Pharo by Example 9.0

Stéphane Ducasse, Sebastijan Kaplar, Gordana Rakic and Quentin Ducasse

July 3, 2021
## Contents

### Illustrations

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A first application</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>The Lights Out game</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Creating a new package</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Defining the class LOCel</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Creating a new class</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>About comments</td>
<td>4</td>
</tr>
<tr>
<td>1.6</td>
<td>Adding methods to a class</td>
<td>5</td>
</tr>
<tr>
<td>1.7</td>
<td>Inspecting an object</td>
<td>7</td>
</tr>
<tr>
<td>1.8</td>
<td>Defining the class LOGame</td>
<td>9</td>
</tr>
<tr>
<td>1.9</td>
<td>Initializing our game</td>
<td>10</td>
</tr>
<tr>
<td>1.10</td>
<td>Taking advantage of the debugger</td>
<td>10</td>
</tr>
<tr>
<td>1.11</td>
<td>Studying the initialize method</td>
<td>13</td>
</tr>
<tr>
<td>1.12</td>
<td>Organizing methods into protocols</td>
<td>14</td>
</tr>
<tr>
<td>1.13</td>
<td>A typographic convention</td>
<td>15</td>
</tr>
<tr>
<td>1.14</td>
<td>Finishing the game</td>
<td>15</td>
</tr>
<tr>
<td>1.15</td>
<td>Final LOCel methods</td>
<td>16</td>
</tr>
<tr>
<td>1.16</td>
<td>Using the debugger</td>
<td>17</td>
</tr>
<tr>
<td>1.17</td>
<td>In case everything fails</td>
<td>20</td>
</tr>
<tr>
<td>1.18</td>
<td>Saving and sharing Pharo code</td>
<td>20</td>
</tr>
<tr>
<td>1.19</td>
<td>Iceberg: Pharo and Git</td>
<td>20</td>
</tr>
<tr>
<td>1.20</td>
<td>Saving code in a file</td>
<td>22</td>
</tr>
<tr>
<td>1.21</td>
<td>About Setter/Getter convention</td>
<td>24</td>
</tr>
<tr>
<td>1.22</td>
<td>On categories vs. packages</td>
<td>24</td>
</tr>
<tr>
<td>1.23</td>
<td>Chapter summary</td>
<td>25</td>
</tr>
</tbody>
</table>

### Bibliography

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
</tr>
</tbody>
</table>
Illustrations

1-1 The Lights Out game board. .............................................. 1
1-2 Create a Package and class template. .............................. 2
1-3 Filtering our package to work more efficiently. .................. 3
1-4 LOCell class definition .............................................. 3
1-5 The newly-created class LOCell. .................................... 4
1-6 Initializing instance of LOCell ....................................... 5
1-7 The newly-created method initialize. ............................. 6
1-8 The inspector used to examine a LOCell object. ............... 8
1-9 When we click on an instance variable, we inspect its value (another object). 8
1-10 A LOCell opened in the World. .................................... 9
1-11 Defining the LOGame class ........................................ 9
1-12 Initialize the game .................................................. 10
1-13 Declaring cells as a new instance variable. .................... 10
1-14 Pharo detecting an unknown selector. ............................ 11
1-15 The system created a new method with a body to be defined. 12
1-16 Defining cellsPerSide in the debugger. ........................ 12
1-17 Initialize the game .................................................. 13
1-18 The callback method ............................................... 16
1-19 Drag a method to a protocol. ..................................... 16
1-20 A typical setter method ........................................... 17
1-21 An event handler ................................................... 17
1-22 The debugger, with the method toggleNeighboursOfCell:at: selected. 18
1-23 Fixing the bug. ..................................................... 19
1-24 Overriding mouse move actions .................................. 19
1-25 Repositories screen. ................................................ 21
1-26 Creating new repository. ........................................... 21
1-27 Updated repositories screen. ...................................... 22
1-28 Iceberg working copy dialog. ..................................... 22
1-29 Iceberg working copy dialog. ..................................... 23
1-30 File Out our PBE-LightsOut. ..................................... 23
1-31 Import your code with the file browser. ......................... 24
In this chapter, we will develop a simple game: LightsOut (http://en.wikipedia.org/wiki/Lights_Out_(game)). Along the way we will show most of the tools that Pharo programmers use to construct and debug their programs, and show how programs are shared with other developers. We will define a buggy method and show you how to fix it. We will present the browser, the object inspector, the debugger and the way to version code.

