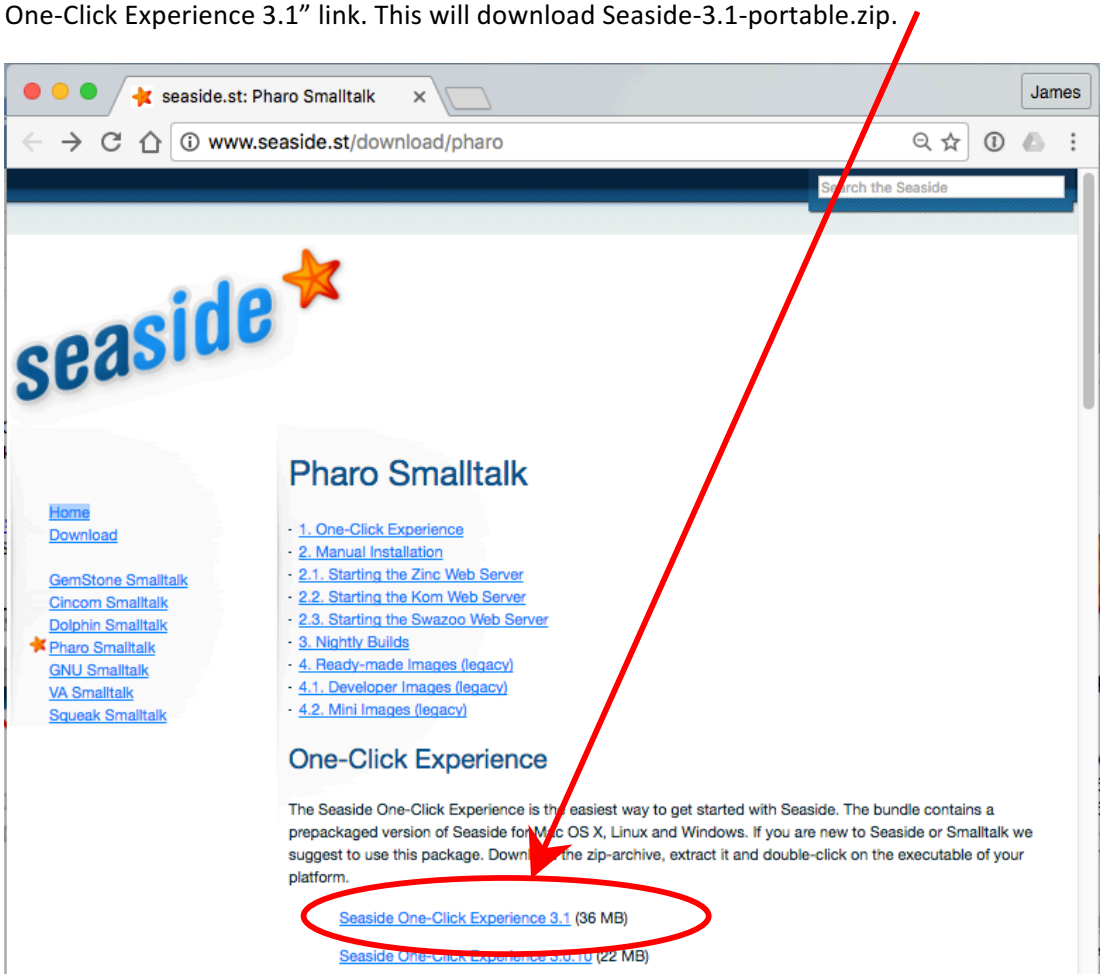
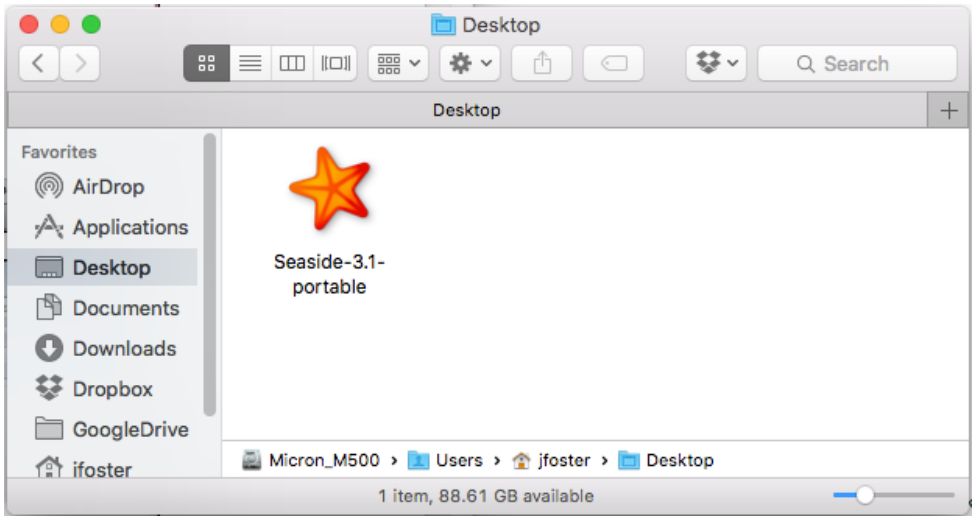
For our first Seaside project we will use the "Seaside One-Click Experience" in Pharo. If this tutorial came with a DVD, copy 'Seaside-3.1-portable.app' to your desktop and skip to step #3. Otherwise, follow steps 1 -2 to download Seaside from the web.

