Pharo 9 by Example

Stéphane Ducasse and Gordana Rakic with Sebastijan Kaplar and Quentin Ducasse

February 27, 2022

Chapter 6
A first application

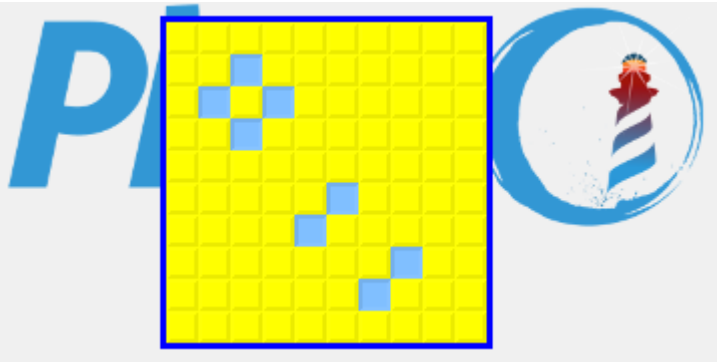

Figure 6-1 The Lights Out game board

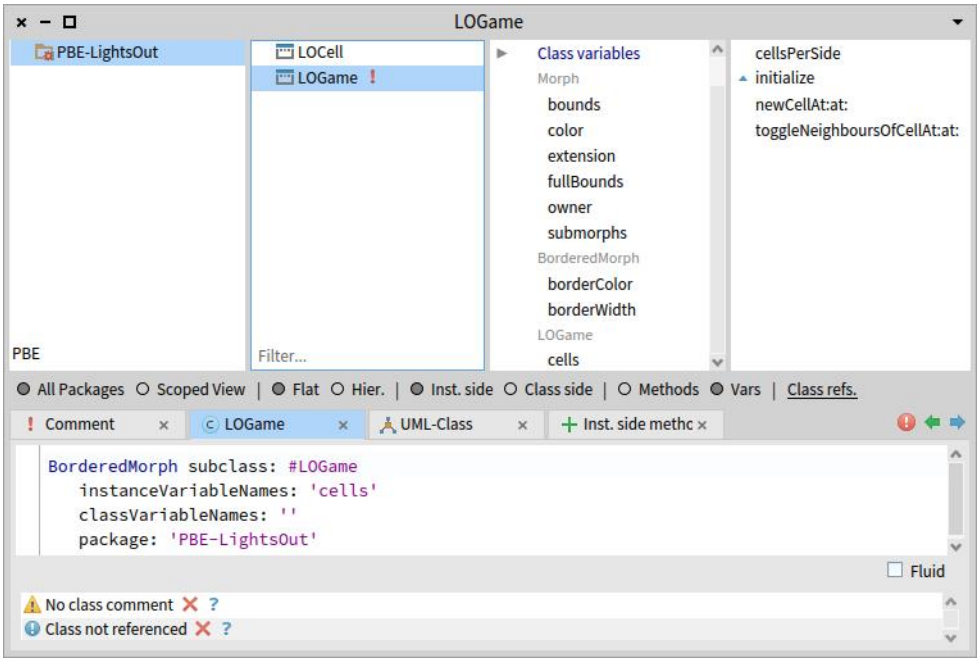

Figure 6-15 The LOGame's instance variables

6.17. Improving the game

We have just finished, and we have already thought about improving our product. Is it convenient for you to close the game with the Morphic halo? We don't either. It is not possible to aim the mouse pointer at the boundary of the game board... It seems worth making it wider. But a wide black frame will look sad, so let's change its color as well. To achieve this, you will need to conduct a small study of the class hierarchy from which the `LOGame` class inherits.

Change the *initialize* method

The values of the thickness of the border and its color should be stored somewhere. Let's find out which instance variables the class `LOGame` inherited. Open the class

browser on `LOGame` and select *Vars* on the browser's "***Methods / Vars***" toggle (below the protocol pane). The protocol pane will display a list of all instance variables, structured by the classes in which these variables are declared, as shown in Figure 6-15. We see that the `LOGame` superclass contains exactly the variables we need: `borderColor` and `borderWidth`.