In Pharo you can develop in a traditional way, by defining a class, then its instance variables, then its methods. However, in Pharo your development flow can be much more productive than that! You can define instance variables and methods on the fly. You can also code in the debugger using the exact context of currently executed objects. This chapter will sketch such alternate way and show you how you can be really productive.

We will code the game but doing so we will make mistakes and we will show you how we recover from these mistakes. So this may be a bit frustrating for you and more boring for us to describe but this is a key aspect of program-

![Lights Out game board](Figure 1-1) The Lights Out game board.
A first application

Figure 1-2  Create a Package and class template.

...ming. We have to show how to handle errors and find bugs.

1.1  The Lights Out game

To show you how to use Pharo’s programming tools, we will build a simple game called Lights Out. The game board consists of a rectangular array of light yellow cells. When you click on one of the cells, 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.

Lights Out 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 development tools.

1.2  Creating a new package

We have already seen the browser in Chapter: A Quick Tour of Pharo where we learned how to navigate to packages, classes and methods, and saw how to define new methods. Now we will see how to create packages and classes.

From the World menu, open a System Browser. Right-click on an existing package in the Package pane and select New package from the menu. Type the name of the new package (we use PBE-LightsOut) in the dialog box and click OK (or just press the return key). The new package is created, and positioned alphabetically in the list of packages (see Figure 1-2).

Hints: You can type PBE in the filter to get your package filtered out the other ones (See Figure 1-3).
1.3 Defining the class LOCell

At this point 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 1-3).

This template shows us a Pharo 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 (package) PBE-LightsOut.

1.4 Creating a new class

We simply edit the template to create the class that we really want. Modify the class creation template as follows:

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

You should get the following class definition:

This new definition consists of a Pharo expression that sends a message to the existing class SimpleSwitchMorph, asking it to create a subclass called
A first application

LOCell. (Actually, since LOCell does not exist yet, we passed the symbol #LOCell as an argument, representing 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 on it.

At this point you still have not created anything. Note that the top right of the panel changed to orange. This means that there are unsaved changes. To actually send this subclass message, you must save (accept) the source code. Either right-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. You should get the situation depicted in Figure 1-5.

Once the class definition is accepted, the class is created and appears in the class pane of the browser (see Figure 1-5). The editing pane now shows the class definition. Below you get the Quality Assistant’s feedback: It runs automatically quality rules on your code and reports them.

1.5 About comments

Pharoers put a very high value on the readability of their code, but also good quality comments.
1.6 Adding methods to a class

**Listing 1-6** Initializing instance of LOCell

```plaintext
initialize
    super initialize.
    self label: ' '
    self borderWidth: 2.
    bounds := 0 @ 0 corner: 16 @ 16.
    offColor := Color paleYellow.
    onColor := Color paleBlue darker.
    self useSquareCorners.
    self turnOff
```

Method comments.

People have the tendency to believe that it is not necessary to comment well written methods: it is plain wrong and encourages sloppiness. Of course, bad code should renamed and refactored. Obviously commenting trivial methods makes no sense. A comment should not be the code written in english but an explanation of what the method is doing, its context, or the rationale behind its implementation. When reading a comment, the reader should be comforted that his hypotheses are correct.

Class comments.

For the class comment, the Pharo class comment template gives a good idea of a strong class comment. Read it! It is based on CRC for Class Responsibility Collaborators. So in a nutshell the comments state the responsibility of the class in a couple of sentences and how it collaborates with other classes to achieve this responsibilities. In addition we can state the API (main messages an object understands), give an example (usually in Pharo we define examples as class methods), and some details about internal representation or implementation rationale.

1.6 Adding methods to a class

Now let’s add some methods to our class. Select the Inst. side method tab next to the class definition tab. You will see a template for method creation in the editing pane. Select the template text, and replace it by the following (do not forget to compile it):

Note that the characters ' ' on line 3 are two separate single quotes with nothing between them, not a double quote! ' ' denotes the empty string. Another way to create an empty string is String new. Do not forget to compile this method Pharo using the accept menu item (CMD-s/Option-s).
Notice that the method is called initialize. The name is very significant! By convention, if a class defines a method named initialize, it is called right after the object is created. So, when we execute LOCell new, the message initialize is 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.

Invoking superclass initialization.

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.
About point and rectangle creation.

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.

About the rest.

The rest of the method should be self-explanatory. Part of the art of writing good Pharo code is to pick good method names so that the 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".

Notice that there is a little green arrow next to your method (see Figure 1-7). This means the method exists in the superclass and is overridden in your class.

### 1.7 Inspecting an object

You can immediately test the effect of the code you have written by creating a new LCell object and inspecting it: Open a Playground, type the expression LCell new, and Inspect it (using the menu item with the same name).

