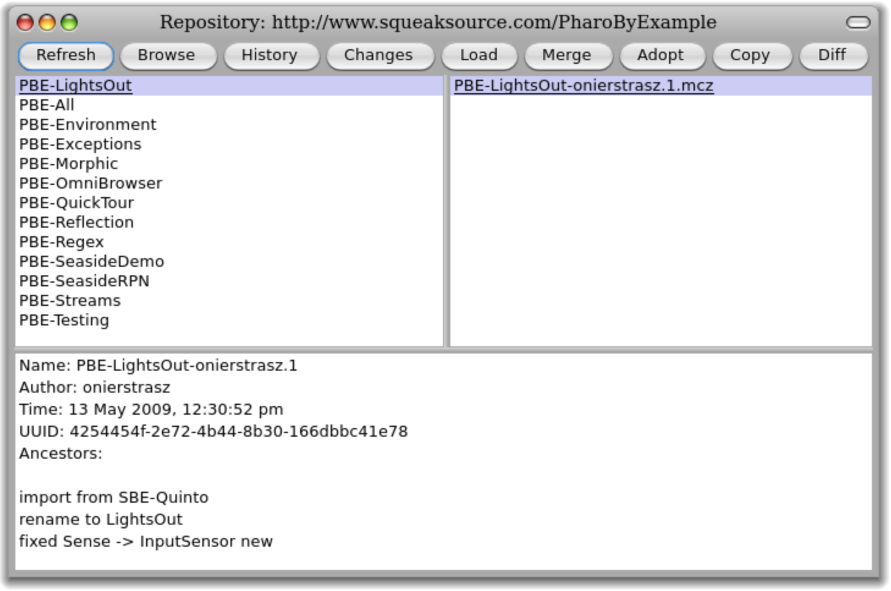
[6] http://sourceforge.net

```
MCHttpRepository
   location: 'http://www.squeaksource.com/YourProject'
   user: 'yourInitials'
   password: 'yourPassword'
```

If you provide empty initials and password strings, you can still load the project, but you will not be able to update it:

```
MCHttpRepository
   location: 'http://www.squeaksource.com/YourProject'
   user: ''
   password: ''
```

Once you have accepted this template, your new repository should be listed on the right-hand side of the Monticello browser.



```
PBE-LightsOut
PBE-All
PBE-Environment
PBE-Exceptions
PBE-Morphic
PBE-OmniBrowser
PBE-QuickTour
PBE-Reflection
PBE-Regex
PBE-SeasideDemo
PBE-SeasideRPN
PBE-Streams
PBE-Testing

PBE-LightsOut-onierstrasz.1.mcz

Name: PBE-LightsOut-onierstrasz.1
Author: onierstrasz
Time: 13 May 2009, 12:30:52 pm
UUID: 4254454f-2e72-4b44-8b30-166dbbc41e78
Ancestors:

import from SBE-Quinto
rename to LightsOut
fixed Sense -> InputSensor new
```

Figure 2.16: Browsing a Monticello Repository

Click on the Save *button to save a first version of your Lights Out game on SqueakSource.*

To load a package into your image, you must first select a particular version. You can do this in the repository browser, which you can open using the Open button or the action-click menu. Once you have selected a version, you can load it onto your image.

Open the PBE–LightsOut *repository you have just saved.*

Monticello has many more capabilities, which will be discussed in depth in Chapter 6. You can also look at the on-line documentation for Monticello at http://www.wiresong.ca/Monticello/.

2.10 Chapter summary

In this chapter you have seen how to create categories, classes and methods. You have see how to use the browser, the inspector, the debugger and the Monticello browser.

- Categories are groups of related classes.
- A new class is created by sending a message to its superclass.
- Protocols are groups of related methods.
- A new method is created or modified by editing its definition in the browser and then *accepting* the changes.
- The inspector offers a simple, general-purpose GUI for inspecting and interacting with arbitrary objects.
- The browser detects usage of undeclared methods and variables, and offers possible corrections.
- The initialize method is automatically executed after an object is created in Pharo. You can put any initialization code there.
- The debugger provides a high-level GUI to inspect and modify the state of a running program.
- You can share source code *filing out* a category.
- A better way to share code is to use Monticello to manage an external repository, for example defined as a SqueakSource project.