Laser Game Tutorial: A Development Example for Pharo

Ben Coman Damien Cassou Stéphane Ducasse
Steven Wessels

Version of 2014-06-08

Contents

Chapter 1

Introduction

This book is an adaptation for Pharo of the tutorial written originally by Stephan Wessels for Squeak and available at http://squeak.preeminent.org/tut2007/.

Stéphane Ducasse adapted this tutorial to Pharo (3.0). Here is a list of changes:

- Added Pharo installation instructions (see appendix)
- The flow of the tutorial has been adapted from place to place.
- New material and more explanation has been added such as design discussions, use of refactorings...
- Less screenshots have been used to avoid the obsolescence when the IDE will change.
- All the code has been typed and shown in the text because different outputs may place pictures in different places.

BTC: I don't understand the point above. (and btw can the notes be made to stand out more)

- Figures are now cross referenced.
- Topics on package management and coding in the debugger have been added.
- All the code has been adapted to Pharo and is available from SmalltalkHub in the LaserGame project under StephaneDucasse account (http://smalltalkhub.com/#!/~StephaneDucasse/LaserGame).

Many tutorials showcase just the technology, by presenting the shortest (unrealistic) development path to the final product. This tutorial takes anote nother path, documenting the warts and all development of a project from first concept through to deployment. Each step is described in detail, including (especially) when mistakes were made.

The aim here is to show how natural it is to start from a rough concept, then iterate over the design, refactoring the implementation with the confidence that comes from test driven development. This tutorial demonstrates not only how to program Pharo, but an effective way to evolve an application over time, which provides Pharo its agility for fast application development.

Stéphane really wants to thank Steven for his gift to the Pharo and Smalltalk community.

1.1 Conventions used in this book

A large part of this tutorial provides sample code and discussion of that code. Within the narrative, references to a class name or method name will look like **MyClass** and **myMethod** respectively. To efficiently specify which class a method belongs to we write **MyClass**>>**myMethod**.

Blocks of sample code are displayed as follows:

```
MyClass>>myMethod
  ^ 1 + 2.
```

Note the class name prefix and ">>" are not typed into the code pane. This is a documentation convention only. So the above sample code when entered into the system would look like this:

```
myMethod
  ^ 1 + 2.
```

The carat "↑" is the return symbol, here indicating a value of 3 is returned by the method. Note if there is no return symbol, self is returned by default.

BTC-2014-05-24 The carat symbol "↑" seems to have been Squeakified. How to fix?

1.2 License

Figure 1.1: Book License.

Chapter 2

Game Overview

We're going to write a new game for Pharo. The following ideas should be represented in our game.

The game will be played on a grid. We will imagine having a laser beam that can be activated by the user. This will fire into the grid from a specific location. The laser beam will always fire from the bottom edge underneath the first column of cells (marked by the arrow in Figure 2.1). As the laser beam traverses inside our grid it can hit deflecting mirrors. These mirrors will divert the laser beam's direction as it travels. Ultimately the beam should hit a target location inside our grid. Once the laser is fired we will see how the cells guide the laser to a destination.

The cells of our grid will either be blank, have a target (shown as a circle here) or a deflecting mirror. The mirrors may be oriented in either of two ways, leaning left or right.

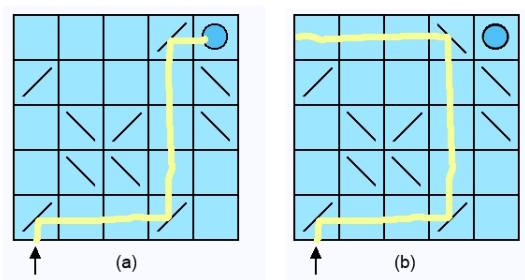

Figure 2.1: (a) Starting from its origin, the laser beam should reach the target. (b) Rotating a mirror changes the path of the laser beam.

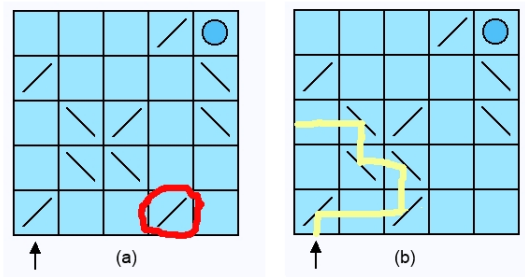

Figure 2.2: Finding a longer path, first move, slide mirror.

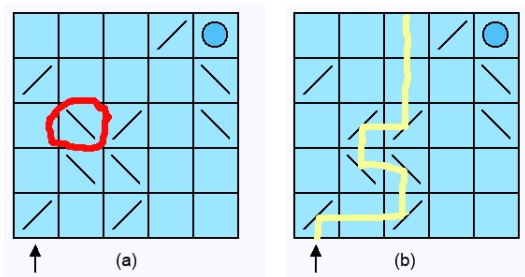

Figure 2.3: Finding a longer path, second move, rotate mirror.

Initial locations and orientations of the mirror cells will be randomly controlled by the game. In 2.1(a) the mirrors yield a correct result. However, the user has control over mirror rotation and position (to a certain amount). The user can click on a mirror cell and cause the mirror cell to rotate 90 degrees. This could alter the laser's direction as shown by Figure 2.1(b). When the laser hits a grid wall, its path ends.

Since, initially, this is a solitaire game, it becomes more interesting to have the user manipulate the mirrors to find the longest possible path to the target.

The user may click on a mirror cell and cause it to slide one grid-cell in a vertical or horizontal direction. Note that a mirror cell cannot move through the grid walls nor through another mirror cell nor the target cell. In this way the user can adjust the mirror cells to get an intermediate result as shown by Figure 2.2.

A further intermediate result is produced by rotating the mirror shown in Figure 2.3.

As a last step, one more mirror is moved to get the laser back to the target

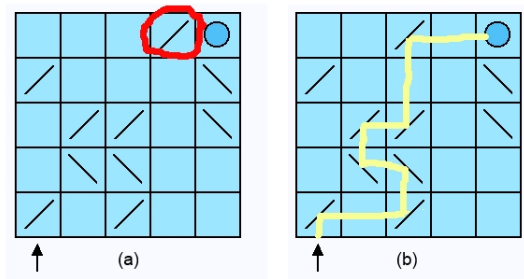

Figure 2.4: Finding a longer path, last move, rotate mirror.

cell as shown in Figure 2.4. This time the path of the laser is longer than before. And of course it was strictly a random coincidence that the initial cells configuration already provided a correct path for the laser.

That's the general idea. We can add laser cell-path counters and other game instrumentation as we develop.

Chapter 3

Discovery of Objects

When we look over the game drawings and think about what objects our game may need, a few come immediately to mind. There must be some kind of Grid and several Cells. There are different kinds of Cells too.

Cells and a Grid are obvious objects of the game and we'll probably discover other objects as we explore a little. It's perfectly fine to explore and then throw code away if we later learn we're not heading in the right direction.

Exploring with objects is easy to do with Pharo. We very much enjoy spending time thinking about designs but we've also learned you need to *dive in* sometimes to better understand the objects needed in your design.

As we once heard someone say, *You can read about swimming all you want. But, you'll never learn about swimming until you actually dive in the water.*

Of course we believe the water here is safe. The worst we could do is write some code and throw it away. And that's nothing to worry about. A per method version history in maintained you to quickly review and restore previous code. Development in Pharo encourages experimentation and quick idea exploration.

Let's have a look at the objects.

3.1 Grid

The Grid holds our Cells. It also contains the source of our laser beam. Let's go with the idea that the user will ask the Grid to fire our laser beam.

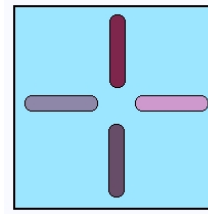

Figure 3.1: Blank Cell.

3.2 Cells

We identify three kinds of Cells.

- Blank, or empty
- Target
- Mirror

The basic responsibility of a Cell concerns what happens to the laser beam when it enters the Cell. Also, the Mirror Cells need to know something about their orientation. A Mirror Cells can be thought of as leaning "left" or leaning "right".

If we dig a little deeper into our understanding of these Cells we can imagine that each Cell has four internal line-segments. Something like an LED clock. We'll use these segments to indicate the path of the laser beam. Let's explore how that would work with our different Cell types.

The Blank Cell

We label the four segments **north, east, south**, and **west**. If the laser beam enters the **west** segment then we know the east segment will light-up. If the laser beam enters south then the **north** segment would also light-up as shown by Figure 3.1

The Mirror Cell

Depending on the orientation of the mirror we can also determine the path of the laser beam through the Cell. We identify the corresponding segments using the same technique.

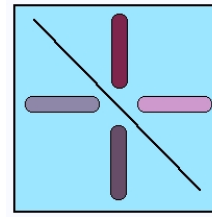

Figure 3.2: Left Leaning Mirror Cell.

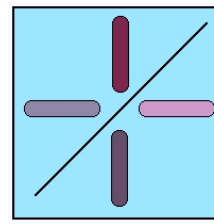

Figure 3.3: Right Leaning Mirror Cell.

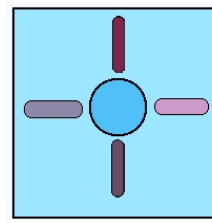

Figure 3.4: Target Cell.

For the left leaning Mirror Cell, if the laser beam enters the **west** segment we know the **south** segment will also light-up (See Figure 3.2). If the laser beam enters the right leaning Mirror Cell from the **west** it will exit by the **north** segment (See Figure 3.3).

The Target Cell

In the case of the Target Cell no other segment will light-up. The laser beam ends its path here (see Figure 3.4).

3.3 Identifying Classes

Visually, each of the three Cell types would render differently. So they can be seen to have some things in common and some things that are unique to each. Let's define our initial classes to be:

- **Grid**
- **BlankCell**
- **MirrorCell**
- **TargetCell**

We suspect that there may an abstract class that will unify common behavior between the classes. For now, let's not do that and just stick with these classes until the need to create other classes actually exists.

Instances of **Grid** will be responsible for our grid and overall management of the cells.

Instances of **BlankCell** will be the default condition in our grid. **MirrorCell** instances will not be as frequent and will also be contained in our grid. The **TargetCell** will have one instance in our grid.

3.4 Package Definition

First we should define a package to contain our classes. When we save a package all the classes will be saved together. It is a good practice to group together classes that work together.

Right-click on the background and open a System Browser.

In the package list of the System Browser, choose "add package" to add a new package. Use the name "LaserGame-Model".

You should get a system like that shown in Figure 3.5. Later we'll define two other packages to hold the tests and graphical elements of the game. We'll work out the presentation layer of our game (the graphics) later in the development process. For now we'll concern ourselves with the model.

3.5 Some Model Classes Creation

http://squeak.preeminent.org/tut2007/html/015.html

Now we are ready to define the first class of our model. Make sure that you selected the "LaserGame-Model" package where we want to define the

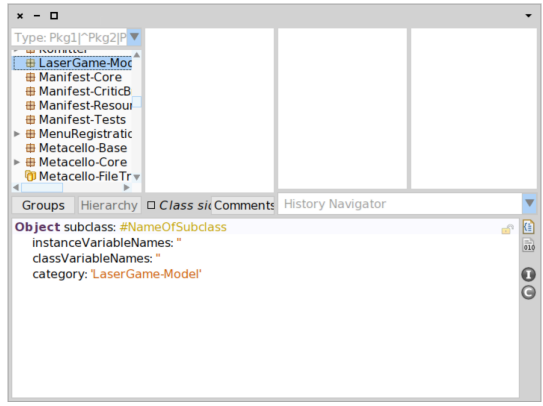

Figure 3.5: The class browser showing a newly created package.