The left-hand column of the inspector shows a list of instance variables and the value of the instance variable is shown in the right column (see Figure 1-8).

If you click on an instance variable the inspector will open a new pane with the detail of the instance variable (see Figure 1-9).

Executing expressions.

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

Go to that Playground at the bottom of the pane and type the text self bounds: \((200@200\) corner: \(250@250\)) Do it. To refresh the values, click on the update button (the blue little circle) at the top right of the pane. The bounds variable should change in the inspector. Now type the text self openIn-World in the mini-playground and Do it.
Figure 1-8  The inspector used to examine a LOCell object.

Figure 1-9  When we click on an instance variable, we inspect its value (another object).
1.8 Defining the class LOGame

Figure 1-10  A LOCell opened in the World.

Listing 1-11  Defining the LOGame class

BorderedMorph subclass: #LOGame

  instanceVariableNames: ''
  classVariableNames: ''
  package: 'PBE-LightsOut'

The cell should appear near the top left-hand corner of the screen (as shown in Figure 1-10) and 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. (You may have to click refresh to see the new bounds value.) Delete the cell by clicking on the x in the pink handle.

1.8 Defining the class LOGame

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

Make the class definition template visible in the browser main window. Do this by clicking on the package name (or right-clicking on the Class pane and selecting Add Class). Edit the code so that it reads as follows, and Accept it.

Here we subclass BorderedMorph. Morph is the superclass of all of the graphical shapes in Pharo, and (unsurprisingly) a BorderedMorph is a Morph with a
A first application

Listing 1-12  Initialize the game

```smalltalk
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 := Array2D new: n tabulate: [ :i :j | self newCellAt: i at: 
        j ]
```

Figure 1-13  Declaring cells as a new instance variable.

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.

1.9  Initializing our game

Now let's define an initialize method for LOGame. Type the following into the browser as a method for LOGame and Accept it.

Pharo will complain that it doesn't know the meaning of cells (see Figure 1-13). It will offer you a number of ways to fix this.

Choose Declare new instance variable, because we want cells to be an instance variable.

1.10  Taking advantage of the debugger

At this stage if you open a Playground, type LOGame  new, and Do it, Pharo will complain that it doesn't know the meaning of some of the terms (see Figure 1-14). It will tell you that it doesn't know of a message cellsPerSide, and will open a debugger. But cellsPerSide is not a mistake; it is just a method that we haven't yet defined. We will do so, shortly.

Now let us do it: type LOGame  new and Do it. Do not close the debugger.
Click on the button Create of the debugger, when prompted, select LOGame,
1.10 Taking advantage of the debugger

![Image](image.png)

Figure 1-14 Pharo detecting an unknown selector.

the class which will contain the method, click on ok, then when prompted for a method protocol enter accessing. The debugger will create the method cellsPerSide on the fly and invoke it immediately. As there is no magic, the created method will simply raise an exception and the debugger will stop again (as shown in Figure 1-15) giving you the opportunity to define the behavior of the method.

Here you can write your method. 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.

```small
  cellsPerSide
  "The number of cells along each side of the game"
  ^ 10
```

Define the method cellsPerSide in the debugger. Do not forget to compile the method definition by using Accept. You should obtain a situation as shown by Figure 1-16. If you press the button Proceed the program will continue its execution - here it will stop since we did not define the method newCellAt:.

We could use the same process but for now we stop to explain a bit what we did so far. Close the debugger, and look at the class definition once again (which you can do by clicking on LOGame on the second pane of the Sys-
Figure 1-15  The system created a new method with a body to be defined.

Figure 1-16  Defining `cellsPerSide` in the debugger.
1.11 Studying the initialize method

Listing 1-17 Initialize the game

```smalltalk
initialize |
  sampleCell width height n |
super initialize.
n := self cellsPerSide.
sampleCell := LOCell new.
width := sampleCell width.
height := sampleCell height.
sel bounds: (50 @ 50 extent: (width * n) @ (height * n) + (2 *
sel borderWidth)).
cells := Array2D
  new: n
  tabulate: [:i :j | self newCellAt: i at: j ]
```

1.11 Studying the initialize method

Let us now study the method `initialize`.

Line 2

At line 2, the expression `| 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. Lines 4-7 set the value of these variables.

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 name `cellsPerSide`, and which we will write in a minute or two. Don't be put off by this: it is actually good practice to code by referring to 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.

