CHAPTER 1

A first application

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 see the browser, the object inspector, the debugger and the Monticello package browser.

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.

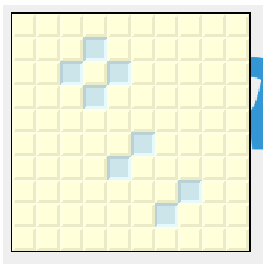
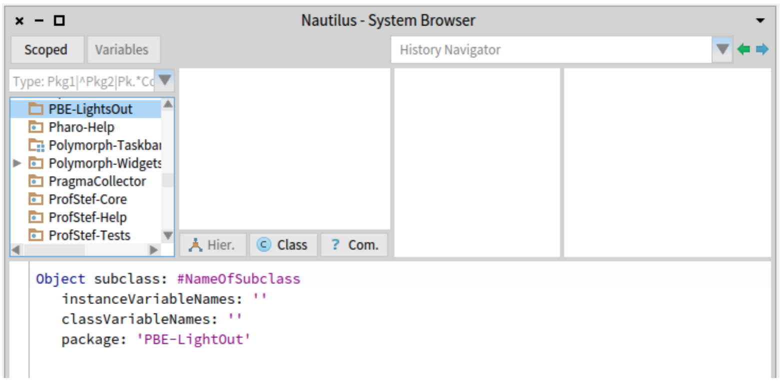


Figure 1.1: The Lights Out game board



```
Object subclass: #NameOfSubclass
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-LightOut'
```

Figure 1.2: Create a Package and class template

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.

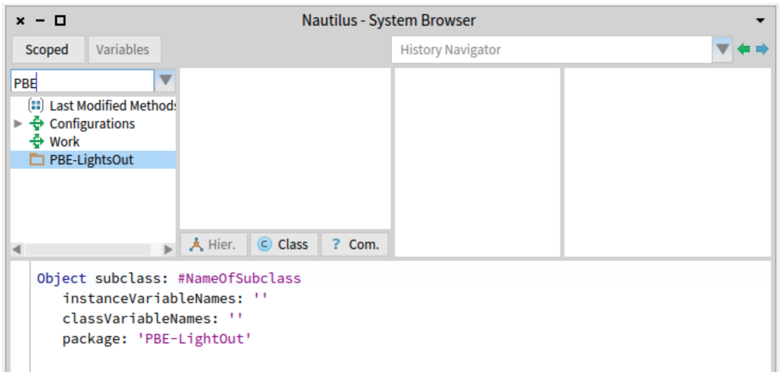
Lights Out is made up of two kinds of objects: the game board itself, and 100 individual cell objects. The Pharo code to implement the game will contain two classes: one for the game and one for the cells. We will now show you how to define these classes using the Pharo programming tools.

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 `Add 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).



```
Object subclass: #NameOfSubclass
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-LightOut'
```

Figure 1.3: Filtering our package to work more efficiently

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`.

Creating a new class

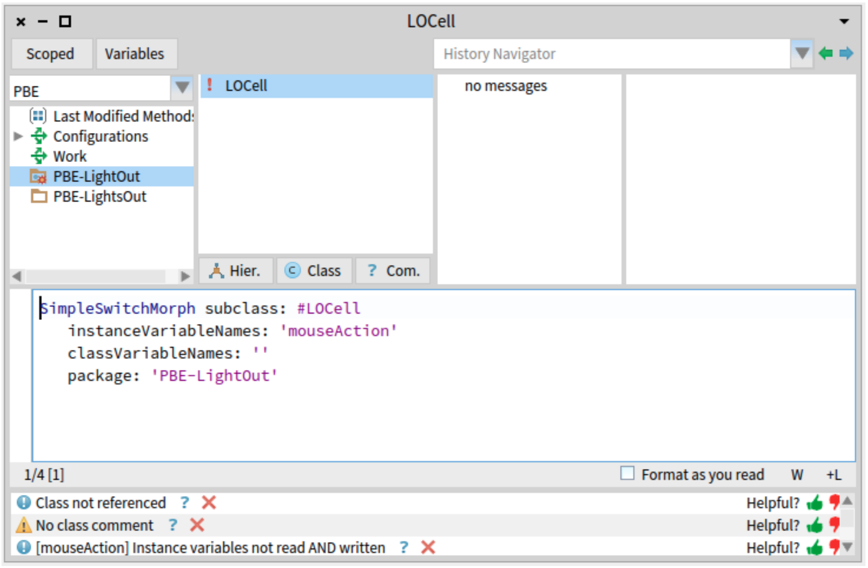
We simply edit the template to create the class that we really want. Modify the class creation template as follows:

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

You should get the following class definition:

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

This new definition consists of a Pharo 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 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



```
SimpleSwitchMorph subclass: #LOCell
    instanceVariableNames: 'mouseAction'
    classVariableNames: ''
    package: 'PBE-LightOut'
```

Class not referenced ?
No class comment ?
[mouseAction] Instance variables not read AND written ?

Figure 1.4: The newly-created class LOCell

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 1.4.