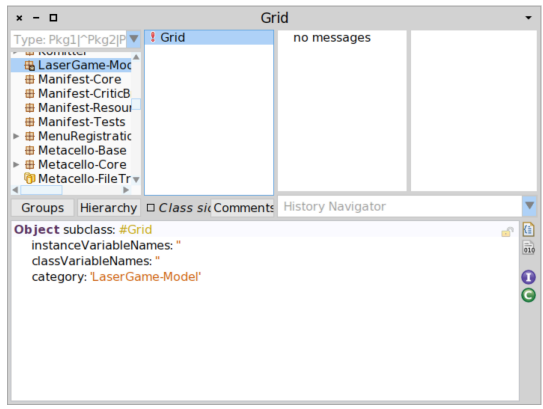

Figure 3.6: The class browser showing a newly created Grid class.

four model classes we identified before: **Grid**, **BlankCell**, **MirrorCell** and **TargetCell**.

Figure 3.5 shows a class creation template in the bottom pane of the Browser. Each class is required to have a superclass. For now the "easiest thing to do" for now is just subclass from **Object**. This will refactor later as the design structure is fleshed out.

Fill out the template as below, then right-click and choose to "accept" the class definition (or alternatively use the keystroke alt-s or cmd-s, depending on platform).

```
Object subclass: #Grid
```

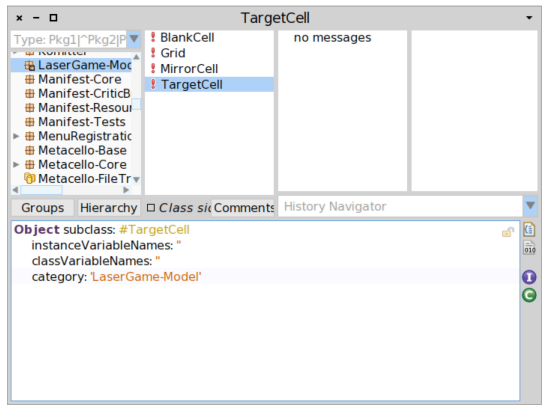

Figure 3.7: The class browser showing a newly created Grid class.

```
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

You result should be Figure 3.6. Note that for now we don't define any instance variables, just plain empty classes.

Then similarly define the following three classes.

```
Object subclass: #BlankCell
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

```
Object subclass: #MirrorCell
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

```
Object subclass: #TargetCell
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

Once you are done you should get a system similar to the one shown in Figure 3.7

Before continuing to define the behavior of our model, we will start to define tests to capturesuch behavior. It will help us to make sure that our

model is correct, it will document its behavior in a way that can be automatically checked.

Chapter 4

Test Driven Development

We will use the SUnit unit testing framework to implement the game model (have a look at the Pharo by Example SUnit Chapter for a deeper discussion). More than likely we'll not concern ourselves with writing unit tests for the actual behavior of the GUI because it is tedious. But there's plenty we can accomplish by unit test driving our development of the game model.

We are not too attached to how we code the very first few lines of code. An approach we've seen used by many people is to begin with the Unit Tests, even to the point of not having any objects to test before the first test is written. Another approach is to implement some basic model and then drive from that point forward using Unit Tests.

Now what is important to understand is that Unit tests accomplish several things:

- They help us to develop our objects because creating tests to exercise an object's api makes us consider how our objects will be used.
- They provide a consistency check as we continue to add and evolve code. This helps us to quickly identify when and where we break existing code, which is a tremendous help to stay on track.

Good unit tests capture your requirements and help you as you implement design. Often, when you write a unit test you are forced to think about your design as if it were finished.

4.1 Unit Test the Initial Model

The **BlankCell** is an excellent place to begin writing our unit tests. We want a new package to hold our test classes, which in turn will hold the methods

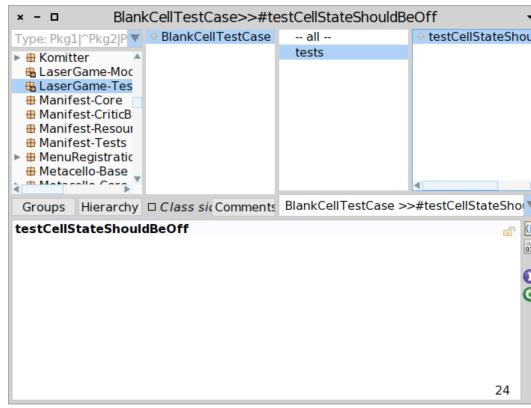

Figure 4.1: An empty test method.

defining our test cases. By convention the test classes will have "Test" appended to the name of the class being tested. So we'll define a new package *LaserGame-Tests* and inside it a new class **BlankCellTest**.

Right-click on the packages list and choose "Add package…", type the package name "LaserGame-Tests" and click "OK".

The newly created package is automatically selected, and a class definition template displayed in the code-pane at the bottom of the System Browser.

Edit the class definition template as follows, then right-click and choose "Accept it". This causes the class definition to be compiled, creating a subclass of **TestCase** named **BlankCellTest** in the running image.

```
TestCase subclass: #BlankCellTest
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LaserGame-Tests'
```

Pay attention to the class definition for your new **BlankCellTest**. It must be a subclass of the class **TestCase**. This is easy to overlook. If you did it wrong already, no problem, just go back and correct it now and then save (accept) the code again.

Within a class, methods are organised into "protocols" that are shown in the third browser pane from the left of Figure 4.1. We will group our "tests" inside a protocol of the same name.

Ensure that class **BlankCellTest** is selected then right-click the protocol list, choose "Add protocol…" from the menu. Type "tests" and click "OK".

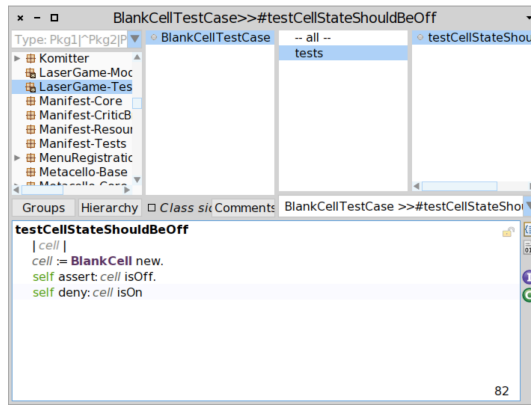

Figure 4.2: Fully defined test method.

Be sure that your browser is showing "instance" methods when you add this new method protocol and further methods - because the methods we define for now will correspond to messages sent to instances and not their classes. This is done by observing the "Class side" check-box (beneath the second browser pane) is unselected, as shown in Figure 4.1.

BTC-2014-05-24 The selected class BlankCellTestCase in Figure 4.1 doesn't match BlankCellTest used in the narrative.

Select the protocol named **tests**.

A method template is displayed in the "code pane" at the bottom of the browser. Our first test will be a simple check to see if the cell is "off" or "on" by default. We expect it to be "off" and should test for that. Just to start simple, we'll create a method with an empty body.

Clear the method template and type just the method name **testCellStateShouldBeOff**, then "Accept it".

The browser should now look like Figure 4.1. Now if no protocol was selected when you created the method it would be placed in a special protocol named 'as yet unclassified'. If that occurs, just drag the method from the method list onto the **tests** protocol.

Update method **testCellStateShouldBeOff** as follows, which results in Figure 4.2.)

```
BlankCellTest>>testCellStateShouldBeOff
   | cell |
   cell := BlankCell new.
   self assert: cell isOff.
   self deny: cell isOn
```

To efficiently select a piece of text just, place the cursor before the beginning or after the end of the text then click again, and the complete text is selected.

Before we run this test we need simple dummy definitions of methods **isOn** and **isOff** - so the methods exist before they are used by the test. (However later you will see how you don't necessarily need to define methods before they are used, they can be created "on the fly".) For now, we do the following the steps:

Select the class **BlankCell** in the LaserGame-Model package.

Add a protocol named 'testing' to class **BlankCell**, and ensure this protocol is selected.

Replace the method template with the following code for **isOff**, bring up the context menu and "Accept it" (or use shortcut command-S).

```
BlankCellTest>>isOff
   "dummy definition"
   ^ false.
```

Observe that while you are typing, the top right corner of the code pane is tinged with orange, indicating that the edit of the method has not been compiled. Once you compile the method with "Accept it", the orange tinge goes away.

Each time you modify the code of an existing method and you "Accept It", the named method will be replaced by the new code.

An efficient way of creating methods with similar code is to just change the method name in the code pane, and "Accept it".

Proceed with defining the following similar method **isOn**.

```
BlankCell>>isOn
   "dummy definition"
   ^ false.
```

4.2 Run Our First Unit Test

Even though we know the **isOn** and **isOff** methods are wrong (since we hard-coded both to answer **false**) we should run the **testCellStateShouldBeOff** test. Enough code has been written already that we want to be sure that **BlankCell** objects are created correctly and the test produces the "wrong" result we are expecting. Lets do that now.

BTC-2015-05-21 Rather than give two ways to do this, we mention just one we a demonstrating. I moved the mention of the second to later. Also, note I'm just using the "dothis" Pharo icons for code changes, and not here.

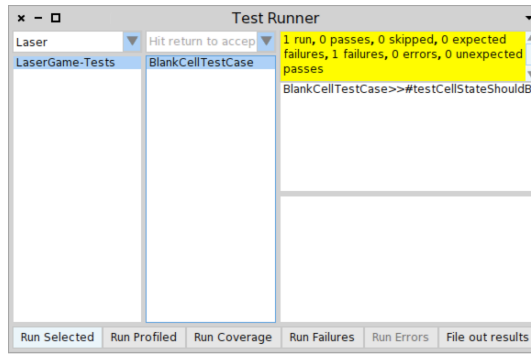

Figure 4.3: Running a test that does not pass.

- Open the Test Runner from the world Menu.
- Filter the listed packages by typing 'Laser' in the top left input field as shown in Figure 4.3.
- Select the package LaserGame-Tests.
- Select the class **BlankCellTest**.
- Press the button 'Run Selected', and you should get the situation shown in Figure 4.3.

If you do not see *LaserGame-Tests* in the list pane of the Test Runner, you probably made a mistake creating the test case class. The class **BlankCellTest** must be a subclass of **TestCase**. You may want to go back and check your work.

Getting to the Error

In the yellow pane Test Runner shows "1 failure". Beneath that the failing test methods are listed. We can open a debugger to investigate where the problem comes as follows:

- In Test Runner, click once on the failed test. A Debugger Preview window opens displaying the call stack as shown in Figure 4.4.

- Click the *Debug* button and select the test method named **testCellStateShouldBeOff** from the call-stack to see where it actually failed. You should see something like Figure 4.5.

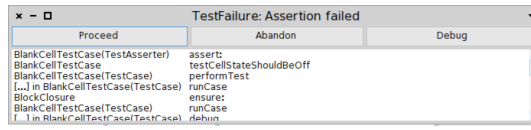

Figure 4.4: The Debugger preview.

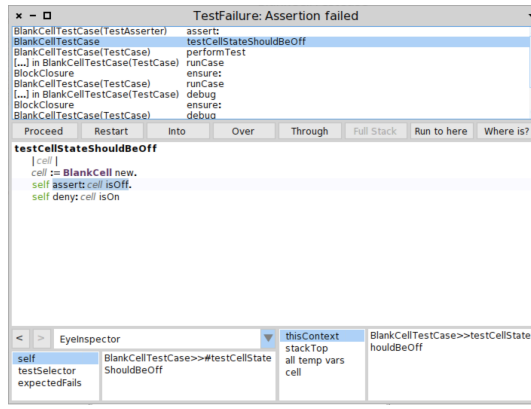

Figure 4.5: Navigating the stack of the debugger.

BTC-2014-05-24 The activeSegments instance variable in Figure 4.5 has not been created yet. Can probably just edit the bitmap to fix.

When we navigate the stack (by clicking on different lines in the top pane), the variables listed down below change. We can inspect objects held by these variables by choosing *Inspect it* from their context menu. You can also select any part of the code to evaluate and inspect. Figure 4.6 shows the Inspector that appeared after selecting the text "cell" and from its context menu, choosing to *Inspect it*. Later we will show that we can also change the method definition and proceed on the fly.