Line 4

The fourth line uses this method, `n := 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
Line 8 sets the bounds of the new object. Without worrying too much about the details just yet, believe us that the expression in parentheses creates a square with its origin (i.e., its top-left corner) at the point (50,50) and its bottom-right corner far enough away to allow space for the right number of cells.

Last line
The last line sets the LOGame object's instance variable cells to a newly created Array2D with the right number of rows and columns. We do this by sending the message new: tabulate: to the Array2D 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 Array2D rows: 5 columns: 2 has 5 rows and 2 columns, and not 2 rows and 5 columns.

Array2D new: n tabulate: [ :i :j | self newCellAt: i at: j ]
creates a new n X n two dimensional array (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.

1.12 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, the protocol pane lets us categorize methods so that we are not overwhelmed by a very long list of method names in the method pane. These groups of methods are called "protocols".

By default, you will have instance side virtual protocol, which contains all of the methods in the class.

If you have followed along with this example, the protocol pane may well contain the initialization and overrides protocols. These protocols are added automatically when you override initialize. Pharo 8 System Browser organizes the methods automatically, and add them to the appropriate protocol, when possible.
How does the **System Browser** know that this is the right protocol? Well, in general Pharo can't know exactly, but for example if there is also an `initialize` method in the superclass, and it assumes that our `initialize` method should go in the same protocol as the one that it overrides.

The protocol pane may contain the protocol **as yet unclassified**. Methods that aren't organized into protocols can be found here. You can right-click in the protocol pane and select **categorize all uncategorized** to fix this, or you can organize manually.

### 1.13 A typographic convention

Pharoers frequently use the notation `Class >> method` to identify the class to which a method belongs. For example, the `cellsPerSide` method in class `LOGame` would be referred to as `LOGame >> cellsPerSide`. Just keep in mind that this is not Pharo syntax exactly, but merely a convenient notation to indicate "the instance method `cellsPerSide` which belongs to the class `LOGame". The corresponding notation for a class-side method would be `LOGame class >> #someClassSideMethod`.

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.

### 1.14 Finishing the game

Now let's define the other method that are used by `LOGame >> initialize`. Let's define `LOGame >> newCellAt: at:` in the initialization protocol.

```smalltalk
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 ].
```

Pay attention the previous code is not fully correct. Therefore, it will produce an error and this is on purpose.
A first application

Listing 1-18  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 ]

Listing 1-20  A typical setter method

LOCell >> mouseAction: aBlock

    mouseAction := aBlock

Formatting.

As you can see there are some tabulation and empty lines. To keep the same
convention you can right-click on the method edit area and click on Format
(or use CMD-Shift-f shortcut). This will format your method.

Toggle neighbours.

The method defined above created a new LOCell, initialized to position (i, j)
in the Array2D 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.

The method toggleNeighboursOfCellAt:at: 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. (Right-click in the
protocol pane to add a new protocol.) To move (re-classify) the method, you
can simply click on its name and drag it to the newly-created protocol (see
Figure 1-19).

1.15  Final LOCell methods

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

The method above does nothing more than set the cell’s mouseAction vari-
able to the argument, and then answers the new value. Any method that
1.16 Using the debugger

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 Playground, type `LOGame new openInHand` 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 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 code being executed in that method, with the part that triggered the error highlighted.
A first application

Figure 1-22 The debugger, with the method toggleNeighboursOfCell:at: selected.

Click on the line labeled LOGame >> toggleNeighboursOfCellAt: at: (near the top). The debugger will show you the execution context within this method where the error occurred (see Figure 1-22).

At the bottom of the debugger is a variable zone. 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. You can also see the values of the method arguments.

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 playground, and, most surprisingly to those used to other debuggers, change the code while it is being debugged! Some Pharoers 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. The syntax to return a value from a method in Pharo is `^`.
Listing 1-23  Fixing the bug.

```
| c origin |
c := LOCell new.
origin := self innerBounds origin.
sel addMorph: c.
c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
c mouseAction: [ self toggleNeighboursOfCellAt: i at: j ].
^ c
```