At once there is a temptation to set these variables desirable values in the `LOGame >> initialize` method. For example, as follows:

```
initialize
  | sampleCell width height n |
  super initialize.
  borderColor := Color blue.
  borderWidth := 3.
  n := self cellsPerSide.
  . . .
```

If we would save the changes and run the game, we would see that everything works, the game has acquired the same look as in Figure 6.1. But not everything is as good as it may seem at first glance. After all, the variables `borderColor` and `borderWidth` are declared in the class `BorderedMorph` which must set their initial values. It looks like we've re-initialized the `borderColor` and `borderWidth` variables. This is twice wrong: redundant work is done, and new values set in a subclass can cause unexpected errors. Let's continue exploration the code and come up with the better way to achieve what we want.

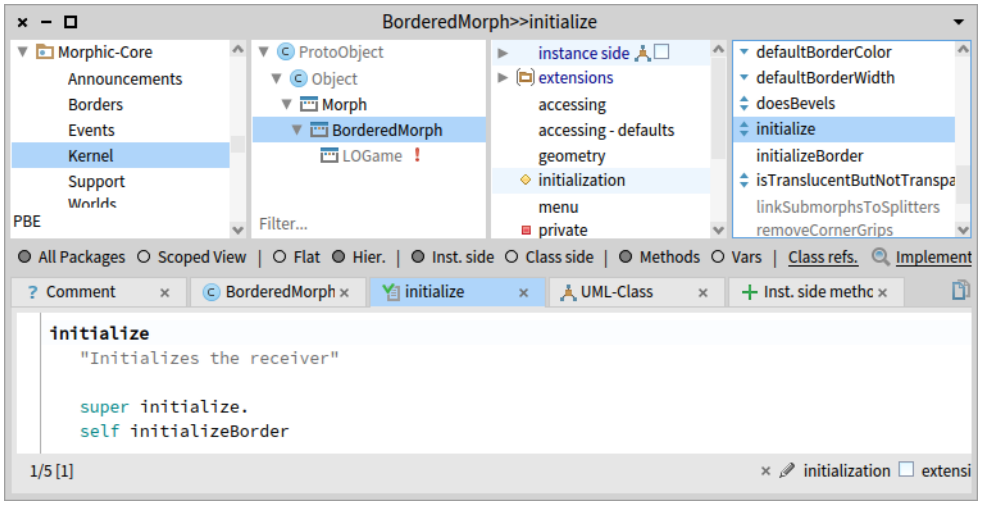

Figure 6-16 LOGame inheritance hierarchy

The initial values of the `borderColor` and `borderWidth` variables should be specified by the `BorderedMorph >> initialize method`. Find it in the class browser. Below the class pane of the Browser is the "*Flat / Hier.*" toggle, which hides or shows in the pane a class inheritance hierarchy regardless of their package membership. Select ***Hier.***, click the `BorderedMorph` class, then the `initialization` protocol and the `initialize` method. You should proceed to the superclass initialization method, as shown in Figure 6-16.

It is easy to guess from the text of the method that the method `BorderedMorph >> initializeBorder` deals with the variables `borderColor` and `borderWidth`. And in the body of the last one we find:

```
BorderedMorph >> initializeBorder
      "Initialize the receiver state related to border."

      borderColor:= self defaultBorderColor.
      borderWidth := self defaultBorderWidth
```

Now everything becomes clear: the values of color and border thickness are set by the corresponding methods of the `BorderedMorph` class. If we want to specify other values in our subclass, we need to override the `defaultBorderColor` and `defaultBorder-Width` methods. Recall that in the Pharo the pseudo-variable `self` refers to the receiver of the message (the executor of the method) and serves as a tool with which the class can send messages to its subclasses, even not written yet.

So, let's add two new methods to the initialization protocol of the LOGame class (and remove assignments to the variables in the `LOGame >> initialize`).

```
LOGame >> defaultBorderColor
      "answer the default border color for the game board"
      ^ Color blue

LOGame >> defaultBorderWidth
      "answer the default border width for the game board"
      ^ 3
```

Now the game works as it should, and we are convinced once again that defining constants as methods is the right approach. It is easy to modify such values in subclasses without breaking code of other methods.

Change the run

We can run all components of Pharo programmatically, or by a menu command, or even by a keyboard shortcut. And we run our game only programmatically (from the Playground). Let's correct this injustice and add the appropriate command to some menu! For example, to the World Menu. We don't know how to do that yet, but Pharo is an open system, so we're trying.