That looks great. Okay. Let's go back and make our model pass the tests. Close the debugger and inspector we opened. You can leave the Test Runner open for later. Go back to the class browser.

4.3 An alternative way to run tests

In Pharo there are two ways to execute tests. You've seen one already, using Test Runner. The other is from the System Browser. Next to each method and class is a small circle icon that indicates the success/failure status of the

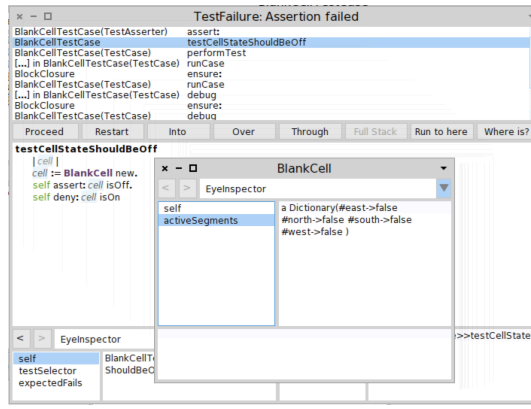

Figure 4.6: You can inspect arguments or receiver. Here the cell is ok.

last test run. Clicking on the icon next to a method runs just that test, while clicking on one next to a class runs all the tests for that class. For now, we will continue to use Test Runner.

4.4 Saving a Milestone on SmalltalkHub

Saving packages is a nice way to be able to reload your work in another image and to go back in the past of your project. We show you now how to proceed. We suggest you to create an account on SmalltalkHub (http://www.smalltalkhub.com) then create a project to save your version of this tutorial.

Assuming you've signed up and created a "LaserGame" project similar to that shown in Figure 4.7, we can illustrate the process of saving to that project repository.

- Open the Monticello Browser from the World menu,
- On the left, select the LaserGame-Model package.
- Press the '+Repository' button to add the location of your SmalltalkHub project.
- Choose an HTTP repository.
- Paste your "Monticello registration" information from SmalltalkHub (similar to that highlighted in Figure 4.7)
- Click the 'Open' button to check the repository opens without error, though it is likely empty.

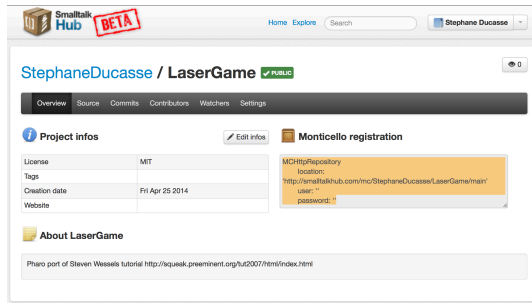

Figure 4.7: Defining a project on SmalltalkHub.

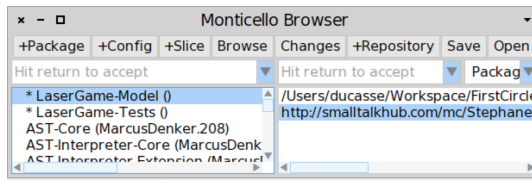

Figure 4.8: Defining a project on SmalltalkHub.

- Repeat the same steps for the 'LaserGame-Tests' package.

We added the following repository to both our packages, but yours will differ slightly.

```
MCHttpRepository
    location: 'http://smalltalkhub.com/mc/StephaneDucasse/LaserGame/main'
    user: 'mylogin'
    password: 'mysecret'
```

You should get a situation similar to the one depicted by Figure 4.8. Note that the star appearing in front of your packages means that there are changes in the system that have not been saved in the code repository. Dirty packages are sorted to the top of the list.

- Select one of the dirty packages and press the 'Save' button. Fill out the form with some useful information (e.g. 'first version with dummy testing methods'), after which you will observe the start is removed from the package, which is now resorted down with the rest of the clean packages. You should get the situation in the Monticello browser as depicted by Figure 4.9.

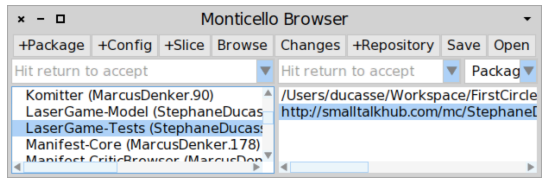

Figure 4.9: Monticello Browser with saved packages.

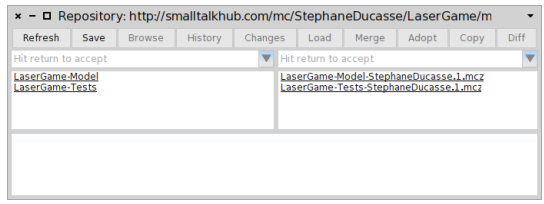

Figure 4.10: Monticello packages saved in the SmalltalHub repository.

- Scroll back up to the other dirty package and save that also. You can reuse the previous log message by clicking 'Old log messages'.

- Now *Refresh* your SmalltalkHub repository (or open it again) and you should see something like Figure 4.10.

We like to save code with green tests. This way we can load the code and continue from a stable milestone. We suggest you to do the same. We will indicate possible saving points in the course of this tutorial.

Now we are ready to refine our methods to make our tests passed.

Chapter 5

Getting our first test to pass

Remember our first test **testCellStateShouldBeOff** which was defined as:

```
BlankCellTest>>testCellStateShouldBeOff
    | cell |
    cell := BlankCell new.
    self assert: cell isOff.
    self deny: cell isOn
```

Of course we know this will fail before we even run it because we haven't fully defined either the **isOff** or **isOn** instance methods. Following on from the LED clock analogy presented earlier, to know if a cell is "on" we need to examine its four internal line segments. If any of the four internal segments are lit-up then the cell can be thought of as *on*.

However, before we can finish our **isOn** or **isOff** methods we need to define the line segments on our class. We need an instance variable to hold the active segments, using a dictionary to store pair directions and their associated values.

In the *LaserGame-Model* package, select the **BlankCell** class. Edit the class definition to add the instance variable **activeSegments** and then 'Accept it'.

```
Object subclass: #BlankCell
    instanceVariableNames: 'activeSegments'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

This modifies the class definition such that, as well as newly created **BlankCell** objects having that instance variable, any existing **BlankCell** objects within Pharo's live environment will also have the instance variable added to them.

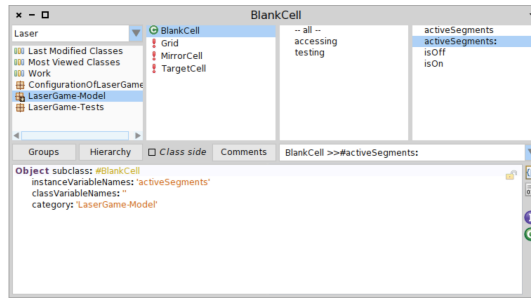

Figure 5.1: Accessors are now created and sorted in the 'accessing' protocol.

BTC-2014-05-30 The follow text is just a distraction here. There should be opportunity to demonstrate this soon. Fit it in somewhere later... "Note that we do not have to define an instance variable this way, we could simply type a method using it and compile the method, at that time the system will prompt us to know how to declare the variable (either as an instance variable or a local variable)."

We will then use Pharo's built-in refactoring tools to create accessors for this new instance variable.

Select the **BlankCell** class. From its context menu select Refactoring > Inst Var Refactoring > Accessors.

From the instance variable list, select **activeSegments**.

Review the list of proposed changes, then 'Accept' them.

This will automatically create the following methods categorized in the 'accessing' protocol as shown in Figure 5.1.

BTC-2014-05-30 The (C) icon next to BlankCell in Figure 5.1 indicates its been commented already. Was that done yet?

```
BlankCell>>activeSegments
  ^ activeSegments
```

```
BlankCell>>activeSegments: anObject
  activeSegments := anObject
```

Now since we will store a Dictionary object inside the **activeSegments** variable we will rename its argument of its setter from anObject to aDictionary. Note this is just for convenience as a form of documentation. It does not affect compilation or operation of code.

```
BlankCell>>activeSegments: aDictionary
  activeSegments := aDictionary
```

Creating accessors is not mandatory, especially for private state, so that instance variables can be only accessed directly from within the methods of the class. Different schools propose different practices with different pros and cons. We created here to provide uniformity during the complete book.

5.1 Initializing Instances

We need to initialize the new **activeSegments** variable. Let us proceed.

Add a protocol 'initialize' to the **BlankCell** class.

Define the following new instance method named **initializeActiveSegments**. The default condition for these should all be false therefore it should look like the following one:

```
BlankCell>>initializeActiveSegments

    self activeSegments: Dictionary new.
    self activeSegments at: #north put: false.
    self activeSegments at: #south put: false.
    self activeSegments at: #west put: false.
    self activeSegments at: #east put: false.
```

The method **initializeActiveSegments** needs to be activated appropriately. There are several ways to do so: (1) specialize the default **initialize** method or (2) use a lazy initialization. Let's implement and explain the first approach.

Define the method **initialize** as follows to send the message **initializeSegments** we previously defined.

```
BlankCell>>initialize

    super initialize.
    self initializeActiveSegments
```

In Pharo, when we create a new instance by sending the message **new** to a class, the newly created instance receives the message **initialize**. Therefore we get a chance to customize the default initialization.

The first thing we do in our **initialize** method is to call **super initialize**. This is a good practice to remember. Since we are implementing an **initialize** method on our new class it is conceivable that we are overriding an important **initialize** operation somewhere in our superclass hierarchy. By calling **super initialize** we give the superclasses in our hierarchy the opportunity to complete their initialization steps first.

Alternate Approach: Lazy Initialization.

Another approach is to use lazy-initialization, which initialises the variable the first time it is used. We first need check whether the variable has already been initialized (uninitialized instance variables points to **nil**) so that we don't do this twice. Usually this is done in one location in the getter method, to be called by other methods of the class instead of them accessing the instance variable directly.

The getter method would be redefined as follows:

```
BlankCell>>activeSegments
  ^ activeSegments ifNil: [ self initializeActiveSegments ]
```

Lazy initialization is useful in a live programming environment where new instance variables are added to existing objects. The **initialize** method of those objects would already been run once, and running it again would be awkward. Also, when systematically initializing an instance at creation takes too much time, lazy initialization delays initialization until the the moment where it is really necessary. This effectively trades creation time with a little extra cost (the **ifNil:** check) for each access to the variable. The effectiveness of either approach is influence by application, which would be determined by performance profiling.

5.2 Getting our Test Green

We redefine the method **isOn** as follows:

```
BlankCell>>isOn

  ^ self activeSegments values anySatisfy: [ :each | each = true ]
```

Here **activeSegment** returns the dictionary whose **values" are booleans indicating which segments are on,** anySatisfy **returns true as soon as one element is true. Now since** each" will hold booleans, you could equivalently write:

```
BlankCell>>isOn

  ^ self activeSegments values anySatisfy: [ :each | each ]
```

Once the method **isOn** is defined, **isOff** naturally follows as:

```
BlankCell>>isOff

  ^ self isOn not
```

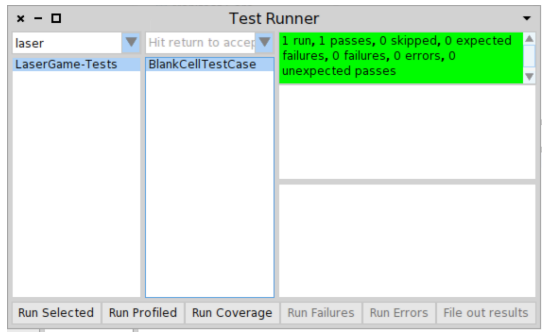

Figure 5.2: Verifying that our definitions are correct by checking our first test.

Now if you rerun the test, it passes as shown by Figure 5.2.

Adding a class comment

It is not a good practice to leave our classes undocumented. The Browser alerts this condition as shown by the exclamation mark against a class.

- Ensure the **BLankCell** class is selected.
- Press the *Comment* button.
- Add a comment like the following one for example:

```
An empty cell. A beam entering the west will exit east and the same in reverse. A
    beam entering the south will exit north and same in reverse.
```

- Accept the contents. The red exclamation mark should be gone.

5.3 Save your packages

Now that our tests are green we suggest to save a new configuration of our project following the process explained in the following chapter.

BTC-2014-05-30 Might be better to leave Versioneer to later on, when they've got something graphical to share.

Chapter 6

Using Versionner

When you save several packages you would like that when you reload them you get the correct versions of each packages. In addition, you want to make sure that the dependencies between packages are well defined. For example, it does not make sense to load LaserGame-Tests without LaserGame-Model. We say that LaserGame-Tests depends on LaserGame-Model. Pharo uses Metacello (explained in depth in Deep into Pharo http://www.deepintopharo.org) to manage dependencies within and between projects. The key concept with Metacello is the notion of configuration of a project. Here we will show how to use Versionner a new tool to define a configuration for the laser game. Versionner allows you to save a new development structure (package structure and dependencies) and from there to release and commit specific versions of your packages.

Let us see how we do that for our project.

First Steps: Adding and Naming a New Project

- Open Versionner using the Tools Menu available in the World menu.
- Add a New Project by pressing the New Project button. Name the new project 'LaserGame'. You should get a window similar to one shown by Figure 6.1.

Adding Packages

Now we will list the packages that should be managed by the configuration.

- On the left list, select the development item

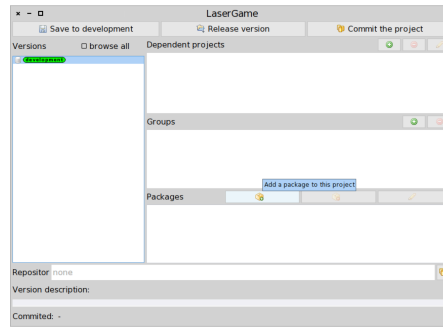

Figure 6.1: An empty configuration.

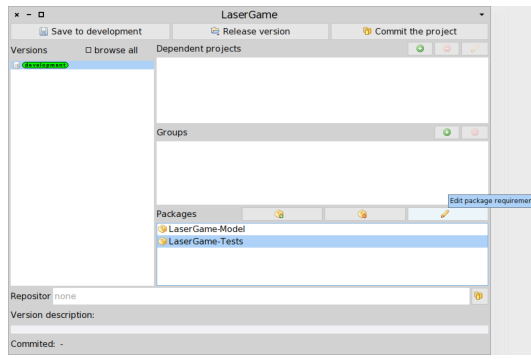

Figure 6.2: Ready to add required package.

- In package pane press add package button
- Once prompted, pick up the package LaserGame-Model
- Add the package LaserGame-Tests

You should get two packages in the package list.

Declaring Package Dependencies

Now we are ready to declare package dependencies: we want to express that Tests depends on Model.

- Select laserGame-Tests package and edit the requirement (pen icon) (See Figure 6.2).

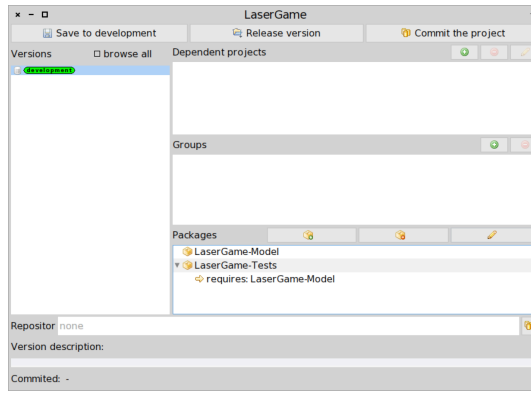

Figure 6.3: With Dependent Package.

- Press the '+' button and select the LaserGame-Model (this means that LaserGame-Model is required to load LaserGame-Tests), you should now get a system similar to the one shown in Figure 6.3.

Save to Development

The button 'Save to Development' is meant to save each time you are changing the structure of your project i.e., if you add a package or add or change a dependency. Note that loading a development version means that the system will load the latest version of a package found in the project repository. This is not exactly what we want for now but we should save to development to save the structure of our project first. So we do it.

- Press the button 'Save to development'. Pay attention the configuration is not committed to your remote server - smalltalkHub here yet. The configuration is created inside Pharo.

Adding a Repository and Committing

Now we should be able to save a development configuration. We just need to add the repository of our project (as shown in Figure 6.4).

- Add a repository by pressing the little cylinder icon and select the one of your project

Now we can commit (i.e., save the configuration to a remote server such as your smalltalkHub project).

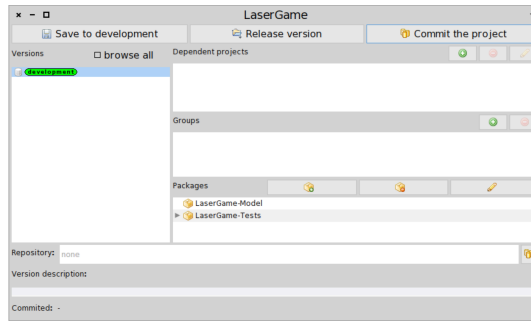

Figure 6.4: With our repository.

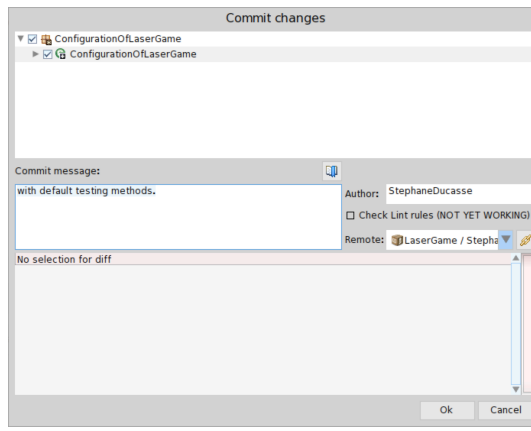

Figure 6.5: With our repository.

- Press the button 'Commit the project'. It will bring a new window showing the changes your did. It should look similar to Figure 6.5. Press the 'Ok' Button and your configuration will be saved (currently only the development one). Now our the server we saved a first configuration defining the packages and dependencies of our project.

Releasing a version

Now we are ready to release commit versions of our projects.

- Press the button 'Release version'. It will update the configuration with a new milestone representing the fact that this milestone is composed

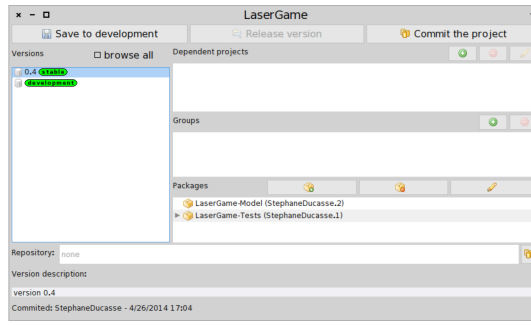

Figure 6.6: The project now has a stable version and a development one.

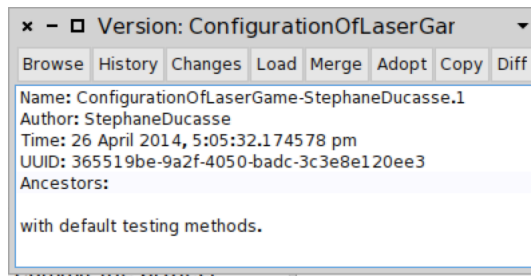

Figure 6.7: A Configuration Commit Log.

of the two specific version of our two packages. Pay attention once again the configuration is defined in Pharo but not save in the repository. Therefore we should commit it.

- Commit again the configuration by pressing the button 'Commit the project'. You should obtain a configuration state as shown in Figure 6.6. You should see that the package list now also shows you the exact version of the packages.

Each time you commit a configuration, the system produces a log of the commit for you as shown in Figure 6.7.

Ready to Continue

Now each time you want to save your work you can either select the development and commit or release a version and commit it.

Chapter 7

Coding in the debugger

Certain agile developers like to define tests first, make sure that the test fails, and define the methods within the debugger. Why? Because inside the debugger you can access *live* objects. We want to show you that we can code in the debugger. Since you save your code you will be able to continue without losing anything.

7.1 Setting up the context

So let us start.

- Redefine the methods isOn and isOff as follows

```
BlankCell>>isOn
   "dummy definition"
   ^ self halt.
```

```
BlankCell>>isOn
   "dummy definition"
   ^ self halt.
```

The self halt expression will raise an error when the methods will be executed. Here this is just to get an error and show how to fix it hot.

7.2 Coding in the debugger

- Execute the test testCellStateShouldBeOff, it will bring the preview debugger as shown in Figure 7.1.

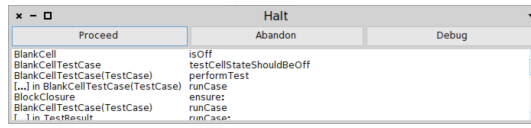

Figure 7.1: Getting the preview debugger.

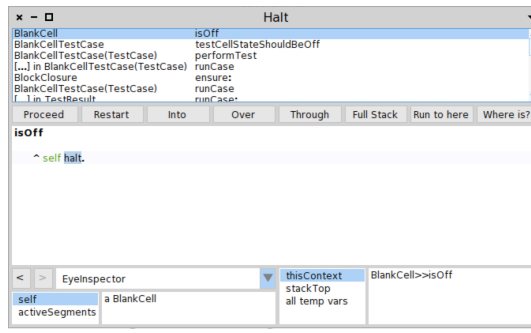

Figure 7.2: Getting the stack debugger.

- Press the 'Proceed' button, you should obtain the following debugger 7.2. You can navigate the stack and find the method raising the error, usually it is on the top of the list.

When methods are not defined, sending a message that would lead to their execution will raise an error and bring a debugger.

Now we redefine the method isOn directly in the debugger as shown in Figure 7.3. Don't forget to compile (using Command-s).

- Then we can press the button 'Proceed' and this will bring you to the next method isOff (See Figure 7.4) that we can then defined as shown in Figure 7.5

- Press proceed and you will see that the test gets green.

7.3 Conclusion

We will not repeat this process further in the tutorial but we use it daily in our development.We can only encourage you to try this great way of developing software.

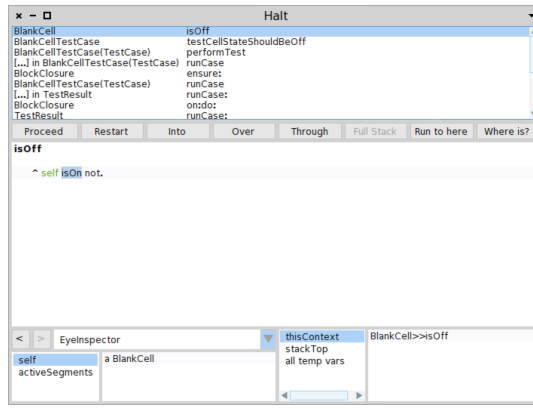

Figure 7.3: The method isOn directly in the debugger.

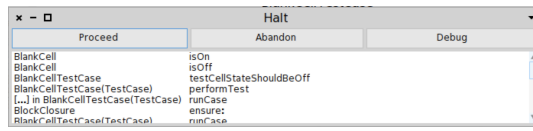

Figure 7.4: Getting to the method isOff.

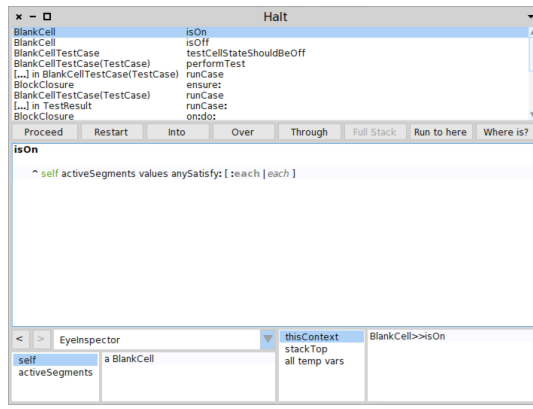

Figure 7.5: Now we can redefine the other method.

Chapter 8

Improving our Model

Now that we saved our previous version with green tests, we can add new behavior to our model.We will do it test first.

8.1 Adding a new method

We want to add a new method named isSegmentOnFor: which will tell us whether a segment for a giving direction is on. Let us define first a simple test named testIsSegmentOn defined as follow.