Listing 1-24  Overriding mouse move actions

```
[LOCell >> mouseMove: anEvent

Close the debugger window. Add the expression ^ c to the end of the method
LOGame >> newCellAt:at: so that it returns c.

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 CMD-Alt-Shift and click), and create a new one.

Execute LOGame new openInHand again because if you use the old game instance it will still contain the block with the old logic.

Now the game should work properly... or nearly so. If we happen to move the mouse between clicking and releasing, then the cell the mouse is over will also be toggled. This turns out to be behavior that we inherit from SimpleSwitchMorph. We can fix this simply by overriding mouseMove: to do nothing:

Finally we are done!

About the debugger.

By default when an error occurs in Pharo, the system displays a debugger. However, we can fully control this behavior. For example we can write the error in a file. We can even serialize the execution stack in a file, zip and re-open it in another image. Now when we are in development mode the debugger is available to let us go as fast as possible. In production system, developers often control the debugger to hide their mistakes from their clients.
1.17 **In case everything fails**

First do not stress! It is normal to mess up and there is no point to have Second if you do not succeed to delete the game. Try to get an inspector on any graphical element of the game using the halos: Option-Shift+Click and choose menu and the debug... menu and inspect Morph.

From there you can execute

- if you are inspecting the game itself: `self delete`.
- if you are inspect a game cell: `self owner delete`.

1.18 **Saving and sharing Pharo code**

Now that you have **Lights Out** working, you probably want to save it somewhere so that you can archive it and 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.

We’ll discuss the various ways to save and share code in a subsequent chapter, Chapter ???. For now, here is an overview of some of the available methods.

1.19 **Iceberg: Pharo and Git**

Iceberg is the new default tool for versioning your code using git and handling git repositories directly from Pharo images.

Declared repositories.

Iceberg is accessible through world menu *Tools > Iceberg*. When opened you will first see a *Repositories* screen. There you can find all git repositories managed by Iceberg. Do not care about the Pharo repository, it is there for people that want to contribute to Pharo.

Adding a new repository.

To manage our example with git and Iceberg, we should add it first. Press Add on the toolbar of *Repositories* screen. On the Figure 1-26 there are multiple options. The one we are interested in is **New repository**. Enter the project name like in the Figure 1-26, and if you wish you can leave the src blank. You can name your project PBE-LightsOut.
Adding a package.

After creating new repository you will notice that the Repositories screen has changed. It will contain newly created repository, with the "Not Loaded" status, which basically means, that now your repository is empty and that you should add packages to it (see Figure 1-27).

Now, to add packages, double click on the PBE-LightsOut repository, and in the new dialog press Add package in the toolbar. Select checkmark before the PBE-LightsOut package and press Add, it is also possible to add multiple packages by selecting checkmark before the package name. The repository now contains the added package, with the status Uncommitted changes, as shown in Figure 1-28.

Committing changes.

One more thing remains is to actually commit those changes. Select Commit on the toolbar to get the commit dialog. Here you can see all the files that are being committed, with a green plus next to their name (Figure 1-29). Before committing enter the meaningful message for that particular commit,
and finally press Commit. Note that Iceberg adds some metadata about the file format and the location of the code without the directory.

In this example, we used Iceberg to version our project using git. Have in mind that all committed changes, are performed on your local machine. To connect your local repository, to remote repository (e.g., GitHub) you need to add it, you can do that by selecting Repository on the Working copy dialog browser (Figure 1-28) and select the Add remote button on the top right. This is covered in Chapter ??.

1.20 Saving code in a file

If you do not want to use a version control system such Git you can save the code of a package, class, or method simply.

You can also save the code of your package by "filing out" the code. The
1.20 Saving code in a file

Figure 1-29  Iceberg working copy dialog.

Figure 1-30  File Out our PBE-LightsOut.

Right-click menu in the Package pane will give you the option to Extra > File Out the whole of package 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.

Right-click on the PBE-LightsOut package and file out the contents (see Figure 1-30). You should now find a file named 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 (see Figure 1-31) and fileIn. Verify that the game now works in the new image.
1.21 About Setter/Getter convention

If you are used to getters and setters in other programming languages, you might expect these methods to be called `setMouseAction` and `getMouseAction`. The Pharo 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 Pharo, all instance variables are private to the object that owns them, so the only way for another object to read or write those variables is through accessor methods like this one. In fact, the instance variables can be accessed in subclasses too.

1.22 On categories vs. packages

Historically, Pharo packages were implemented as "categories" (a group of classes). With the newer versions of Pharo, the term category is being deprecated, and replaced exclusively by package.

If you use an older version of Pharo or an old tutorial, the class template will be as follow:

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

It is equivalent to the one we mentioned earlier. In this book we only use the term *package*. The Pharo package is also what you will be using to ver-
1.23 Chapter summary

In this chapter you have seen how to create packages, classes and methods. In addition, you have learned how to use the System browser, the inspector, the debugger and Iceberg to version your code using git.

- Packages are groups of related classes.
- A new class is created by sending a message to its superclass.
- Protocols are groups of related methods inside a class.
- 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 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 by filing out a package, class or method.
- Using Iceberg and git.
Bibliography