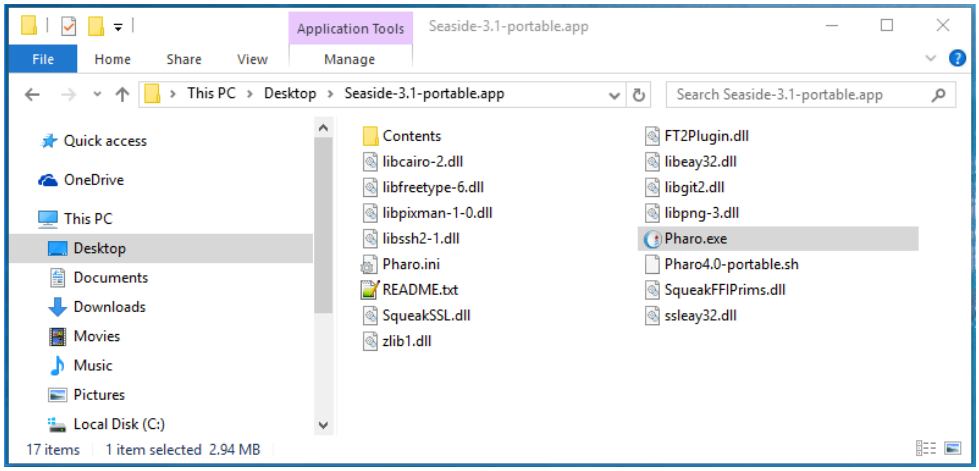
1. Open a web browser on http://www.seaside.st/download/pharo/ and click on the "Seaside One-Click Experience 3.1" link. This will download Seaside-3.1-portable.zip.