Open Spotter and type "world menu" in the search bar. The first link in the search results – Breakpoint class >> # debugWorldMenuOn: – shows how system classes add commands to the menu. At the end of the list of results there is a link to the "World Menu Items" section of the help system – this is exactly what we need. Spotter itself will not open the help, but it will not be difficult for us to find it through the command "Help> Help browser" of the World Menu. The section claims that in order to add a new item to the World Menu, we need to define a class method called `menuCommandOn:`. The help also provides examples of declaring such a method in some imaginary class. It will also be useful to look for real classes in Pharo that implement the `menuCommandOn:` method and see how it is done.

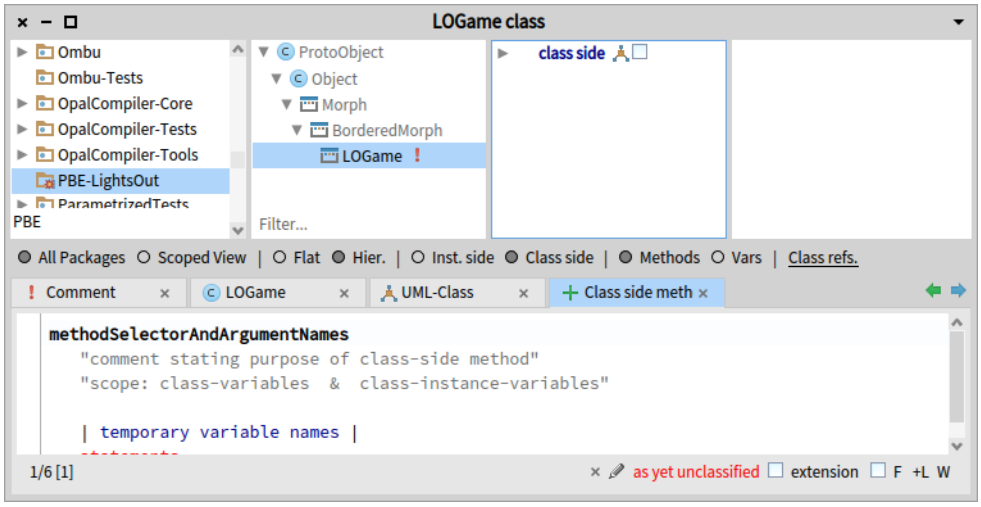

Figure 6-17 Definition of a class side method

Let's think about what we want the menu command to perform. It should execute the code we typed earlier in the Playground to create and run an instance of the Lights Out game. But it would be good to have a command to close the game as well. Occupying two lines of the main menu is too much luxury, so combine the opening and closing commands into one submenu.

Click in the Browser `LOGame` and select ***Class side*** in the "Inst. side / Class side " toggle (see Figure 6-17). The Browser will display a list of methods of the LOGame class (it is still empty). Pharo classes are also objects that can have their own methods. We need to define the method listed in Listing 6.10.

Listing 6.10

```
LOGame class>>menuCommandOn: aBuilder
      <worldMenu>
      (aBuilder item: #LightsOut)
         order: 5.0;
         withSeparatorAfter;
         with: [ (aBuilder item: #Run)
               order: 0;
               action: [ self open ].
            (aBuilder item: #Quit)
               order: 1;
               action: [ self close ] ]
```

The `<worldMenu>` pragma indicates the special purpose of the method (we'll talk about pragmas later), the message `item:` creates a menu item called "LightsOut" and all other messages are sent to this menu item. Note that the lines of code end with a semicolon. This is a sequence of messages or cascade in Pharo. Each a message in the cascade comes to the same receiver (to the menu item in our case). The message `order:` specifies the location in the menu, `withSeparatorAfter` separates the item from next parts of the menu by line, and `with:` specifies the structure of the submenu. Submenu items are described similarly, only `action: aBlock` is used instead of `with:` to set the menu response on command selection.