```
BlankCellTest>>testIsSegmentOn
   | cell |
   cell := BlankCell new.
   self deny: (cell isSegmentOnFor: #north).
```

Note that the message will be displayed with a different indicating that the messsage does not already exist in the system. Proceed and compile the method. Pharo will prompt you, so confirm that you are defining a new method.

Then execute the new test, and open the debugger, you should get a debugger as the one shown in Figure 8.1. The debugger shows you that the message isSegmentOn: not understood by the receiver.

Click on the 'Create' button, you will get prompt to know on which class you want to define the method isSegmentOn:. Pick BlankCell as shown in Figure 8.2.

You should get now a debugger on the method automatically created by the debugger as shown in Figure 8.3.

Now we can define inside the debugger the method isSegmentOnFor:.

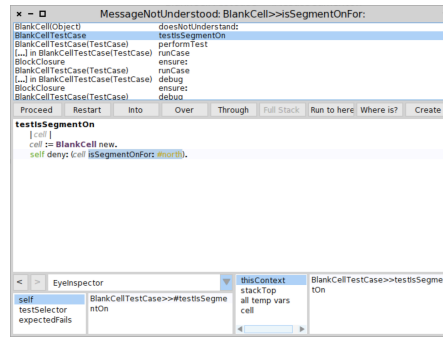

Figure 8.1: Debugger opened for a message not understood.

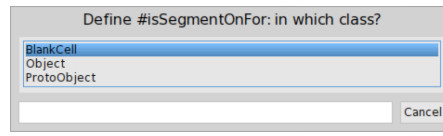

Figure 8.2: Selecting the class to create the new method.

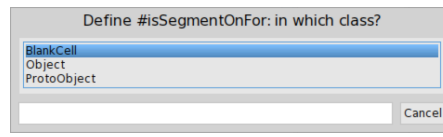

Figure 8.3: An automatically created method.

```
BlankCell>>isSegmentOnFor: aSymbol
    ^ self activeSegments at: aSymbol
```

And when we press proceed, the test is run.

Defining another test

Now we can define a test testing all the directions.

```
BlankCellTest>>testNewCellDoesNotHaveSegmentOn
    | cell |
    cell := BlankCell new.
    self deny: (cell isSegmentOnFor: #north).
    self deny: (cell isSegmentOnFor: #south).
```

```
    self deny: (cell isSegmentOnFor: #west).
    self deny: (cell isSegmentOnFor: #east).
```

Execute the tests to verify that everything is working. Your three tests should be green.It is a good occasion to save your code.

8.2 Handling Beam Exist Path

We've written methods to check the state of individual segments within our cell. Now we also need methods to answer the output direction for a given laser beam input direction. The BlankCell class needs another instance variable that will encode the path of a beam: i.e., for an input direction provides an output direction.

```
Object subclass: #BlankCell
    instanceVariableNames: 'activeSegments exitSides'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

As before, create accessors and we will define another initialization method to define the beam path in a cell. But let us write first a test method.

```
BlankCellTest>>testCellExit
    | cell |
    cell := BlankCell new.
    self assert: (cell exitFor:  #north) = #south.
    self assert: (cell exitFor:  #south) = #north.
    self assert: (cell exitFor:  #west) = #east.
    self assert: (cell exitFor:  #east) = #west.
```

Now define the method initializeExitSides to initialize the path.

```
BlankCell>>initializeExitSides

    self exitSides: Dictionary new.
    self exitSides at: #north put: #south.
    self exitSides at: #south put: #north.
    self exitSides at: #east put: #west.
    self exitSides at: #west put: #east.
```

And modify the initialize method to call it.

```
BlankCell>>initialize

    super initialize.
    self initializeActiveSegments.
    self initializeExitSides
```

Now we can define the method exitFor: which given a direction returns the exit that the laser beam will follow.

```
BlankCell>>exitFor: aSymbol

  ^ self exitSides at: aSymbol
```

Run your tests and once they are green this is a good occasion to commit your work.

8.3 Beam Entering Side

Another behavior we need, is to tell the cell that the laser beam has entered the cell. We will need to specify which side the laser beam entered by. Then our new test should check that the expected line segments in the cell are lit-up when the method completes. Just add the test in the test case class. We already know the new method laserEntersFrom: does not exist yet.

```
BlankCellTest>>testLaserEntersNorthShouldLitUpSouth
  | cell |
  cell := BlankCell new.
  cell laserEntersFrom: #north.
  self assert: ( cell isOn ).
  self assert: (cell isSegmentOnFor: #north).
  self assert: (cell isSegmentOnFor: #south).
  self deny: (cell isSegmentOnFor: #west).
  self deny: (cell isSegmentOnFor: #east).
```

Execute the test and it should failed as expected. Now define the method laserEntersFrom: in the class BlankCell.

http://squeak.preeminent.org/tut2007/html/023.html

```
BlankCell>>laserEntersFrom: aSymbolDirection

  | exit |
  self activeSegments at: aSymbolDirection put: true.
  exit := self exitSideFor: aSymbol.
  self activeSegments at: exit put: true.
```

Run your tests. They should all be green.

Note that we could provide methods to set or toggle the state of a direction, instead of exposing the fact that we manipulate a dictionary. It could be setOnSegmentFor: and setOffSegmentFor:.

```
BlankCell>>setSegmentOnFor: aSymbolDirection
  self activeSegments at: aSymbolDirection put: true
```

```
BlankCell>>setOffSegmentOffFor: aSymbolDirection
   self activeSegments at: aSymbolDirection put: false


BlankCell>>laserEntersFrom: aSymbolDirection

   self setSegmentOnFor: aSymbolDirection.
   self setSegmentOnFor: (self exitFor: aSymbol)
```

Add a new test to cover setOnSegmentFor: and setOffSegmentFor:. For example a good test is the following one.

```
BlankCellTest>>testSetSegment
   | cell |
   cell := BlankCell new.
   self deny: (cell isSegmentOnFor: #north).
   cell setSegmentOnFor: #north.
   self assert: (cell isSegmentOnFor: #north).
   cell setSegmentOffFor: #north.
   self deny: (cell isSegmentOnFor: #north).
```

Run your tests. They should all be green.

8.4 A Discussion About Directions

The design we proposed to you use symbols to represent direction. It is ok now there are still some place for improvements that you could do. Carrying around symbol in all the code is prone to errors. Imagine that you mistype west as vest then your system will not work well anymore. A way to improve is to define class methods (north, south, west, east) on the class Cell and to use them in place of symbol. We could also use class variables or a pool dictionary to hold such values (Read the model chapter of Pharo by Example to know more).

A good acid test is that we can replace the symbol #north in the method north by a number such as 0 and the system should continue to work.

Another way to improve could be to represent direction as real objects. We let you experiment with such design.

Chapter 9

Enhancing MirrorCell

A mirror cell is a cell which reflects the beam in a different direction. This cell is different from the Blank Cell because it has a mirror and the mirror can be oriented. The output direction of the entering laser beam is also different because of the mirror.

9.1 Adding a class comment

Let us start to add a class comment. We propose something like the following.

```
A mirror is a cell that reflects beam based on its orientation.
There are two main orientation: towards left and towards right.

With left, a beam coming from the south will exit left.
With right, a beam coming from the south will exit left.
```

9.2 Capturing Mirror Orientation

Then we need a way to define the orientation of the mirror. We can begin by adding the instance variable "leansLeft" to the class and creating the accessors too.

```
Object subclass: #MirrorCell
   instanceVariableNames: 'leansLeft'
   classVariableNames: ''
   category: 'LaserGame-Model'
```

Add some initialization code. Let the default condition be that the mirror leans left unless otherwise specified.

```
MirrorCell>>initialize
  super initialize.
  self leansLeft: true
```

Add testing methods for isLeft and isRight. These methods should be defined with the protocol 'testing'.

9.3 Introducing a Common Ancestor

This was all straight-forward. But what about the behavior that is similar to the Blank Cell such as isSegmentOn: or laserEntersFrom: or simply encoding the output direction of a beam? If we follow the way we defined this behavior in BlankCell. We'll need the same instance variables 'activeSegments' and 'exitSides'. The behavior of how these values are set will be different. Having to define nearly the same behavior and state in two different classes is a sign that we can do better. We want to re-use the code as much as possible. This is a good case for an abstract class that would define what the two cells have in common.

SD: we should add a diagram showing the transformation

Let's create an abstract class called Cell.

```
Object subclass: #Cell
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LaserGame-Model'
```

Pharo comes with a powerful tool to manipulate code (moving methods between classes, moving instance variables within a hierarchy...). A refactoring is a behavior preserving operation. This means that the behavior of a program should not change when applying a refactoring. This tools is called the RefactoringEngine (and it was the first fully working refactoring engine available for a programming language). The refactogin engine is available in the default class browser under the Refactoring menu item.

Inheriting from Cell

First let redefine MirrorCell as a subclass of Cell.

```
Cell subclass: #MirrorCell
   instanceVariableNames: 'leansLeft'
   classVariableNames: ''
```

```
    category: 'LaserGame-Model'
```

And BlankCell as a subclass of Cell

```
Cell subclass: #BlankCell
    instanceVariableNames: 'activeSegments exitSides'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

You can run the tests and they should pass.

Moving up Segment and Exit to Cell

Since a BlankCell and a MirrorCell have to manage which segments are on and how a beam cross them, we will move the instance variable activeSegments and exitSides to the superclass.

Let us start to pull up the instance variable using the refactoring engine. Click on the class BlankCell and bring the menu and select 'Refactoring', then 'Inst Var Refactoring' and finally 'pull up'. Select the activeSegments instance variable. Do the same for the 'exitSide' instance variables.

You should get now your classes as follows:

```
Object subclass: #Cell
    instanceVariableNames: 'activeSegments exitSides'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

```
Cell subclass: #BlankCell
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

Your tests should still be passing.

What is nice with refactorings is that they verify a lot of possible mistakes for example a subclass of Cell could have already an instance variable with the same name and this would be problem. Note that refactorings are good also for more complex situations involving multiple classes.

Moving up Methods to Cell

Now we can do the same to move up the following methods:

- The accessors activeSegments, activeSegments:, exitSides, exitSides:,
- the testing methods isSegmentOnFor:, isOn, isOff, and

- some operations such as setSegmentOffFor:, setSegmentOnFor:, laserEntersFrom: and exitFor:.

Select these methods, bring the menu, and select the 'Refactoring' submenu and from this one the 'Push up' submenu item.You will move all the methods at once to the superclass.

Your tests should still be passing.

Splitting object initialization

The initialize methods need to be integrated into the new class.

- The initializeActiveSegments method will work for any of these cells because they all have 4 segments with a default condition of false. This method clearly belongs on the superclass. So push it up to the superclass as we did before.

- The initializeExitSides does not because it defines part of what makes the BlankCell unique from the other cells.

Here's how the initialize methods turn out for Cell.

```
Cell>>initializeActiveSegments

    self activeSegments: Dictionary new.
    self activeSegments at: #north put: false.
    self activeSegments at: #south put: false.
    self activeSegments at: #west put: false.
    self activeSegments at: #east put: false.
```

We could use the method setSegmentOffFor:

```
Cell>>initializeActiveSegments

    self activeSegments: Dictionary new.
    self setSegmentOffFor: #north.
    self setSegmentOffFor: #south.
    self setSegmentOffFor: #west.
    self setSegmentOffFor: #east.
```

```
Cell>>initialize

    super initialize.
    self initializeActiveSegments.
```

Here's how the initialize methods turn out for Cell.