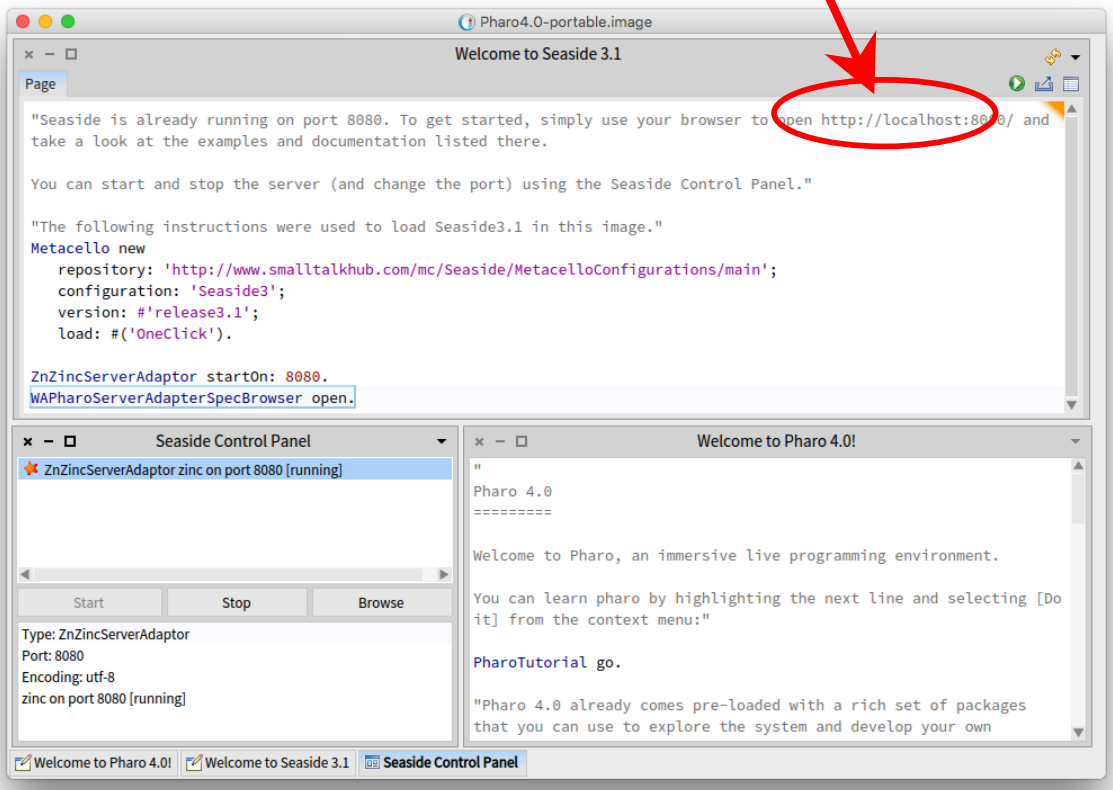
2. Locate and unzip the file (if the download process did not unzip it for you). This will create a folder named "Seaside-3.1-portable.app" that you should place in a convenient location (such as the desktop). Note that if you are on a Macintosh this "folder" will be an application package.

3. Find the executable appropriate for your operating system: Mac, Windows, or Linux. On the Macintosh, the folder will appear as a starfish icon representing an application package. On Windows, the folder can be opened to find various items, including an executable named 'Pharo.exe'. On Linux, the folder can be opened to find various items, including 'Pharo4.0-portable.sh'. Double click on the appropriate icon to launch the executable.

