At this point we are going to start building a simple application to demonstrate some basic Seaside functionality and introduce you to more Smalltalk and Pharo as we go along. The application will simulate a trivial airline reservation system in which you find and select a flight. Instead of starting with a user interface, we will start with a domain model that holds a date/time and price.

1. Start the Seaside One-Click Experience and in the System Browser click on 'GLASS' in the first column to get a new class-creation template. Edit the template to match the following and save the text.

```
Object subclass: #FlightInfo
   instanceVariableNames: 'when price'
   classVariableNames: ''
   category: 'GLASS'
```

   This creates a new subclass of Object named 'FlightInfo,' gives it two instance variables ('when' and 'price'), and puts it in the 'GLASS' category.

2. Next we will define a couple 'accessor' (or 'getter') methods that simply return the value of the instance variable. (The method return is signaled by the up arrow or caret at the beginning of an expression.) These methods are necessary because in proper Smalltalk there is no direct structural access to the instance variables (or properties or fields) of an object. This language design enforces encapsulation and allows the implementation of an object to change (perhaps the 'price' is calculated every time it is requested rather than saved with the object). Note that these are two separate methods.

   To get to a method creation template, click on 'GLASS' in the first column, click on 'FlightInfo' in the second column, click on '-- all --' in the third column, click in the text area at the bottom of the system browser, and finally select all using <Ctrl>+<A> (or click in the text area after the end of the last line). Enter the first method (three lines), save (using <Ctrl>+<S>), and then select all, delete, and enter the second method (three lines), and save.

```
price

   ^price.
```

```
when

   ^when.
```

 Copyright © 2016 by GemTalk Systems LLC

3. Next we will add a method to calculate the price. (Of course, this calculation is purely arbitrary and used in this tutorial to give an example of some further Smalltalk syntax.) This method has a temporary variable (hours) that is declared in line 3 (within the vertical bar characters) and set in line 4 with the assignment operator (recall that Smalltalk uses colon-equals which leaves plain equals available for comparison). When you enter the code, you may leave out the line number comments (or leave them it if you want!).

```
calculatePrice

"3"   | hours |
"4"   hours := when asSeconds // 3600.
"5"   price := (hours degreeSin asScaledDecimal: 2) * 30 + 300.
```

This is the first time we have seen multiple messages in an expression and it calls for some explanation. In Smalltalk, the three message types (unary, binary, and keyword) have precedence such that all unary messages are evaluated in left-to-right order before any other messages are evaluated. Thus, the 'asSeconds' message is sent to the object in the 'when' instance variable and the 'asSeconds' method in DateAndTime returns an object (we know it is an Integer). Now that we have evaluated all the unary messages in the expression, we move to the binary messages (the next level of precedence) and evaluate them in left to right order. The expression on line 4 has only one binary message, the integer divide message (//). By taking an integer that represents seconds and dividing by 3600 and ignoring the fractional portion, we get an integer that represents hours. Note that the '//' message is simply a message that is understood by instances of Number (a superclass of Integer).

Line 5 is an even more complex expression. In expression evaluation, precedence is always given to parenthesis so we start with the subexpression in the parenthesis. The parenthesis pair contains two messages, 'degreesSin' (a unary message) and 'asScaledDecimal:' (a keyword message). Since unary messages are evaluated before keyword messages, we first send the message 'degreesSin' to the receiver 'hours' (an Integer we got from line 4). The object returned by the 'degreesSin' method is an instance of Float (that will be in the range of -1 to +1). Next we send the message 'asScaledDecimal:' to the instance of Float to convert it to a number that shows two digits past the decimal point. This completes the expression in the parenthesis.

After evaluating the expression in the parenthesis, we have three objects (an instance of ScaledDecimal and two SmallIntegers) and two binary messages ('*' and '+'). Evaluating them left-to-right, we send the '*' message to the ScaledDecimal with an argument of '30.' The method that responds to the '*' message in ScaledDecimal returns another ScaledDecimal instance representing the number obtained from the multiplication by 30. This object is sent the '+' message with the argument of '300' and returns another ScaledDecimal that will happen to be in the range of 270 to 330. A reference to this object is placed in the 'price' instance variable for the receiver (an instance of FlightInfo). (You might wish to explore the impact of left-to-right evaluation of binary operators by trying 2 + 3 * 4 in a workspace.)

 Copyright © 2016 by GemTalk Systems LLC

4. Now we will add a couple methods that set the 'when' instance variable and, as a side-effect, recalculate the price.

```
when: aDateAndTime

   when := aDateAndTime.
   self calculatePrice.
```

```
addHours: anInteger

   when := when + (Duration hours: anInteger).
   self calculatePrice.
```

5. Next, add a method to print the object on a stream. This is used to create a text representation of the object. This method is a common one in classes, and can be particularly helpful for debugging.

```
printOn: aStream

   when printYMDOn: aStream.
   aStream space.
   when printHMSOn: aStream.
   aStream skip: -3.
   aStream nextPutAll: ' $'.
   price printOn: aStream.
   aStream skip: -2.
```

6. Finally, add a method to initialize the object. In Pharo, the 'initialize' method is called on all newly-created objects to give them a chance to give their instance variables initial values.