Once the class definition is accepted, the class is created and appears in the class pane of the browser (see Figure 1.4). 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.

About comments

Pharoers put a very high value on the readability of their code, but also good quality comments.

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.

Select the comment button and define a class comment following this template

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:

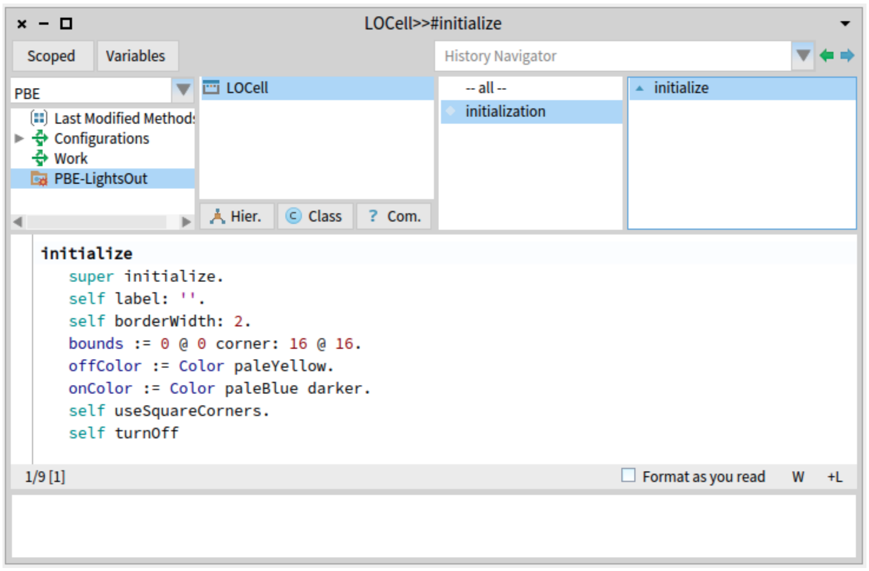
```
SimpleSwitchMorph subclass: #LOCell
   instanceVariableNames: 'mouseAction'
   classVariableNames: ''
   category: 'PBE-LightsOut'
```

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 version your source code using the Monticello versioning tool.

1.4 Adding methods to a class

Now let’s add some methods to our class. Select the protocol `'no messages'` in the protocol pane. 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):

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



```
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
```

Figure 1.5: The newly-created method `initialize`

```
    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. Another way to create an empty string is `String new`. Do not forget to **accept** this method definition.

Initialize methods. 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 1.5). This means the method exists in the superclass and is overridden in your class.

1.5 Inspecting an object

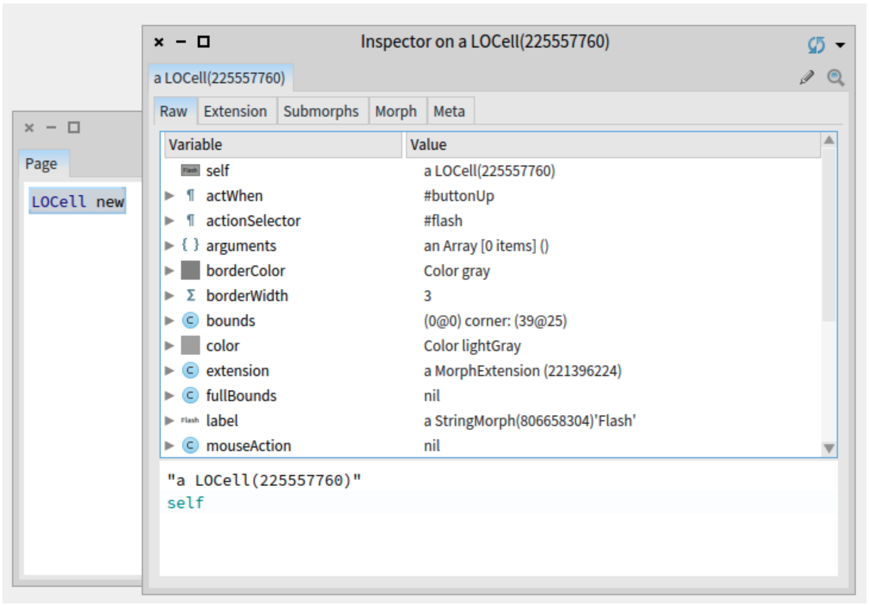
You can immediately test the effect of the code you have written by creating a new `LOCell` object and inspecting it: Open a `Playground`, type the expression `LOCell new`, and **Inspect** it (using the menu item with the same name).

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 1.6).

If you click on an instance variable the inspector will open a new pane with the detail of the instance variable (see Figure 1.7).

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.

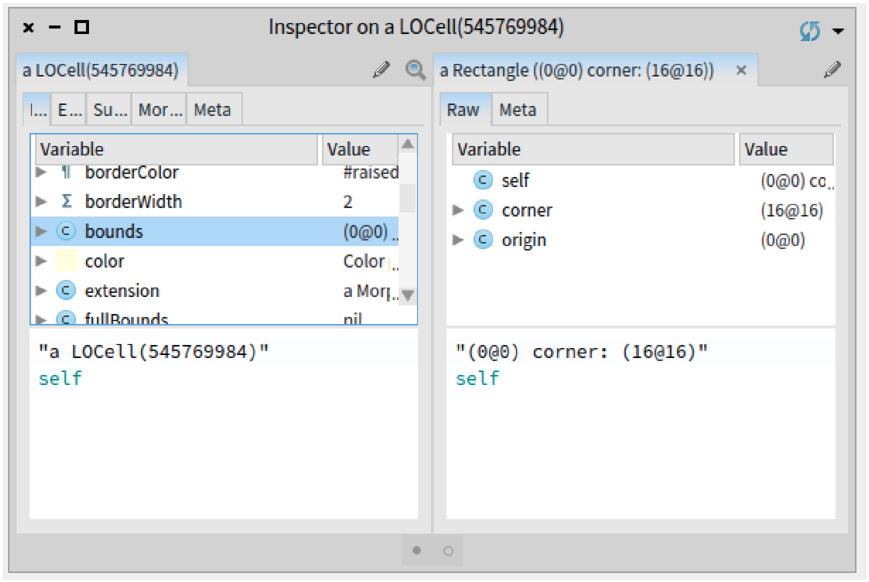
Go to that Playground at the bottom of the pane and type the text `self bounds: (200@200 corner: 250@250)` **Do it**. To refresh the values, click on the `update` button (the blue little circle) at the top right of the pane. The `bounds` variable should change in the inspector. Now type the text `self openInWorld` in the mini-playground and **Do it**.



Variable	Value
self	a LOCell(225557760)
actWhen	#buttonUp
actionSelector	#flash
arguments	an Array [0 items] ()
borderColor	Color gray
borderWidth	3
bounds	(0@0) corner: (39@25)
color	Color lightGray
extension	a MorphExtension (221396224)
fullBounds	nil
label	a StringMorph(806658304)'Flash'
mouseAction	nil