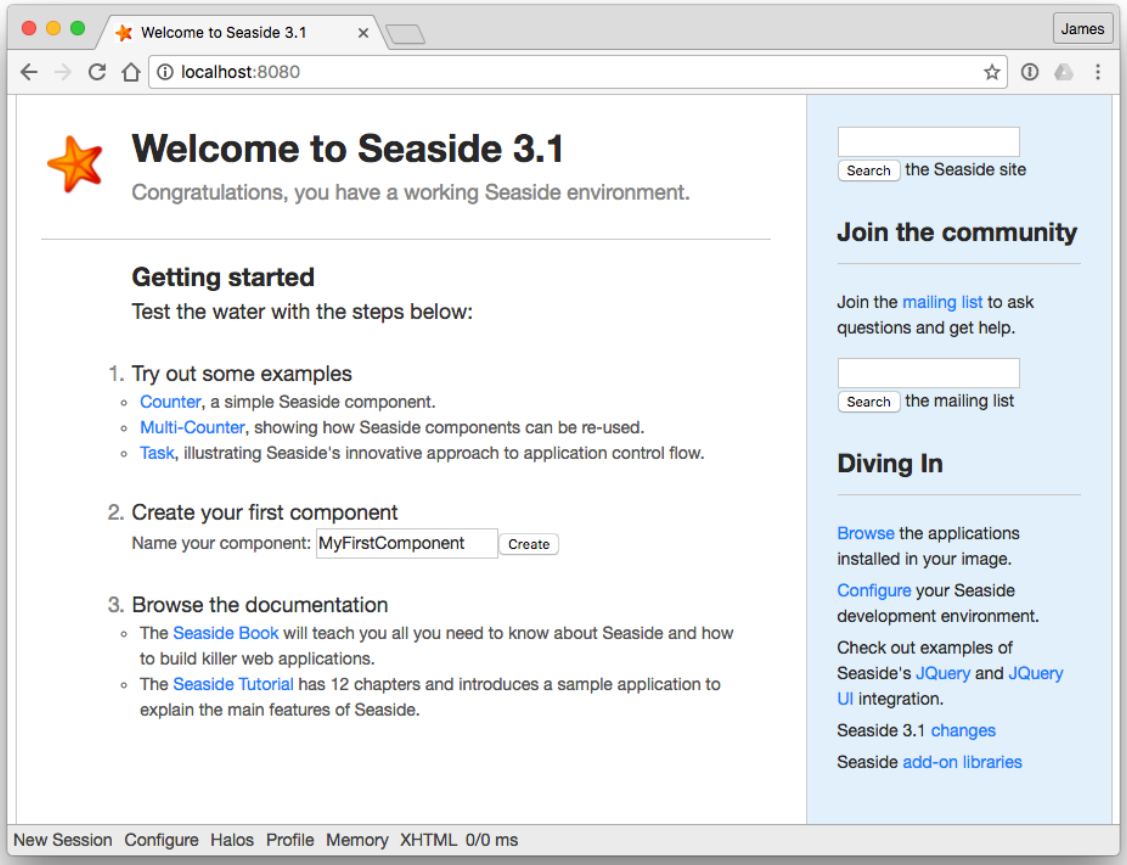




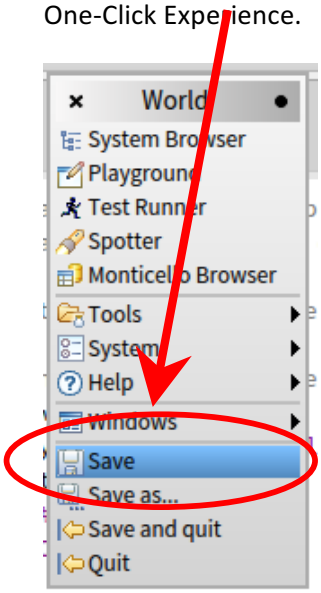
4. When you launch the executable, you are actually starting a web server in Pharo Smalltalk. As we continue through this tutorial, we will refer to this executable as “Pharo” to distinguish it from the Seaside code framework that Pharo contains. Like most Smalltalk dialects, Pharo runs as a “virtual machine” on your host operating system. You should see a new main window containing three smaller windows. This main window gives you a graphical user interface into an object space loaded into memory from the file ‘Pharo4.0-portable.image’ that was part of the earlier download. This ‘image’ is a copy of one with Seaside and a simple web server already loaded. The inner windows will be the same on all three platforms and will not look like windows created by other applications on your platform. One of the windows, labeled “Welcome to Seaside 3.1,” contains a comment telling you that a web server is already running and you can point your web browser to http://localhost:8080/ to give it a try.