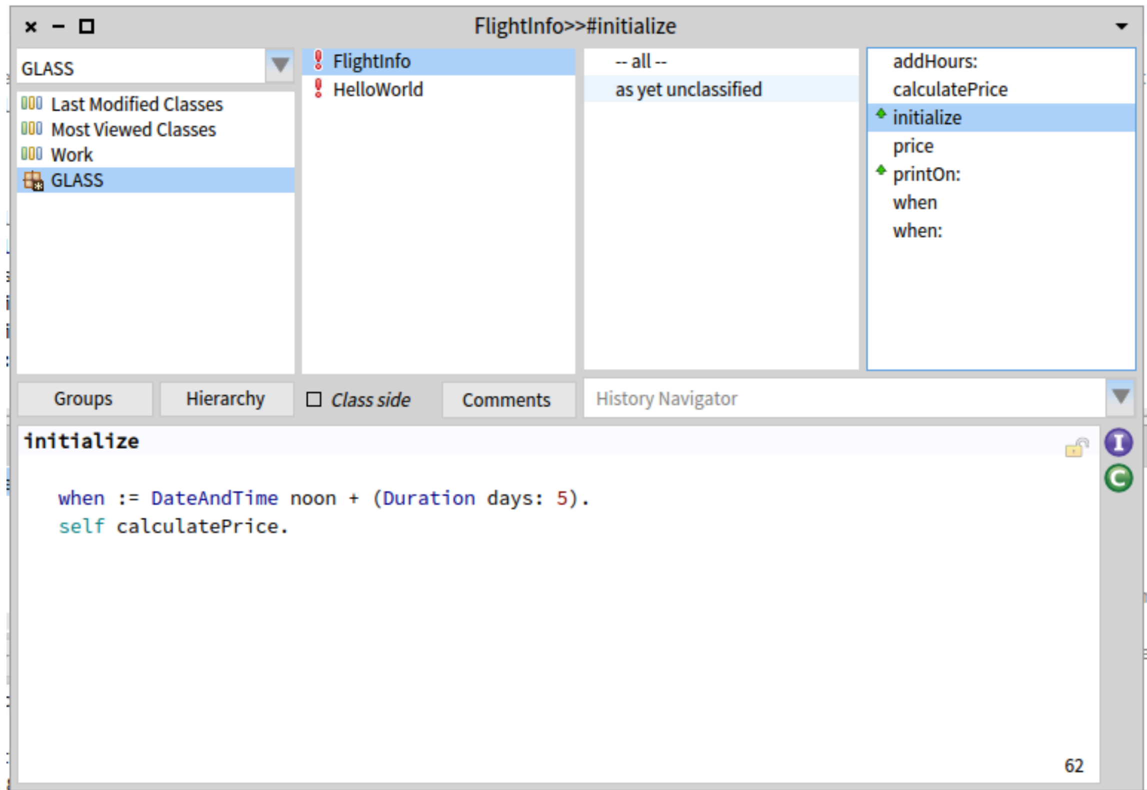
```
initialize

   when := DateAndTime noon + (Duration days: 5).
   self calculatePrice.
```

Some Smalltalk dialects (such as GemStone) do not do the automatic initialize but instead leave it up to the programmer to override 'new' on the class side to call 'initialize' if instances of the class require it. If you are writing an application that might be ported from one dialect of Smalltalk to another, you can minimize these differences by subclassing your domain objects from GRObject. GRObject is provided by Grease, a package designed to make porting easier.

Copyright © 2016 by GemTalk Systems LLC

7. At this point the System Browser should show seven methods. The 'initialize' and 'printOn:' methods have a green arrow pointing up to signify that your method overrides a superclass implementation of the same name.



8. Now that we have our domain object defined we are ready to create a user-interface class. In Seaside, the basic user interface class is a subclass of WAComponent and any such component can be a root (the starting point for an application). In the System Browser, click on 'GLASS' in the first column to get a new class-creation template. Define the new class as follows:

```
WAComponent subclass: #FlightInfoComponent
   instanceVariableNames: 'model'
   classVariableNames: ''
   category: 'GLASS'
```

9. Perhaps the most important method in a Seaside component is 'renderContentOn:' that creates the HTML. For starters, just to demonstrate that we have properly created things, we will provide a trivial implementation. Click in the third column to get a method creation template.

```
renderContentOn: html

   html heading: DateAndTime now.
```

 Copyright © 2016 by GemTalk Systems LLC

10. Now we need to inform Seaside that this new component can be used as a root component (this should be familiar from chapter 2). In the Workspace, evaluate the following:

```
WAAdmin register: FlightInfoComponent asApplicationAt: 'FlightInfo'.
```

11. If the above steps were successful, you should be able to open a web browser on http://localhost:8080/browse and see 'FlightInfo' at the top of the list (the initial uppercase letter causes the name to sort first).