```
"a LOCell(225557760)"
self
```

Figure 1.6: The inspector used to examine a `LOCell` object


× − □
Inspector on a LOCell(545769984)
a LOCell(545769984)
I...
E...
Su...
Mor...
Meta
a Rectangle ((0@0) corner: (16@16))
×
Raw
Meta

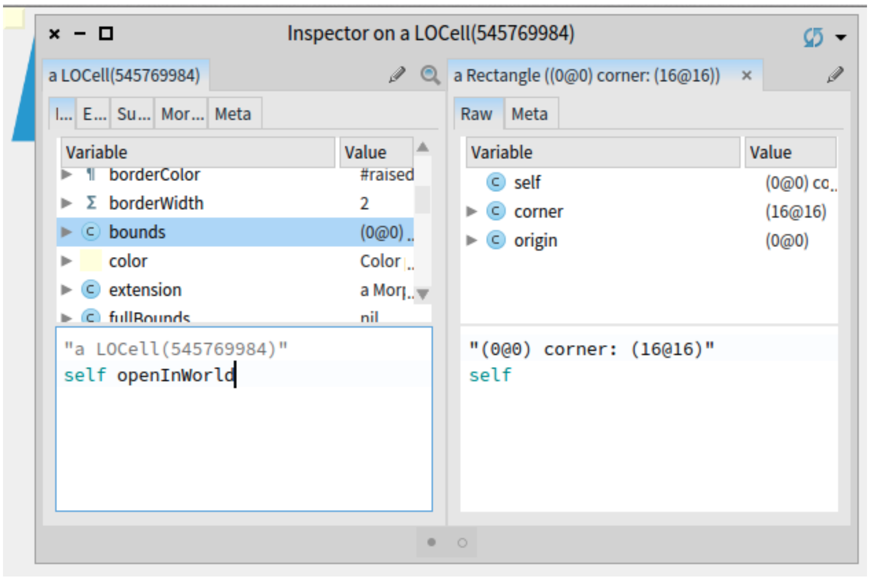
Variable	Value
borderColor	#raised
borderWidth	2
bounds	(0@0) ..
color	Color ..
extension	a Morp..
fullBounds	nil

```
"a LOCell(545769984)"
self
```

Variable	Value
self	(0@0) co..
corner	(16@16)
origin	(0@0)

```
"(0@0) corner: (16@16)"
self
```

Figure 1.7: When we click on an instance variable, we inspect its value (another object)



```
"a LOCell(545769984)"
self openInWorld
```

```
"(0@0) corner: (16@16)"
self
```

Figure 1.8: An LOCell open in world

The cell should appear near the top left-hand corner of the screen (as shown in Figure 1.8) 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.

1.6 Defining the class LOGame

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

Class creation

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.

```
BorderedMorph subclass: #LOGame
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'PBE-LightsOut'
```

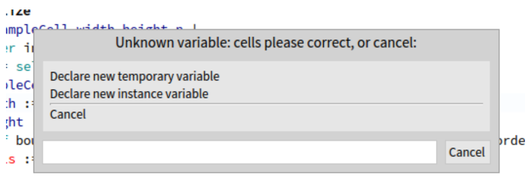

Unknown variable: cells please correct, or cancel:

Figure 1.9: Declaring `cells` as a new instance variable

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 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.

Initializing our game

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.