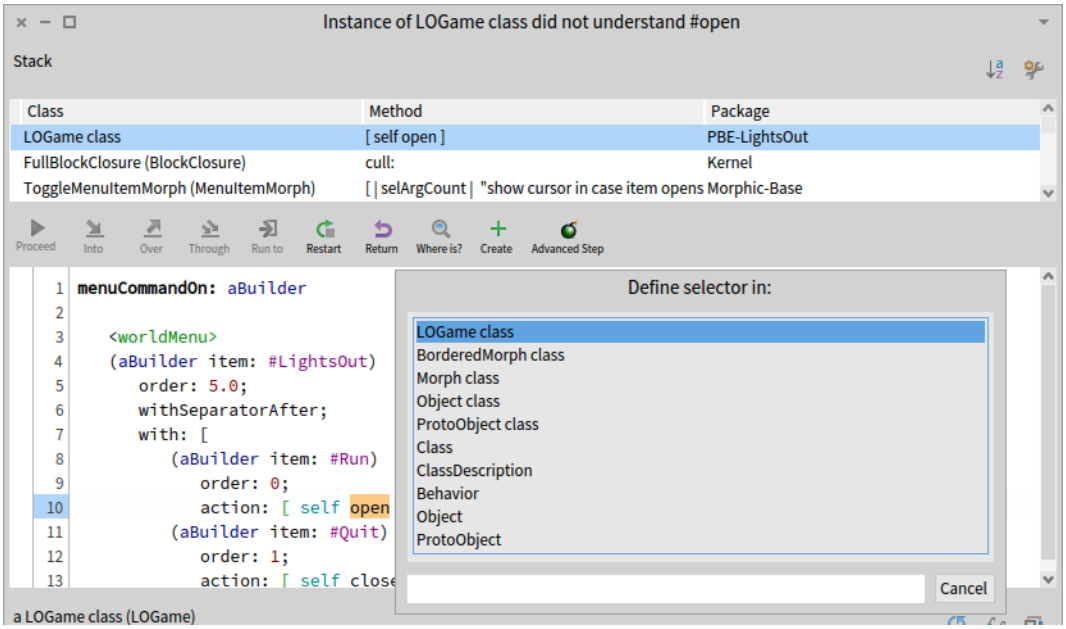

Figure 6-18 A class method creation with help of Debugger

Obviously, the menu command "Run" should call `LOGame class >> open`, and the command "Quit" should call `LOGame class >> close`. We have not defined such class methods yet, but we can do so in the Debugger, as we have defined instance methods earlier. So, let's open World Menu and select "LightsOut> Run"! Predictably, the Debugger will greet us with an error message. We know where it happened, so we can immediately click ***Create***, select the class `LOGame class` (see Fig. 6-18), select "instance creation" protocol and type the text of the method.

The known code "`LOGame new openInHand`" creates an unnamed object that can be interacted with only on the screen by the mouse. How will the closing command find it? Perhaps, we need to save a reference to the instance of the game in a variable so we will be able to send it a message programmatically. Let's use a class variable. (Pharo classes have their own variables!) We will call it "`TheGame`".

Enter the text of the method from Listing 6.11 into Debugger's code editor pane.

Listing 6.11 The game instance creation

```
LOGame class >> open
  TheGame ifNil: [
     TheGame := self new.
     TheGame openInHand ]
```

Listing 6.12 The game instance deleting

```
LOGame class >> close
  TheGame ifNotNil: [
     TheGame delete.
     TheGame := nil ]
```

Accept your code and the Debugger will ask you how to declare the name `TheGame`. Select "*Declare new class variable*". The method is defined so that only one instance of the game can be opened. The reference to it will be stored by a class variable. By the

way, look for the `LOGame` definition in the Browser (you need to go back to ***Inst. Side***). It should look like this:

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

Debugger added the name of the class variable `TheGame` as argument in the third line.

Click the Debugger's ***Proceed*** button and it will close, and a game window will appear! Play for a moment, but we need to determine another method. So, choose "World > LightsOut > Quit". Our old friend Debugger will re-report the missing method and help us to create it (see Listing 6.12). The method can be attributed to the releasing protocol.

Click on Proceed again – both the Debugger and the game will close. The World Menu commands will now work properly. You can test their effect (see Fig. 6-19).

Is the game over? Maybe so. But you may want to keep game statistics: count the number of clicks made, the number of cells switched off, and so on. It would also be good to display these statistics on the screen, but we will leave further improvements to the game to readers as an exercise.

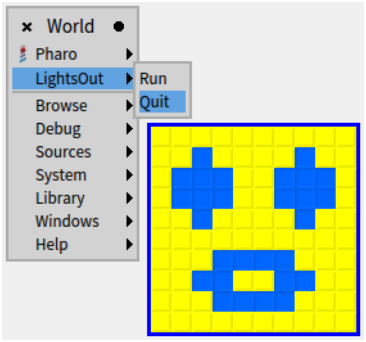

Figure 6-19 Extended World Menu and the game