```
BlankCell>>initializeExitSides

    self exitSides: Dictionary new.
    self exitSides at: #north put: #south.
    self exitSides at: #south put: #north.
    self exitSides at: #east put: #west.
    self exitSides at: #west put: #east.
```

```
BlankCell>>initialize

    super initialize.
    self initializeExitSides
```

Run the tests they should pass.

Alternate Design

For the initialization we could also define the initialize method as follows:

```
Cell>>initialize

    super initialize.
    self initializeActiveSegments.
    self exitSides: Dictionary new.
    self initializeExitSides
```

And define only one method initializeExitSides in BlankCell and each future subclasses of Cell.

```
BlankCell>>initializeExitSides

    self exitSides at: #north put: #south.
    self exitSides at: #south put: #north.
    self exitSides at: #east put: #west.
    self exitSides at: #west put: #east.
```

9.4 Coming back to MirrorCell

Now the class MirrorCell can be also a subclass of Cell.

```
Cell subclass: #MirrorCell
    instanceVariableNames: 'leansLeft'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

When we left off coding the MirrorCell we realized that there were a lot of similarities with BlankCell and a few differences.

Let us write a couple of tests to specify the behavior we expect. Add a new test class named MirrorCellTest subclass of TestCase.

```
TestCase subclass: #MirrorCellTest
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LaserGame-Tests'
```

First create a simple test to check that a newly created instance is off.

```
MirrorCellTest>>testCellAtCreationShouldBeOff

   | cell |
   cell := MirrorCell new.
   self assert: cell isOff.
   self deny: cell isOn.
```

```
MirrorCellTest>>testSegmentAtCreationShouldBeOff

   | cell |
   cell := MirrorCell new.
   self deny: (cell isSegmentOnFor: #north).
   self deny: (cell isSegmentOnFor: #south).
   self deny: (cell isSegmentOnFor: #west).
   self deny: (cell isSegmentOnFor: #east).
```

Up to now all the tests should pass.

Handling Orientation

We know we will have to set the orientation for the mirror both when the cell is first instantiated as well as on-the-fly if the user decides to rotate the cell. Therefore we need methods to handle those requests, but first let us write two nice tests to make sure that we will be sure that our methods are correct.

The testCellExitSides method should be different, and should depend on orientation of the mirror. Use 2 new method names to show testing for each orientation.

```
MirrorCellTest>>testCellExitSidesForLeftMirror

   | cell |
   cell := MirrorCell new.
   cell leanLeft.
   self assert: (cell exitFor: #north) = #east.
   self assert: (cell exitFor: #south) = #west.
```

```
    self assert: (cell exitFor: #east) = #north.
    self assert: (cell exitFor: #west) = #south.
```

```
MirrorCellTest>>testCellExitSidesForRightMirror

    | cell |
    cell := MirrorCell new.
    cell leanRight.
    self assert: (cell exitFor: #north) = #west.
    self assert: (cell exitFor: #south) = #east.
    self assert: (cell exitFor: #east) = #south.
    self assert: (cell exitFor: #west) = #north.
```

Now we have to write two test methods to specify the activity of the cell when the beam pass through it.

```
MirrorCellTest>>testLaserEntersNorthShouldLitUpEast
    | cell |
    cell := MirrorCell new.
    cell leanLeft.
    cell laserEntersFrom: #north.
    self assert: ( cell isOn ).
    self assert: ( cell isSegmentOnFor: #north).
    self assert: ( cell isSegmentOnFor: #east).
    self deny: ( cell isSegmentOnFor: #south).
    self deny: ( cell isSegmentOnFor: #west).
```

```
MirrorCellTest>>testLaserEntersNorthShouldLitUpWest
    | cell |
    cell := MirrorCell new.
    cell leanRight.
    cell laserEntersFrom: #north.
    self assert: ( cell isOn ).
    self assert: ( cell isSegmentOnFor: #north).
    self assert: ( cell isSegmentOnFor: #west).
    self deny: ( cell isSegmentOnFor: #south).
    self deny: ( cell isSegmentOnFor: #east).
```

Obviously the four last test methods should fail, since we did not implement yet the correct behavior.But now we are ready to do so.

```
MirrorCell>>initializeExitSides
    self exitSides: Dictionary new.
```

```
MirrorCell>>initialize
    super initialize.
    self initializeExitSides.
    self leanLeft.
```

```
MirrorCell>>leanRight

    self leansLeft: false.
    self exitSides at: #north put: #west.
    self exitSides at: #east put: #south.
    self exitSides at: #south put: #east.
    self exitSides at: #west put: #north.
```

```
MirrorCell>>leanLeft

    self leansLeft: true.
    self exitSides at: #north put: #east.
    self exitSides at: #east put: #north.
    self exitSides at: #south put: #west.
    self exitSides at: #west put: #south.
```

Run your tests and they should pass, else fix your code and rerun your tests.Then this is a good moment to save your work.

9.5 Instance Creation Methods

Whenever we instantiate cells from the MirrorCell we need to take an extra step of orienting the mirror. As a practical matter, the MirrorCell must always have one orientation, left or right. We can streamline this behavior with a few new *class* methods on MirrorCell.

A class method is a method that will be executed when we send a message to a *class*, while an instance method is a method that will be executed when we send a message to an *instance* of this class. Here we want to write MirrorCell leanLeft therefore we should define a class method leanLeft. We are sorted the new class methods in a protocol called "instance creation" as shown in Figure 9.1. Noticed how the browser shows us that we are editing a class method: the *Class side* radio button is set.

```
MirrorCell class>>leanLeft
    ^ self new leanLeft
```

```
MirrorCell class>>leanRight
    ^ self new leanRight
```

Important note! In the browser above we just created a new *Class* method. Not an *Instance* method. Make sure you have the "class" button clicked before creating the leanLeft method. A number of students learning from this tutorial have made this mistake. If you do too, just delete the instance

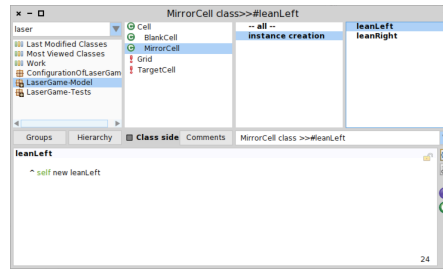

Figure 9.1: Defining a class method: the Class side button is set.

method you created by mistake, click on the "class" button and create a new class method instead.

You can for example add a new test method as the following one:

```
MirrorCellTest>>testMirrorCreationWithClassAPI

    | cell |
    cell := MirrorCell leanRight.
    self assert: (cell exitFor: #north) = #west.
    self assert: (cell exitFor: #south) = #east.
    self assert: (cell exitFor: #east) = #south.
    self assert: (cell exitFor: #west) = #north.
```

9.6 About MirrorCell Design

Instead of having to test all the time state using isLeft and isRight, a better design is to define two subclasses to MirrorCell one for each of the case and define specific methods in each. What is important to see is that each time we send a message, the correct method based on the receiver is automatically selected and executed. This mechanism acts as a systematic conditionals, therefore each time we explicitly test we do not take advantage of this build-in mechanism that the center of object-oriented programming.

Chapter 10

Enhancing TargetCell

A target cell is the last one of our cells. It is unique because it does not have exit. Once the laser beam enters a target cell it does not leave and propagate anymore. A design choice we made here is to set the exit values to be nil. But let us start by writing some tests to clarify what we mean.

10.1 Adding a class comment

Let us start to add a class comment. We propose something like the following.

```
A target cell is a cell does not let the beam leave once entered. Once a target is
    reached the game is finished.
```

10.2 Adding Some Tests

Let us follow our good practices and add some tests.

We add a new test class named TargetCellTest

```
TestCase subclass: #TargetCellTest
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'LaserGame-Tests'
```

At creation time the cell should be off.

```
TargetCellTest>>testCellStateShouldBeOff
  | cell |
```

```
cell := TargetCell new.
self assert: cell isOff.
self deny: cell isOn
```

All its segments should be off too.

```
TargetCellTest>>testSegmentAtCreationShouldBeOff

| cell |
cell := TargetCell new.
self deny: (cell isSegmentOnFor: #north).
self deny: (cell isSegmentOnFor: #south).
self deny: (cell isSegmentOnFor: #west).
self deny: (cell isSegmentOnFor: #east).
```

The testCellLaserActivitytestLaserEntersNorthShouldNotLitUpSouth method should only find one segment lit-up when the laser beam enters.

```
TargetCellTest>>testLaserEntersNorthShouldNotLitUpSouth

| cell |
cell := TargetCell new.
cell laserEntersFrom: #north.
self assert: ( cell isOn ).
self assert: (cell isSegmentOnFor: #north).
self deny: (cell isSegmentOnFor: #south).
self deny: (cell isSegmentOnFor: #west).
self deny: (cell isSegmentOnFor: #east).
```

Now we express that all the exit sides of a target will be nil.

```
TargetCellTest>>testCellExitsAreNil

| cell |
cell := TargetCell new.
#(north #east #south #west)
    do: [ :inputSide | self assert: (cell exitFor: inputSide) isNil ]
```

10.3 Defining the corresponding behavior

Now we are ready to define all the behavior we wrote tests for.

TargetCell is a Cell

The first thing we will do is to make TargetCell inherits from Cell. Do it and you should get an hierarchy as shown by Figure 10.1.

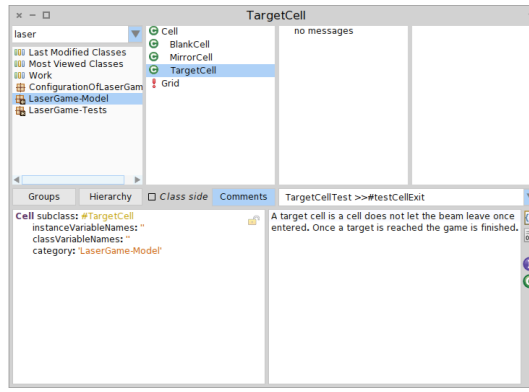

Figure 10.1: TargetCell should also inherit from Cell.

```
Cell subclass: #MTargetCell
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LaserGame-Model'
```

Start to run the tests. Normally two not related to exit faces should already pass.

Handling Exits

Now we should handle exit side. We redefine the initializeExitSides as follows:

```
TargetCell>>initializeExitSides

   self exitSides: Dictionary new.
   self exitSides at: #north put: nil.
   self exitSides at: #east put: nil.
   self exitSides at: #south put: nil.
   self exitSides at: #west put: nil.
```

And again we should define the method initialize as follows.

```

   super initialize.
   self initializeExitSides
```

Run the tests and they should all be green. This is clearly a good occasion to save your code.

10.4 About Design Points

About our initialize methods

It looks like we already defined this method several times and exactly the same. This is clearly a sign that we can do better. In the literature this is called a code smell (with the idea that bad code smells rotten parfums). Propose a solution to this problem without readin ours.

Our solution is the following one: we remove `self exitSides: Dictionary new.` from the `initializeExitSides` and move it to the superclass `initialize` method.

```
Cell>>initialize

    super initialize.
    self initializeActiveSegments.
    self exitSides: Dictionary new.
    self initializeExitSides.

Cell>>initializeExitSides
    "does nothing"
```

```
BlankCell>>initializeExitSides

    self exitSides at: #north put: #south.
    self exitSides at: #south put: #north.
    self exitSides at: #east put: #west.
    self exitSides at: #west put: #east
```

And we can now remove the initialize methods of `BlankCell`.

```
MirrorCell>>initialize
    super initialize.
    self leanLeft
```

And we can remove the method `initializeExitSides` from MirrorCell since it is a copy of its superclass.

```
TargetCell>>initializeExitSides

    self exitSides at: #north put: nil.
    self exitSides at: #east put: nil.
    self exitSides at: #south put: nil.
    self exitSides at: #west put: nil.
```

And we can remove the initialize method of `TargetCell`.