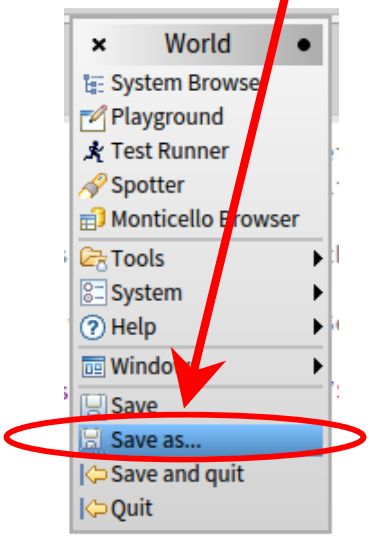
5. Open a web browser on http://localhost:8080/ to confirm that things are up and running correctly. You can poke around a bit here, but don’t get too distracted at this point. We’ll be exploring Seaside in more detail in Chapter 3.



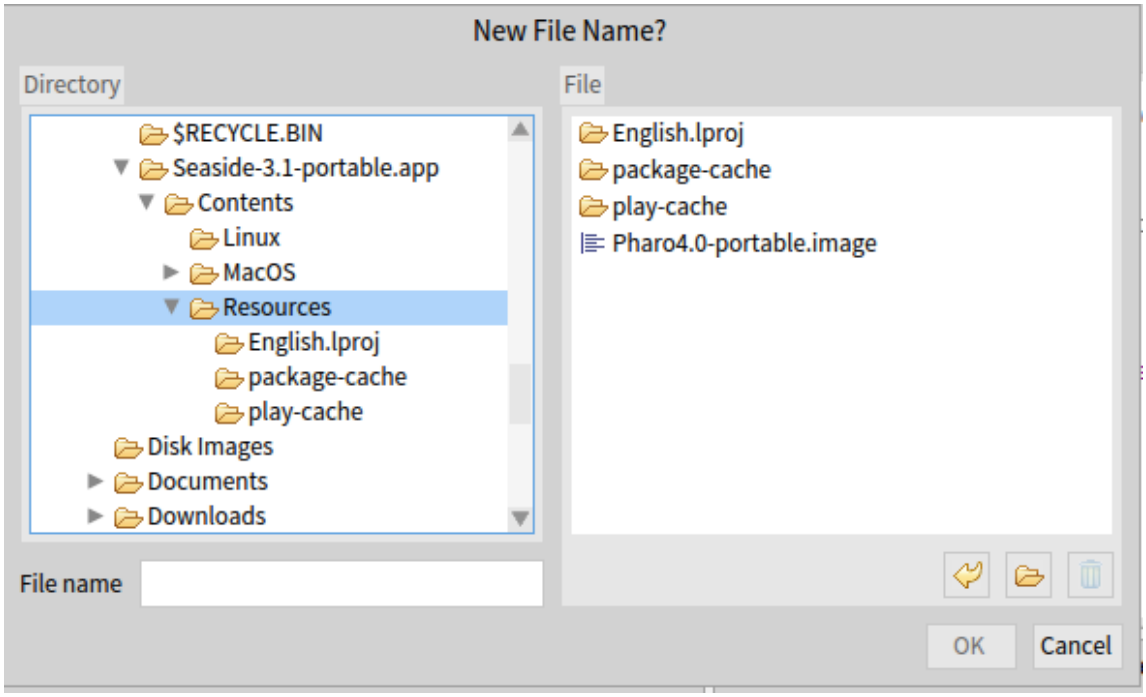
6. As we go along, we will be creating and editing code in the Pharo object space. To make the change persistent, you need to make a snapshot of the current object space—creating a new "image" (following a camera metaphor). To do this, left-click on the background (or desktop) to bring up the World menu and select 'save' from the menu. This will write out your changes to the default 'Pharo4.0-portable.image' file that will be read when you next launch the Seaside One-Click Experience.