12. Click on the 'FlightInfo' link and note that a timestamp is displayed. Click refresh a few times and note that the time changes.

13. Instead of displaying the current date/time, we really want to display our domain model object, an instance of the 'FlightInfo' class. In order to create a new instance of our model, override the 'initialize' method in FlightInfoComponent. This method is called whenever a new application object is instantiated by Seaside in response to a request for the root page of the application.

```
initialize

   super initialize.
   model := FlightInfo new.
```

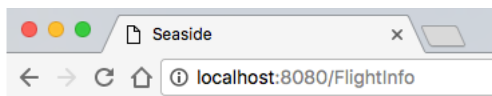
The System Browser should now show two methods for the FlightInfoComponent class, 'initialize' and 'renderContentOn:'. The initialize method also has a green up-arrow next to it to alert you to the fact that this method overrides a superclass method of the same name.

14. Now we will modify our 'renderContentOn:' methods as follows (the modified line is in bold). This is intended to cause the model to be displayed using its 'printOn:' method.

```
renderContentOn: html

   html heading: model.
```

15. Return to your web browser and click the <Refresh> button. You should now see a date/time and price.



2016-09-09 12:00 $302.09

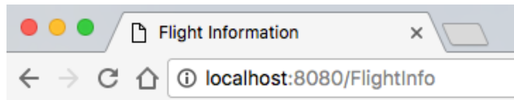
Copyright © 2016 by GemTalk Systems LLC

16. Most web pages provide a title that is used for the window title and (if the browser supports it) the tab title. By default all Seaside pages have the title 'Seaside' (as you can see above) which we would like to change. In the System Browser, add a new method to FlightInfoComponent:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: 'Flight Information'.
```

17. Refresh your web browser and note that the window title and tab heading have changed.



2016-09-09 12:00 $302.09

18. Save your image.

While not much more sophisticated than the "Hello World!" example in Chapter 3, we will continue to use this application to investigate other aspects of Smalltalk, Pharo, and Seaside in subsequent chapters. An important point here is how Seaside generates web pages—completely through Smalltalk code. This is a contrast to the template approach used by most web frameworks.

Web Frameworks

A web application receives a request from a web server and is expected to generate an appropriate response—generally a document formatted using the Hyper-Text Markup Language (HTML)—and pass it back to the web server to be sent to the client browser. An early approach that remains popular among web frameworks is to create a document that looks like HTML (the *template*) but has some additional elements (defined with special tags) used to specify additional content. For example, most web sites have a common footer that includes a copyright notice. Rather than hard-coding the same footer in each page, one can reference (or include) a second file at a particular point in the first file and the current contents of the second file will replace the include directive in the first file. This makes the first file less cluttered and allows changes to the footer to be made more easily and consistently.

An example of this approach is the *include* directive in Java Server Pages (JSP):

```
<%@ include file="footer.jspf" %>
```

In JSP, included files generally have the extension "jspf" (for JSP Fragment). A similar example can be found in ColdFusion (a commercial product from Adobe):

```
<cfinclude template="footer.cfm">
```

The extension ".cfm" identifies a CFML (ColdFusion Markup Language) document. In addition to an include directive, templating systems generally provide ways of evaluating expressions and using the result in the generated page.

For example, a CFML page containing the following element would have the text between the hash characters (#) replaced when the page is requested:

```
This page generated at <cfoutput>#Now()#</cfoutput>
```

A similar example in JSP would look like the following:

```
This page generated at <%= new java.util.Date() %>
```

Embedding programming logic in HTML is not too difficult for a relatively simple web site but does not scale well to a complex application where if/then/else and loops are needed.

In contrast to the templating approach where the program is embedded in what is otherwise an HTML document, some web frameworks embed or create HTML in what otherwise looks like a traditional program. A truly simplistic example of this approach in Perl comes from http://inconnu.isu.edu/~ink/perl_cgi/lesson1/hello_world.html. Obviously, this is an impractical approach, but it gives you an idea of the other extreme.

```
    #!/usr/bin/perl
    print "Content-type: text/html\r\n\r\n";
    print "<HTML>\n";
    print "<HEAD><TITLE>Hello World!</TITLE></HEAD>\n";
    print "<BODY>\n";
    print "<H2>Hello World!</H2>\n";
    print "</BODY>\n";
    print "</HTML>\n";
    exit (0);
```

Seaside is a web framework in which the HTML is generated using regular Smalltalk programming. The power of Smalltalk comes from its ability to represent complex domain models as a rich interaction of simpler objects.

 Copyright © 2016 by GemTalk Systems LLC