```
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 ]
```

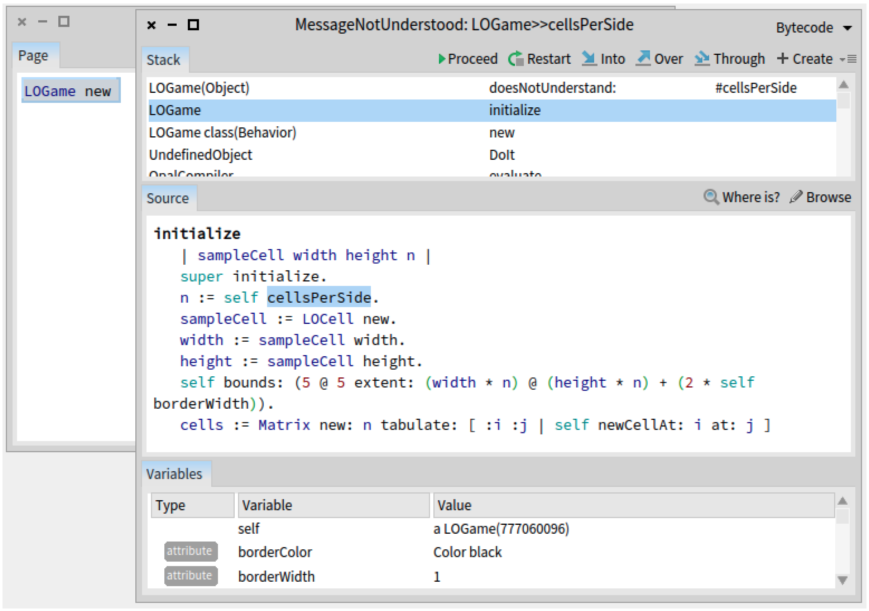
Pharo will complain that it doesn't know the meaning of `cells` (see Figure 1.9). It will offer you a number of ways to fix this.

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

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 1.10). 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**, the class which will contain the method, click on **ok**, then when prompted for



LOGame(Object)	doesNotUnderstand:	#cellsPerSide
LOGame	initialize	
LOGame class(Behavior)	new	
UndefinedObject	DoIt	

```
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 ]
```

Type	Variable	Value
	self	a LOGame(777060096)
attribute	borderColor	Color black
attribute	borderWidth	1

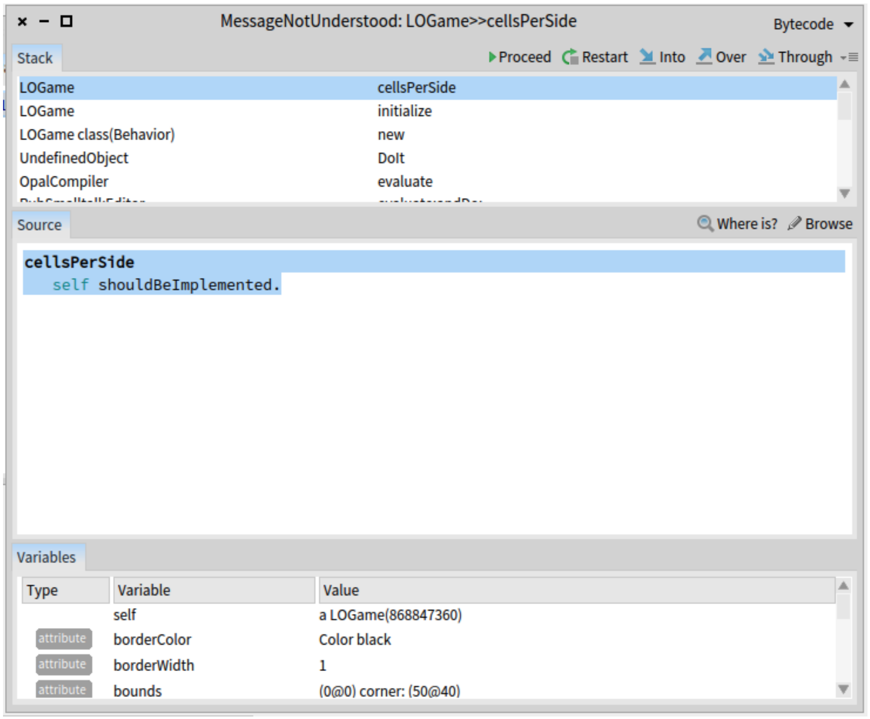
Figure 1.10: Pharo detecting an unknown selector

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 1.11) 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.

```
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 1.12. 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 **System Browser**), you will see that the browser has modified it to include the instance variable `cells`.



MessageNotUnderstood: LOGame>>cellsPerSide

LOGame	cellsPerSide
LOGame	initialize
LOGame class(Behavior)	new
UndefinedObject	DoIt
OpalCompiler	evaluate

```
cellsPerSide
   self shouldBeImplemented.
```

Type	Variable	Value
	self	a LOGame(868847360)
attribute	borderColor	Color black
attribute	borderWidth	1
attribute	bounds	(0@0) corner: (50@40)

Figure 1.11: The system created a new method with a body to be defined.