Now we can run again our tests. And we can safely save a new version.

About TargetCell Design

Using nil to encode that there is no value for the exitSide is not an optimal way. Why? because it may force the client code to always check if it gets true, false or nil. Good object-oriented design is often to have an object accepting the same messages but doing nothing. It is called the NullObject design pattern. What is nice is that it eliminates all the ifNil: tests and follow the "tell don't ask" way of really thinking object-oriented programming.

We will let the use of nil right now but remember it in the future and a good exercise is to propose a new version without this encoding.

Chapter 11

Grid

From what we've seen so far, the Grid is responsible for holding the cells in a matrix. It must have a provision to add specific cells to specific locations, and it should have a mechanism to address cells at specific locations. The Grid will activate the laser beam.

11.1 Adding a class comment

So let us first add a class comment, for example as follows

```
A grid is holding a matrix of cells. It is responsible for emitting the laser beam.
```

11.2 Adding some instance variables

We will add two instance variables, cells and laserIsActive to the Grid class. Create the corresponding accessors.

```
Object subclass: #Grid
    instanceVariableNames: 'cells laserIsActive'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

The instance variable cells will contain a dictionary where the keys are points of location within the grid and the values are the cells. The instance variable laserIsActive is a boolean indicating state of the laser beam.

We will also add two other instance variables that represent the shape of the matrix. Add the instance variables numberOfColumns and numberOfRows and their respective accessors.

```
Object subclass: #Grid
   instanceVariableNames: 'cells laserIsActive'
   classVariableNames: ''
   category: 'LaserGame−Model'
```

11.3 Encapsulating cell accesses

We want to address cells within the grid. Rather than expose how cells are actually stored in our grid, we should provide methods that provide access to and setting of specific cells. Let us define two methods at: aPoint and at: aPoint put: aCell. A point in Pharo is expressed as x @ y and we can access the x of a point just sending the message x to this point.

```
Grid>>at: aPoint
  ^ self cells at: aPoint
```

```
Grid>>at: aPoint put: aCell
  self cells at: aPoint put: aCell
```

Note that using a dictionary to store cells to support their accesses based on their location is a quick and dirty way of doing it. The proper way would be to use an array or a matrix and to access directly the cells doing some simple calculation (something in the spirit of x + y * line number). Now using the at: and at:put: messages is nice because we will be able to change this implementation point at anytime in the future without impacting the rest of the program.

11.4 Initializing a Grid

Now we are ready to initialize a grid. We should set the value of the laserIsActive variable to false and initialize the cells, taking into account the number of rows and columns. So let us start to define the initialize method and see how it goes.

```
Grid>>initialize

   super initialize.
   self laserIsActive: false.
   self initializeCells
```

So now we should define initializeCells. We should iterate over all the positions and create a corresponding cell.

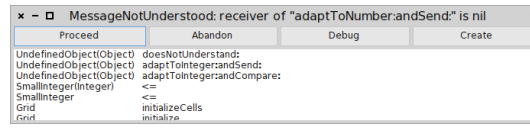

Figure 11.1: Running the test brings the preview debugger.

```
Grid>>initializeCells

    self cells: Dictionary new.
    1 to: self numberOfColumns do: [ :col |
            1 to:  numberOfRows do: [ :row |
                self at: col@row put: BlankCell new ] ]
```

But wait we indeed defined the accessors for numberOfColumns and numberOfRows but we did not initialize them. So two solutions either we change the initialize method to add the following

```
Grid>>initialize

    super initialize.
    self laserIsActive: false.
    self numberOfColumns: 10.
    self numberOfRows: 10.
    self initializeCells
```

or we can use lazy initialization. Since we never used that in practice, let us redefine the getter to implement a lazy initialisation. Pay attention that it can only work if you consistently use getters and never directly access the instance variables else you could get the uninitialized value (nil)

```
    ^ numberOfRows ifNil: [ numberOfRows := 5 ]
```

```
Grid>>numberOfColumns
    ^ numberOfColumns ifNil: [ numberOfColumns := 5 ]
```

Now your test should be green. Well not quite as shown by Figure 12.1. Press the 'Debug' button to try to find what got wrong using the debugger.Navigate the stack to understand how you got the problem. Often the problem happens several steps further down the stack. You should find that the initializeCells method is where the problem is as shown Figure 11.2.

Now try to address the problem before looking at the solution.

We forgot to send numberOfRows to self and the algorithm tried to use the instance variable directly. We knew the variable could have been nil

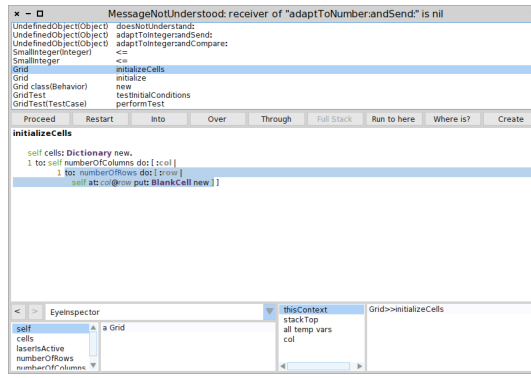

Figure 11.2: Navigating and finding the problem using the debugger.

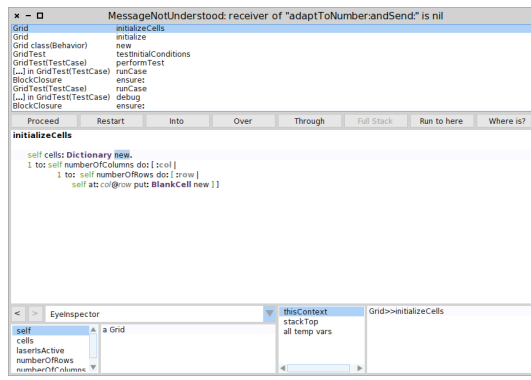

Figure 11.3: TargetCell should also inherit from Cell.

and it was.Okay, that's easy enough of a mistake to fix. Here's the corrected initializeCells method. You can fix it right in the debugger. Just press proceed when after you have saved the change and then re-run the unit tests.

Redefine the method initialize to correctly use the getter for numberOfRows as shown in Figure 11.3.

```
Grid>>initializeCells

    self cells: Dictionary new.
    1 to: self numberOfColumns do: [ :col |
            1 to:  self numberOfRows do: [ :row |
                self at: col@row put: BlankCell new ] ]
```

Now this is the good moment to save our progress.

11.5 Discussions: Definition Order

The order in which we define methods is not important from the standpoint of the program execution. Now in some cases we prefer to be able to test as soon as the method is defined if it is working. In such case we start defining the elementary methods or the methods used by others. Then slowly we build on methods already working to define more complex ones. This strategy is bottom up and can work well when we know what are the low-level methods and object representation. In fact we can define tests covering each of these methods and validate our progress systematically.

Another strategy is to define more complex methods using methods that do not have been yet defined. When executing a test covering the more complex method we will get a debugger with the exact context and it will help us to define the missing methods.

Both strategies have pros and cons. Just be aware of the possibilities and pick the one that fits the best your current process.

11.6 Validating Behavior with Some More Tests

Let's enrich the Grid unit test. We'll specify a column and row size and retest the basics.

```
GridTest>>testNonDefaultGridSizeInitialConditions
    | grid |
    grid := Grid new.
    grid numberOfColumns: 4.
    grid numberOfRows: 4.
    self deny: grid laserIsActive.
    self assert: (grid at: 1@1) class = BlankCell.
    self assert: (grid at: 2@3) class = BlankCell.
    self assert: (grid at: 2@3) isOff.
```

So far so good. But well let us think about a possible problem we can encounter. When the Grid class receives the message new, its initialize method is invoked and the cells are added to the dictionary but.Do you see the problem? But the cells are initialized based on the row and column numbers defined at the time the method is invoked.So what will happen if we change the number of rows after the creation of the grid instance? It will break. Why? Simply because the number of rows will be larger and we will think that we can access cells in this area but in realitythe dictionary will only cover the values with defined in the lazy accessors.

Let us write a test to illustrate our problem: Here we create a new grid instance and after set its new size.

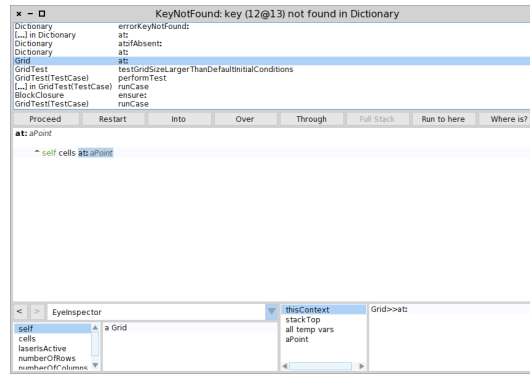

Figure 11.4: Accessing a key that does not exist.

```
GridTest>>testGridSizeLargerThanDefaultInitialConditions
    | grid |
    grid := Grid new.
    grid numberOfColumns: 15.
    grid numberOfRows: 15.
    self deny: grid laserIsActive.
    self assert: (grid at: 1@1) class = BlankCell.
    self assert: (grid at: 12@13) class = BlankCell.
    self assert: (grid at: 12@13) isOff.
```

Run the tests and it will failed as we expected! Exactly a key not found exception is raised. It is raised when we try to access a key that does not exist in a dictionary (as shown in Figure 11.4).

Improving the instance creation of Grid

We can define the class method as follow (pay attention to select the Class side radio button since this is a class method).

```
Grid class>>numberOfColumns: colInteger numberOfRows: rowInteger

    | instance |
    instance := self new.
    instance numberOfColumns: colInteger.
    instance numberOfRows: rowInteger.
    ^ instance
```

What this method does is simple: we create a new instance, assign it to a temporary variable, set the value of the rows and columns and return the

newly created instance. Do you think it will work? It does not. Why because the initialize method is invoked by the new message in Pharo.The solution is to modify the class method to use the message basicNew instead of new.

```
Grid class>>numberOfColumns: colInteger numberOfRows: rowInteger

    | instance |
    instance := self basicNew.
    instance numberOfColumns: colInteger.
    instance numberOfRows: rowInteger.
    instance initialize.
    ^ instance
```

What is the difference between new and basicNew?

- new automatically sends the message initialize to the instance it creates and it can be overloaded with other behavior.
- basicNew just create the instance and return it. It is a primitive creation message and it should never be redefined else you could not build such solution. Note that new in fact uses basicNew.

Now we can also rewrite our second test to use the new class creation method.

```
testGridSizeLargerThanDefaultInitialConditions
    | grid |
    grid := Grid numberOfColumns: 15 numberOfRows: 15.
    self deny: grid laserIsActive.
    self assert: (grid at: 1@1) class = BlankCell.
    self assert: (grid at: 12@13) class = BlankCell.
    self assert: (grid at: 12@13) isOff.
```

Run your tests and they should be green this time.

Understanding yourself

The previous (suboptimal) method numberOfColumns:numberOfRows: (we put back its definition to help you following) is strictly equivalent to the more compact one following it using the message yourself.

```
Grid class>>numberOfColumns: colInteger numberOfRows: rowInteger

    | instance |
    instance := self basicNew.
    instance numberOfColumns: colInteger.
    instance numberOfRows: rowInteger.
    instance initialize.
```

```
^ instance
```

The following method is strictly the same but its uses cascade (;) to send multiple messages to the same object (here the newly created instance returned by self new.

```
Grid class>>numberOfColumns: colInteger numberOfRows: rowInteger

    ^ self basicNew
        numberOfColumns: colInteger;
        numberOfRows: rowInteger;
        initialize;
        yourself.
```

How does it work?

yourself is a message returning the receiver of the message. Here the result of the self new expression. We use yourself to write more compact code. Note that the use of yourself ensures the method will really return the receiver of the message (here the new instance of grid) and not the result of the last method as this would be the case with the following potentially buggy method. Why? What if initialize returns 42 instead of the self default value?Then potentiallySuspectNumberOfColumns:numberOfRows: will return 42 and not the newly created grid instance.