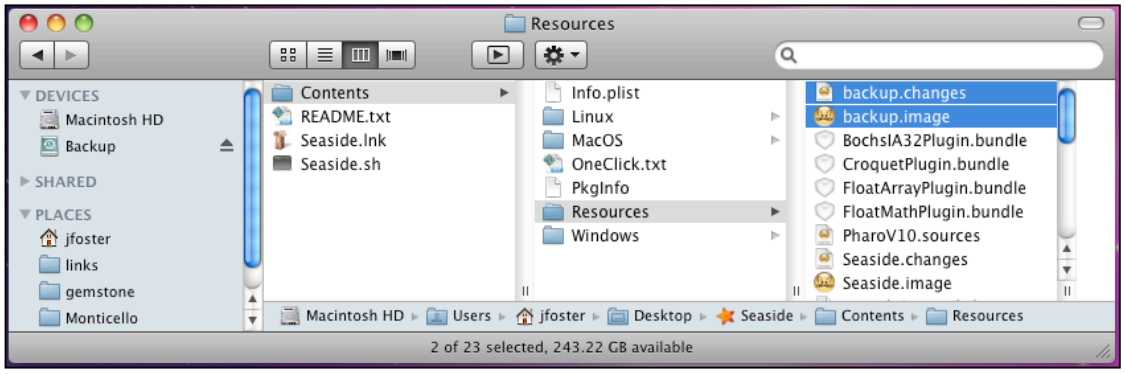
7. Smalltalkers refer to the above action as "saving the image," and this is a handy way to preserve your changes and the environment. Think of it as a suspend action (or hibernate) for your computer. When you come back and restart your computer the same windows will be open in the same location with the same contents. If you are about to try something that might cause a problem, you could save the image before taking the risky action. Then, if things go bad you can quit without saving (see step #10 below) and simply reopen the saved image to get back to the prior state. Alternatively, you can change the name of the saved image by selecting the 'save as…' menu item.



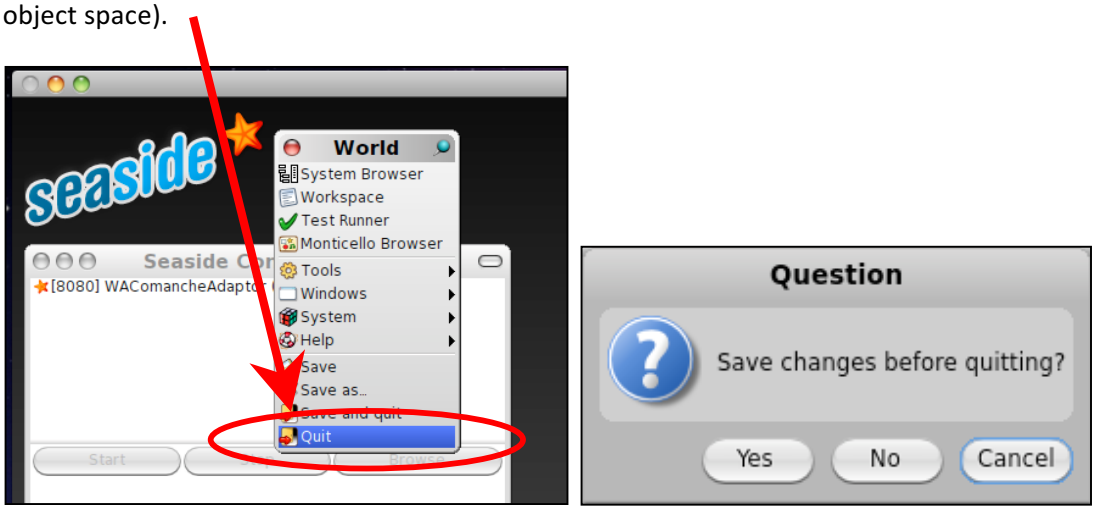
8. This will open a dialog asking for a File name for the new image. Name the new image ‘backup.image’ and click OK or press <Enter>. This gives us a ‘clean’ or ‘virgin’ image that can be restored without downloading everything again from the web or copying from a CD/DVD.



9. To see the new file, open the application folders ‘Contents’ then ‘Resources’ (on the Mac you will need to right-click on the Application icon and select ‘Show Package Contents’). If you ever want to start over, simply delete ‘Pharo4.0-portable.changes’ and ‘Pharo4.0-portable.image,’ copy ‘backup.changes’ and ‘backup.image,’ and rename the copies to ‘Pharo4.0-portable.changes’ and ‘Pharo4.0-portable.image’ (note that capitalization might be important, depending on your platform).