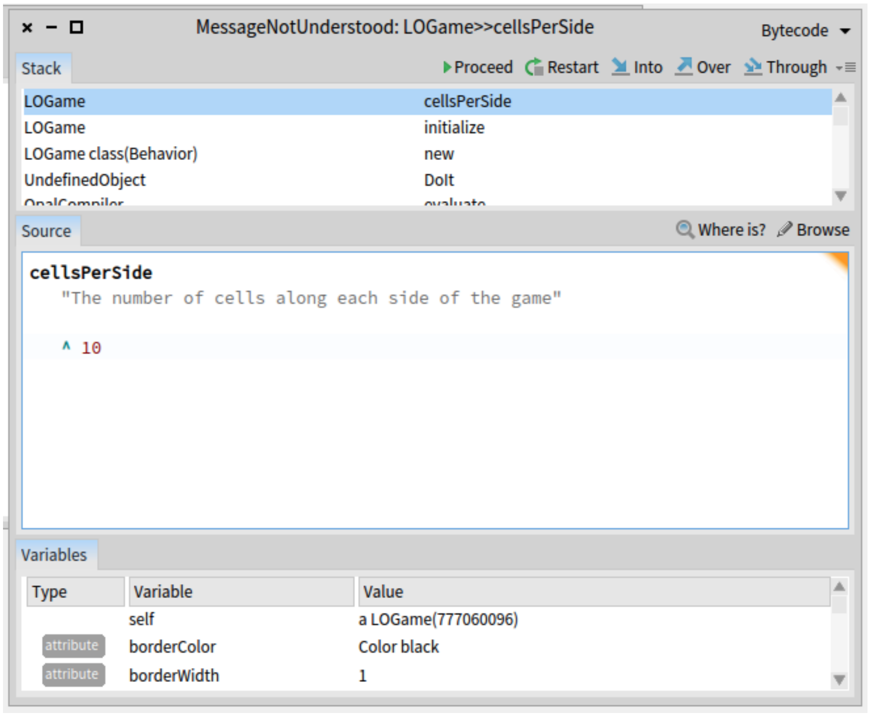
Studying the initialize method

Let us now study the method `initialize`.

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

- 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



LOGame	cellsPerSide
LOGame	initialize
LOGame class(Behavior)	new
UndefinedObject	DoIt

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

   ^ 10
```

Type	Variable	Value
	self	a LOGame(777060096)
attribute	borderColor	Color black
attribute	borderWidth	1

Figure 1.12: Defining cellsPerSide in the debugger

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.

- 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 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 (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 X 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`.

1.7 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, 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".

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 protocol pane may well contain the protocol **as yet unclassified**. Right-click in the protocol pane and select **categorize all uncategorized** to fix this, and move the `initialize` method to a new protocol called **initialization**.

How does the **System Browser** know that this is the right protocol? Well, in general Pharo can't know exactly, but in this case 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.

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`". (Incidentally, 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.

1.8 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.

```
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 ]
```

Formatting. As you can see there are some tabulation and empty lines. In order 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.

The method defined above created a new `LOCell`, initialized 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.

```
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 ]
```

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

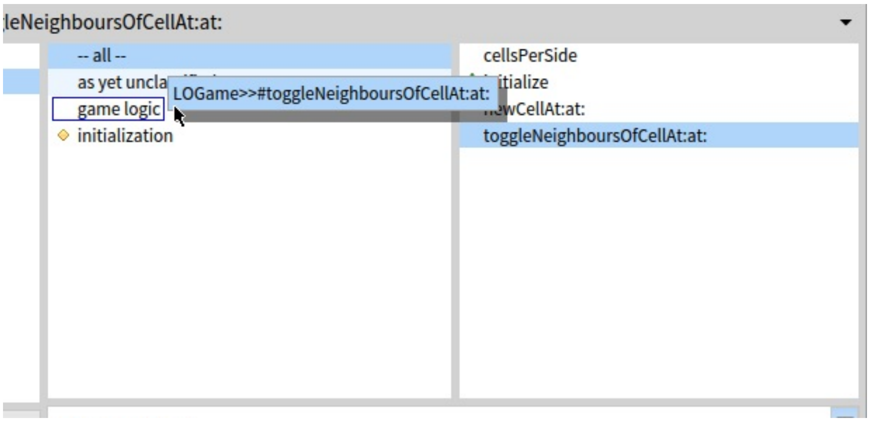


Figure 1.13: Drag a method to a protocol

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 1.13).

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

```
LOCell >> mouseAction: aBlock
   ^ mouseAction := aBlock
```

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

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.

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 cursor is over this cell on the screen. Add the method `LOCell >> mouseUp:` and then **Categorize automatically** the methods.