```
Grid class>>potentiallySuspectNumberOfColumns: colInteger numberOfRows:
        rowInteger

    ^ self new
        numberOfColumns: colInteger;
        numberOfRows: rowInteger;
        initialize
```

11.7 Building a better test context

With the basics on the cells working, and a minimal test for Grid, we can start to write a much deeper unit test for the Grid class, using broader parts of the overall design. Remember the original diagram used to introduce the game idea? Let's use that for our unit test. Here's the empty diagram and the diagram with the laser firing (see Figure 11.5).

Define the following method and add a new protocol *private* or *grids*.

```
GridTest>>generateDemoGrid

    | grid |
    grid := Grid numberOfColumns: 5 numberOfRows: 5.
```

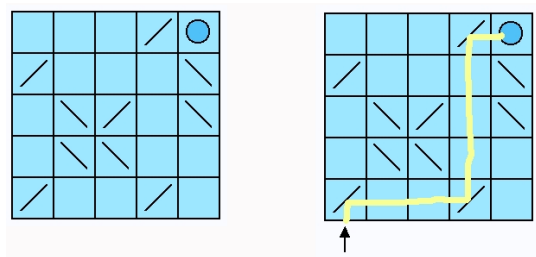

Figure 11.5: A nice context for future tests.

```
    grid at: 5@1 put: TargetCell new.

    grid at: 4@1 put: MirrorCell leanRight.
    grid at: 1@2 put: MirrorCell leanRight.
    grid at: 3@3 put: MirrorCell leanRight.
    grid at: 1@5 put: MirrorCell leanRight.
    grid at: 4@5 put: MirrorCell leanRight.

    grid at: 5@2 put: MirrorCell leanLeft.
    grid at: 2@3 put: MirrorCell leanLeft.
    grid at: 5@3 put: MirrorCell leanLeft.
    grid at: 2@4 put: MirrorCell leanLeft.
    grid at: 3@4 put: MirrorCell leanLeft.
    ^ grid
```

We can now add a new test method. A simple check that the target cell is currently off will be fine for now.

```
GridTest>>testCellInteractions

    self assert: (self generateDemoGrid at: 5@1) isOff
```

While writing the generator method we found myself getting confused about which part of the x@y notation to use for rows and columns. That's a tip-off that we should rewrite some of our methods to make this task easier. Maybe we should have written this test before we even wrote the at: and at:put: methods on Grid? Indeed this is a clear advantage of writing test first: they act as the first client of the code and give you a quality control on it. We'll have to remember to go back and fix these names to be more intention revealing.

In the future we should provide a way to print a grid in ascii to debug it and create a grid based on an ascii representation.

It is a good point to save your work.

11.8 Conclusion

Now all the structure elements of a game are in place and tested so we will be able to start working on the beam computation.

BTC-2014-05-31 Similar to how http://squeak.preeminent.org/tut2007/html/035A.htm provides a check sheet, maybe it would be good to direct the reader how to use Monticello to "Compare Changes".

Chapter 12

Laser Beam Path

We are ready to work on the way to represent the laser beam and its path through the cells. Before jumping in the code let us think a bit about what we will need. We need to have a collection of elements describing the beam (probably such elements will be the basis to represent visually the beam when we will start adding visual elements). When the beam will exist to the east of a given cell we will need to know to which other cell it will enter.This means that we will need a way to handle the cell locations and the interpretation of the direction based on such location.

12.1 Laser Path Element

When we consider the path of the laser beam through the grid, it's pretty easy to imagine the path as an ordered collection. The elements in the path may not be strictly cells since we will need to be aware how the laser is entering each cell. Let us get started by defining a new class named LaserPathElement to represent path element.

Such element is bound to a given cell and an entry side, therefore we add the following instance variables to fully characterize an element.

```
Object subclass: #LaserPathElement
    instanceVariableNames: 'cell entrySide'
    classVariableNames: ''
    category: 'LaserGame-Model'
```

Add the corresponding accessors.

12.2 Adding a Class Comment

So let us first add a class comment, for example as follows

```
I represent the part of a laser beam that is associated with a cell and with an entry
    side.
```

12.3 Back to the Grid

Go back to the Grid and add an instance variable for the laser beam path and then add its accessors. We'll work out initialization in a few minutes.

```
Object subclass: #Grid
  instanceVariableNames: 'cells laserIsActive numberOfRows numberOfColumns
    laserBeamPath'
  classVariableNames: ''
  category: 'LaserGame-Model'
```

We also need a method to answer the starting cell when the laser is activated. This will always be the cell in the bottom left corner.

```
Grid>startingCell

  ^ self at: (1@ self numberOfRows)
```

12.4 Adding Location to Cell

To be able to compute from a cell and a direction its neighbouring cell we will enhance the cells. Each cell should store its own grid location. This will make navigation and position calculations easier. Without the stored location, we can still find specific cell locations within the grid by scanning through all the cells until we find the identical one we are looking for, then we would answer that cell's key (a position) in the cell dictionary. Holding the location within the cell is easier and much faster.

Add the gridLocation instance variable and its accessors.

```
Object subclass: #Cell
  instanceVariableNames: 'activeSegments exitSides gridLocation'
  classVariableNames: ''
  category: 'LaserGame-Model'
```

We should not forget to set the grid location to the cells when the grid adds the cell to its own structure. We modify then the method at:put: of the class Grid as follows:

```
Grid>>at: aPoint put: aCell

    aCell gridLocation: aPoint.
    self cells at: aPoint put: aCell.
```

12.5 Handling Directions

We've been using two kinds of systems to describe cell locations and laser beam direction. In one situation we use x@y (column x, row y) objects to describe where cells are located within the grid. We describe laser beam direction with the symbols #north, #east, #south and #west. We need a way to allow us to combine these concepts when we start to navigate our laser beam across the grid and through cells.

Let us think about a solution: what would be good is to have objects that capture that for example going east is increasing of one the x coordinates. If we are in cell 5@5 going east we should get to the cell 6@5. Similarly going south is changing the y coordinates.

A first reflex could be to write some conditions but we can do better. Indeed we do not want to have to have tests and conditionals and we can imagine that we can get different objects representing different directions and each one answering the delta that should be applied to get from one cell to the other one.

SD: should be better explained. It took me a while to understand it.

SD add a diagram explaining how to use Direction

For this we will create classes to represent the different direction and we will start by defining a common superclass named GridDirection. We will then define a couple of messages on these classes.

```
Object subclass: #GridDirection
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

And we can add a comment.

```
I represent directions for the beam path accross the grid cells.
```

Then we can define one suclass for each direction as follows

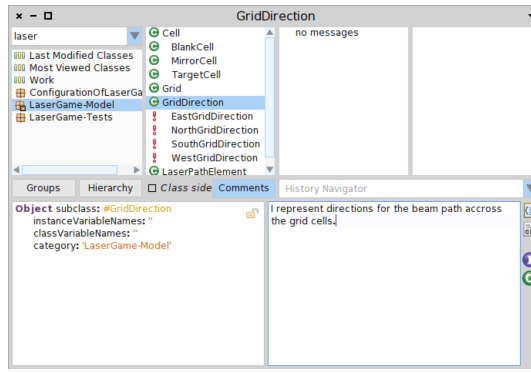

Figure 12.1: A hierarchy of directions.

```
GridDirection subclass: #EastGridDirection
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

```
GridDirection subclass: #WestGridDirection
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

```
GridDirection subclass: #NorthGridDirection
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

```
GridDirection subclass: #SouthGridDirection
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Model'
```

You should obtain an inheritance hierarchy as described in 12.1.

Now for each subclasses, we can define some messages: the vector which returns the delta from the cell to the direction and directionSymbol which returns the symbol encoding the direction.

For example we want to write

```
SouthGridDirection directionSymbol
  -> #south
```

```
SouthGridDirection vector
  −> 0@1
```

Since we want to ask the class for its encodings, we will define *class* methods on each subclasses as below.

```
EastGridDirection class>>vector
   ^ 1 @ 0

EastGridDirection class>>directionSymbol
   ^ #east
```

```
NorthGridDirection class>>vector
   ^ 0 @ −1

NorthGridDirection class>>directionSymbol
   ^ #north
```

```
SouthGridDirection class>>directionSymbol
   ^ #south

SouthGridDirection class>>vector
   ^ 0 @ 1
```

```
WestGridDirection class>>directionSymbol
   ^ #west

WestGridDirection class>>vector
   ^ − 1 @ 0
```

Now we can define a method named directionFor: returning the class of the direction associated to directiondescribed by a symbol.

```
GridDirection directionFor: #east
−> EastGridDirection
```

Here is the class method definition:

```
GridDirection class>>directionFor: aDirectionSymbol

   ^ self subclasses
         detect:  [ :cls | cls directionSymbol = aDirectionSymbol ]
```

12.6 Adding Some Tests

Now we need a unit test for the grid direction vector code. So let us add the new class GridDirectionTestCase is a subclass of TestCase of course.

```
TestCase subclass: #GridDirectionTest
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'LaserGame-Tests'
```

```
GridDirectionTest>>testDirectionSelection
    | direction |
    direction := GridDirection directionFor: #north.
    self assert: direction = NorthGridDirection.
    self assert: direction vector = ( 0 @ -1 ).

    direction := GridDirection directionFor: #east.
    self assert: direction = EastGridDirection.
    self assert: direction vector = ( 1 @ 0 ).

    direction := GridDirection directionFor: #south.
    self assert: direction = SouthGridDirection.
    self assert: direction vector = ( 0 @ 1 ).

    direction := GridDirection directionFor: #west.
    self assert: direction = WestGridDirection.
    self assert: direction vector = ( -1 @ 0 ).
```

Once all the tests pass, it is a good time to publish your code.

12.7 Back to the Laser Beam Path

Now we are read to work on laser path elements. Let us start to define a new class method that eases to instantiate LaserPathElement objects. Do you have any idea how to proceed? When we create a LaserPathElement we should mention the cell it is acting on and also the side of the cells we are talking.

```
LaserPathElement  class>>cell: aCell entrySide: aDirectionSymbol

    | element |
    element := self basicNew.
    element
        cell: aCell;
        entrySide: aDirectionSymbol.
    ^ element
```

SD: add a test

We can also add another method taking a direction instead than a symbol. Objects are always more powerful than mere data so we should favor their use.

```
LaserPathElement class>>cell: aCell entrySideDirection: aDirection

  ^ self cell: aCell entrySide: aDirection directionSymbol
```

SD: add a test

A key functionality that we need is a method that given a path element answers the next cell on grid. Here we will clearly see the importance of direction. Let us think a bit, as input we have an path element (which is linked to a cell and direction), a cell that know how the beam exits and a grid which contains all the cells.

```
LaserPathElement>>nextElementIn: aGrid

  | direction newLocation nextCell |
  direction := GridDirection directionFor: (self cell exitFor: self entrySide).
  newLocation := self cell gridLocation + direction vector.
  nextCell := aGrid at: newLocation.
  ^ nextCell isNil
    ifTrue: [ nil ]
    ifFalse: [ self class cell: nextCell entrySideDirection: direction ]
```

There is a dependency on some behavior in Grid that is expected here. When a location is given that would be invalid for the grid, because of indexes like 0 or ones larger than the number of rows or columns. The expectation here is that the cell found would be set to nil. We should go back to the Grid code and ensure we handle this. Just add the "ifAbsent: []" code.

SD: Add a test

SD: Add more tests

GridDirection directionFor: (self cell exitFor: self entrySide). can be turned into cell method.

12.8 About Encodings Default Values and Shared Pools

Note that there are other ways in Pharo to define constant: Shared pools