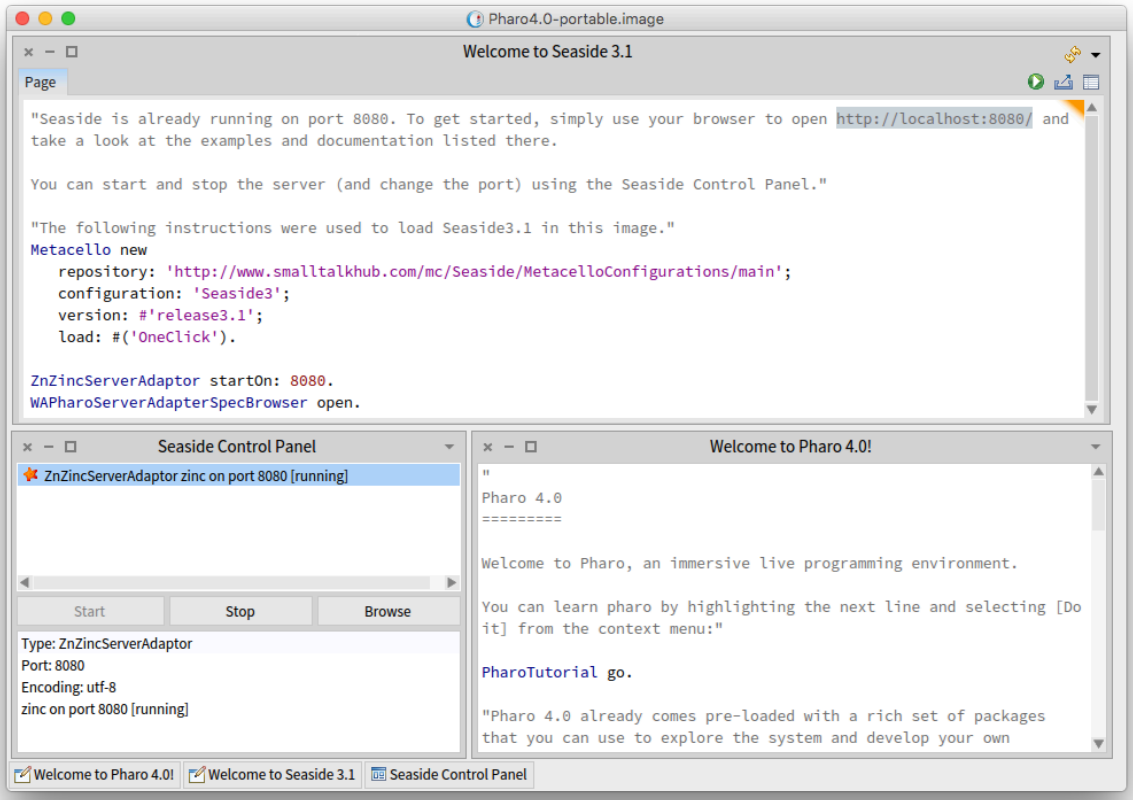


10. Now we are ready to quit the application. Left click on the background to bring up the (now familiar) World menu and select the last item, 'quit.' In the 'Save changes before quitting?' dialog, select 'No' to quit without saving (since we already saved two images of the current object space).



When learning a new programming language it is traditional to start with a ‘Hello World!’ application (see http://en.wikipedia.org/wiki/Hello_world_program). We will follow that practice and use this opportunity to introduce you to the programming environment and the Smalltalk programming language.

1. Start by launching the Seaside One-Click Experience executable as described in chapter 1. This will open a copy of ‘Pharo4.0-portable.image’ using Pharo. Three windows are already open (clockwise from the top): (1) a Workspace titled ‘Welcome to Seaside 3.1’, (2) a Workspace titled ‘Welcome to Pharo 4.0!’, and (3) a Seaside Control Panel. We will briefly examine each of these in turn.