```
LOCell >> mouseUp: anEvent
   mouseAction value
```

What this method does is to send the message `value` to the object stored in the instance variable `mouseAction`. In `LOGame >> newCellAt: i at: j` we created the *block* `[self toggleNeighboursOfCellAt: i at: j ]` which is toggling all the neighbours of a cell and we assigned this block to the `mouseAction` of the cell. Therefore sending the `value` message causes this block to be evaluated, and consequently the state of the cells will toggle.

1.9 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 Playground, type `LOGame new openInWorld` and **Do it** .

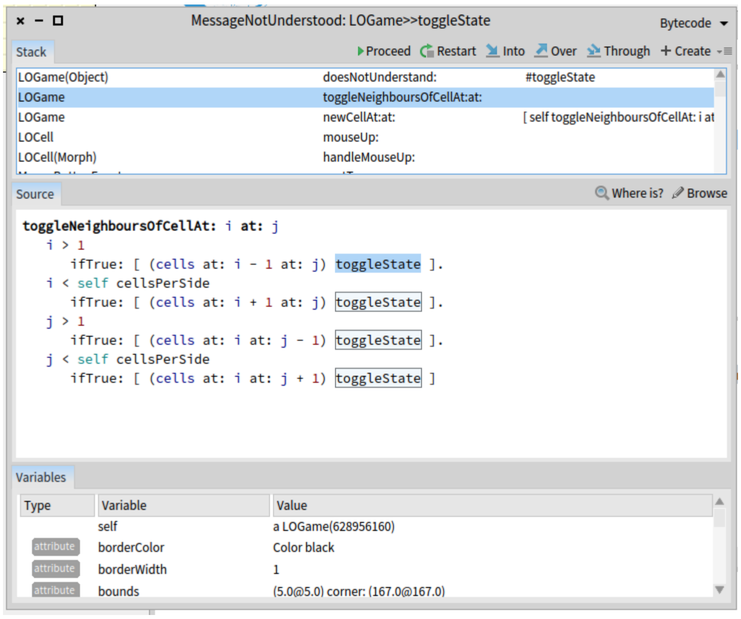
The game will open, and you should be able to click on the cells and see how it works. Well, so much for theory... When you click on a cell, a debugger will appear. In the upper part of the debugger window you can see the execution stack, showing all the active methods. Selecting any one of them will 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 labeled `LOGame >> toggleNeighboursOfCellAt: at:` (near the top). The debugger will show you the execution context within this method where the error occurred (see Figure 1.14).

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



```
LOGame(Object)    doesNotUnderstand:          #toggleState
LOGame            toggleNeighboursOfCellAt:at:
LOGame            newCellAt:at:               [ self toggleNeighboursOfCellAt: i at
LOCell            mouseUp:
LOCell(Morph)     handleMouseUp:
```

```
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 ]
```

Type	Variable	Value
	self	a LOGame(628956160)
attribute	borderColor	Color black
attribute	borderWidth	1
attribute	bounds	(5.0@5.0) corner: (167.0@167.0)

Figure 1.14: The debugger, with the method `toggleNeighboursOfCell:at:` selected

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 `^`.

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

```
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
```

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 openInWorld` 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 `SimpleSwitch-Morph`. We can fix this simply by overriding `mouseMove:` to do nothing:

```
LOCell >> mouseMove: anEvent
```

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 reopen 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.

1.10 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.

> **Note** We'll be discussing the various ways to save and share code in a future chapter, Chapter : Sharing Code and Source Control. For now, here is an overview of some of the available methods.

Saving plain code

The simplest way of doing this is by "filing out" the code. The right-click menu in the Package pane will give you the option to `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.

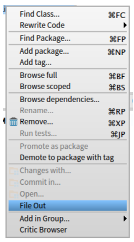
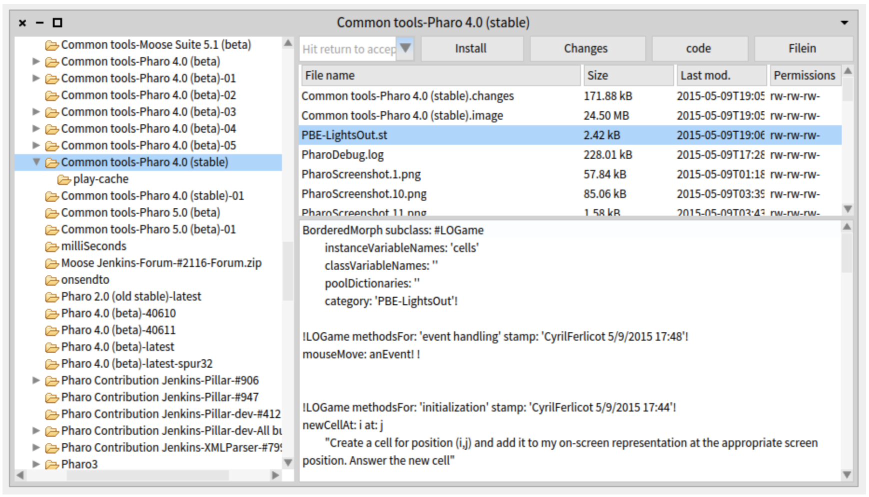


Figure 1.15: File Out our PBE-LightsOut


Common tools-Pharo 4.0 (stable)
Common tools-Moose Suite 5.1 (beta)
Common tools-Pharo 4.0 (beta)
Common tools-Pharo 4.0 (beta)-01
Common tools-Pharo 4.0 (beta)-02
Common tools-Pharo 4.0 (beta)-03
Common tools-Pharo 4.0 (beta)-04
Common tools-Pharo 4.0 (beta)-05
Common tools-Pharo 4.0 (stable)
play-cache
Common tools-Pharo 4.0 (stable)-01
Common tools-Pharo 5.0 (beta)
Common tools-Pharo 5.0 (beta)-01
milliSeconds
Moose Jenkins-Forum-#2116-Forum.zip
onsendto
Pharo 2.0 (old stable)-latest
Pharo 4.0 (beta)-40610
Pharo 4.0 (beta)-40611
Pharo 4.0 (beta)-latest
Pharo 4.0 (beta)-latest-spur32
Pharo Contribution Jenkins-Pillar-#906
Pharo Contribution Jenkins-Pillar-#947
Pharo Contribution Jenkins-Pillar-dev-#412
Pharo Contribution Jenkins-Pillar-dev-All bu
Pharo Contribution Jenkins-XMLParser-#79
Pharo3
Hit return to accep
Install
Changes
code
Filein

File name	Size	Last mod.	Permissions
Common tools-Pharo 4.0 (stable).changes	171.88 kB	2015-05-09T19:05	rw-rw-rw-
Common tools-Pharo 4.0 (stable).image	24.50 MB	2015-05-09T19:05	rw-rw-rw-
PBE-LightsOut.st	2.42 kB	2015-05-09T19:06	rw-rw-rw-
PharoDebug.log	228.01 kB	2015-05-09T17:28	rw-rw-rw-
PharoScreenshot.1.png	57.84 kB	2015-05-09T01:18	rw-rw-rw-
PharoScreenshot.10.png	85.06 kB	2015-05-09T03:39	rw-rw-rw-
PharoScreenshot.11.png	1.58 kB	2015-05-09T03:43	rw-rw-rw-

```
BorderedMorph subclass: #LOGame
    instanceVariableNames: 'cells'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE-LightsOut'!

!LOGame methodsFor: 'event handling' stamp: 'CyrilFerlicot 5/9/2015 17:48'!
mouseMove: anEvent! !


!LOGame methodsFor: 'initialization' stamp: 'CyrilFerlicot 5/9/2015 17:44'!
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"
```

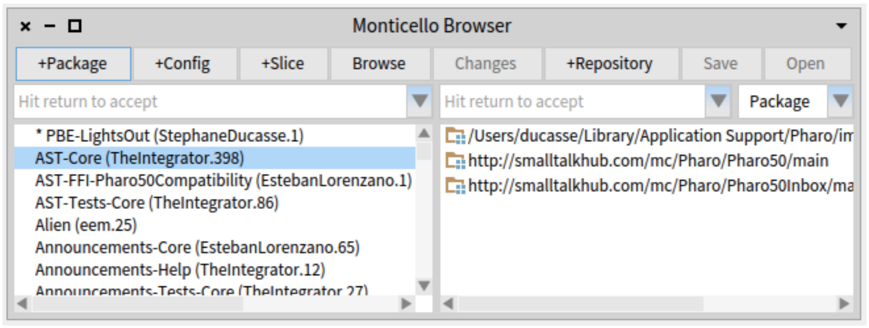
Figure 1.16: Import your code with the file browser

Right-click on the PBE-LightsOut package and file out the contents (see Figure 1.15). 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 1.16) and fileIn. Verify that the game now works in the new image.

Monticello packages

Although fileouts are a convenient way of making a snapshot of the code you have written, they are definitevely ”old school”. Just as most open-source projects find it much more convenient to maintain their code in a repository using SVN or Git, so Pharo programmers find it more convenient to manage their code using Monticello packages. These packages are represented as files



```
* PBE-LightsOut (StephaneDucasse.1)
AST-Core (TheIntegrator.398)
AST-FFI-Pharo50Compatibility (EstebanLorenzano.1)
AST-Tests-Core (TheIntegrator.86)
Alien (eem.25)
Announcements-Core (EstebanLorenzano.65)
Announcements-Help (TheIntegrator.12)
Announcements-Tests-Core (TheIntegrator.27)
```

```
/Users/ducasse/Library/Application Support/Pharo/im
http://smalltalkhub.com/mc/Pharo/Pharo50/main
http://smalltalkhub.com/mc/Pharo/Pharo50Inbox/ma
```

Figure 1.17: Monticello browser. The package PBE-LightsOut is not save yet

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 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.

Monticello Browser

Open the Monticello browser from the World menu (see Figure 1.17).

In the right-hand pane of the browser 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. The top list item is a local directory called the `package-cache`. It 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.

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

SmalltalkHub: a Github for Pharo

We think that the best way to save your code and share it is to create an account for your project in SmalltalkHub. SmalltalkHub is like GitHub: it is a web front-end to a HTTP Monticello server that lets you manage your projects. There is a public server at http://www.smalltalkhub.com/.

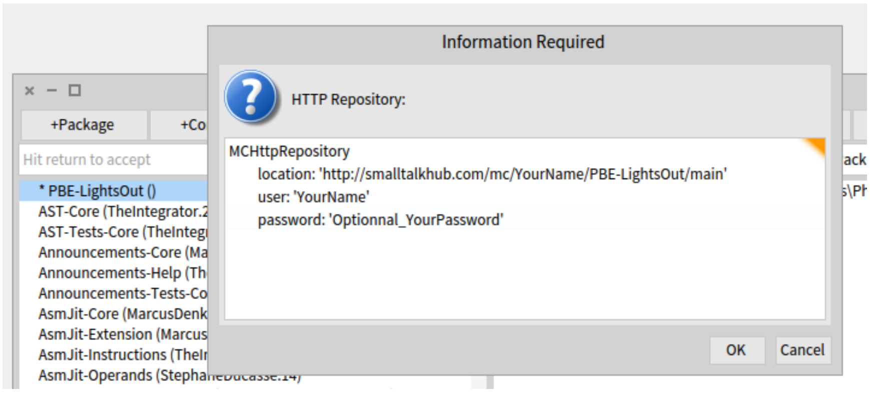
To be able to use SmalltalkHub you will need an account. Open the site in your browser. Then, click on the **Join** link and follow the instructions to create a new account. Finally, **login** to your account. Click on the **+ New project** to create a new project. Fill in the information requested and click **Register project** button. You will be sent to your profile page, on the right side you will see the list of your projects and projects you watch by other coders. Click on the project you just created.

Under **Monticello registration** title label you will see a box containing a smalltalk message similar to

```
MCHttpRepository
    location: 'http://www.smalltalkhub.com/mc/UserName/ProjectName/main'
    user: 'YourName'
    password: 'Optional_Password'
```

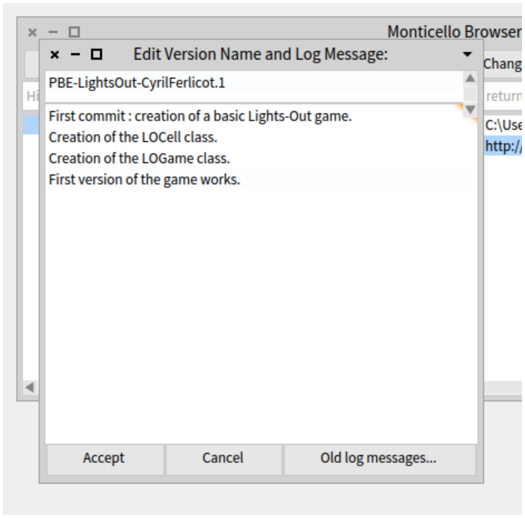
Copy the contents and go back to Pharo. Once your project has been created on SmalltalkHub, you have to tell your Pharo system to use it. With the `PBE-LightsOut` package selected, click the **+Repository** button in the Monticello browser. You will see a list of the different Repository types that are available. To add a SmalltalkHub repository select `smalltalkhub.com`. You will be presented with a dialog in which you can provide the necessary information about the server. You should paste the code snippet you have copied from SmalltalkHub (see Figure 1.18). This message tells Monticello where to find your project online. You can also provide your user name and password. If you do not, then Pharo will ask for them each time you try to save into your online repository at SmalltalkHub.

Once you have accepted, your new repository should be listed on the right-hand side of the Monticello browser. Click on the **Save** button to save a first version of your Lights Out game on SmalltalkHub. Don't forget to put a com-



```
MCHttpRepository
	location: 'http://smalltalkhub.com/mc/YourName/PBE-LightsOut/main'
	user: 'YourName'
	password: 'Optionnal_YourPassword'
```

Figure 1.18: Create your first repository on SmalltalkHub



PBE-LightsOut-CyrilFerlicot.1

First commit : creation of a basic Lights-Out game.
Creation of the LOCell class.
Creation of the LOGame class.
First version of the game works.

Figure 1.19: Do your first commit

ment so that you, or others, will have an idea of what your commit contains (see Figure 1.19).

To load a package into your image, open Monticello , select the repository where the package is located and click open button. It will open a new window with two columns, left one is the Package to be loaded, the right one is the version of the package to be loaded. Select the Package and the version you want to load and click the load button.

Open the `PBE-LightsOut` repository you have just saved. You should see the package you just saved.

Monticello has many more capabilities, which will be discussed in depth in Chapter : Sharing Code and Source Control.

About Git. If you are already familiar with Git and Github there are several solutions to manage your projects with Git. This blog posts provides a summary: http://blog.yuriy.tymch.uk/2015/07/pharo-and-github-versioning-revision-2.html#! There is a chapter in preparation on about how to use Git with Pharo. Request it on the Pharo mailing-list.

1.11 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 the Monticello browser.

- 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.
- A better way to share code is to use Monticello to manage an external repository, for example defined as a SmalltalkHub project.