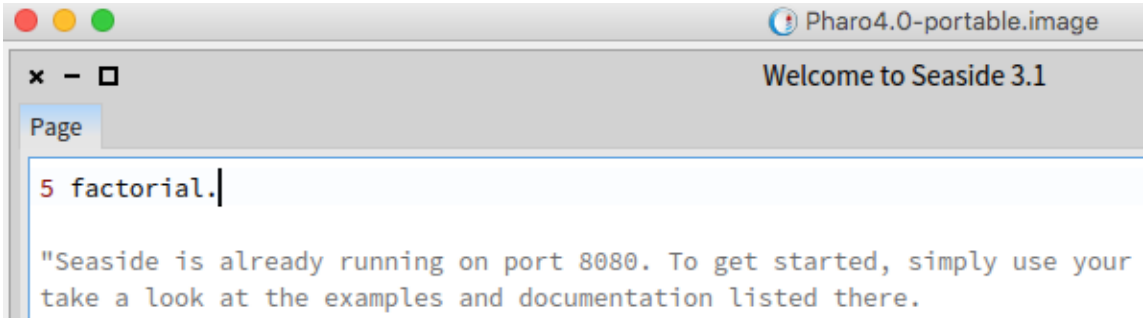
2. The two windows titled “Welcome to …” are examples of a **Workspace**, or a text area. These workspaces are pre-loaded with some information about Seaside and Pharo (respectively). The Pharo welcome workspace includes instructions for starting a tutorial. This is a good exercise.

3. You can also use a workspace to evaluate expressions and (optionally) display or inspect the results. Since this might be your first exposure to Smalltalk code, we’ll give a brief introduction to evaluating Smalltalk code in a workspace. (If you already know Smalltalk or are just impatient, you can skim or skip this discussion!)

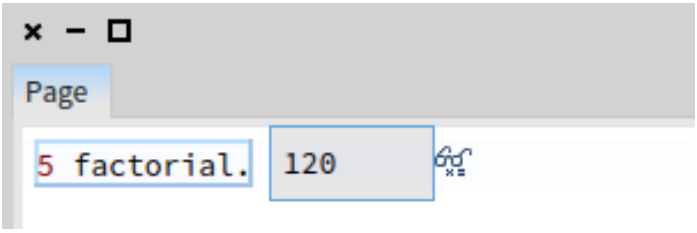
Click in a Workspace window at the top left and press <Enter> a couple times to get some space. Then go back to the top and type the following:

```
5 factorial.
```

The Workspace should look as follows:



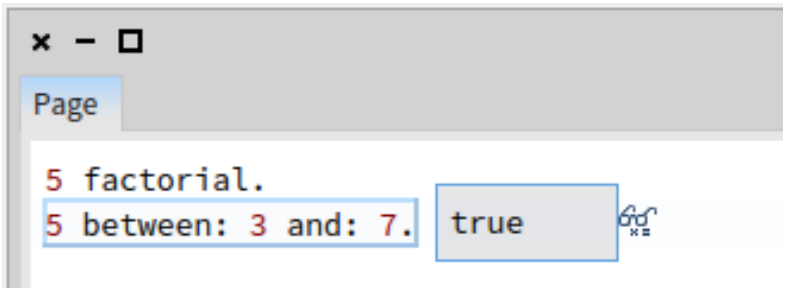
Next, press <Ctrl>+<P> (a short cut for "print-it") to evaluate the expression, and show the returned object. The result should be '120' in a box with a small pair of eyeglasses next to it. Here a *unary* message, 'factorial,' is being sent to the integer 5, which answers the integer 120.



4. In the Workspace add a new line, type the following, and 'print-it' (again, using <Ctrl>+<P>):

```
5 between: 3 and: 7.
```

The Workspace should look as follows:



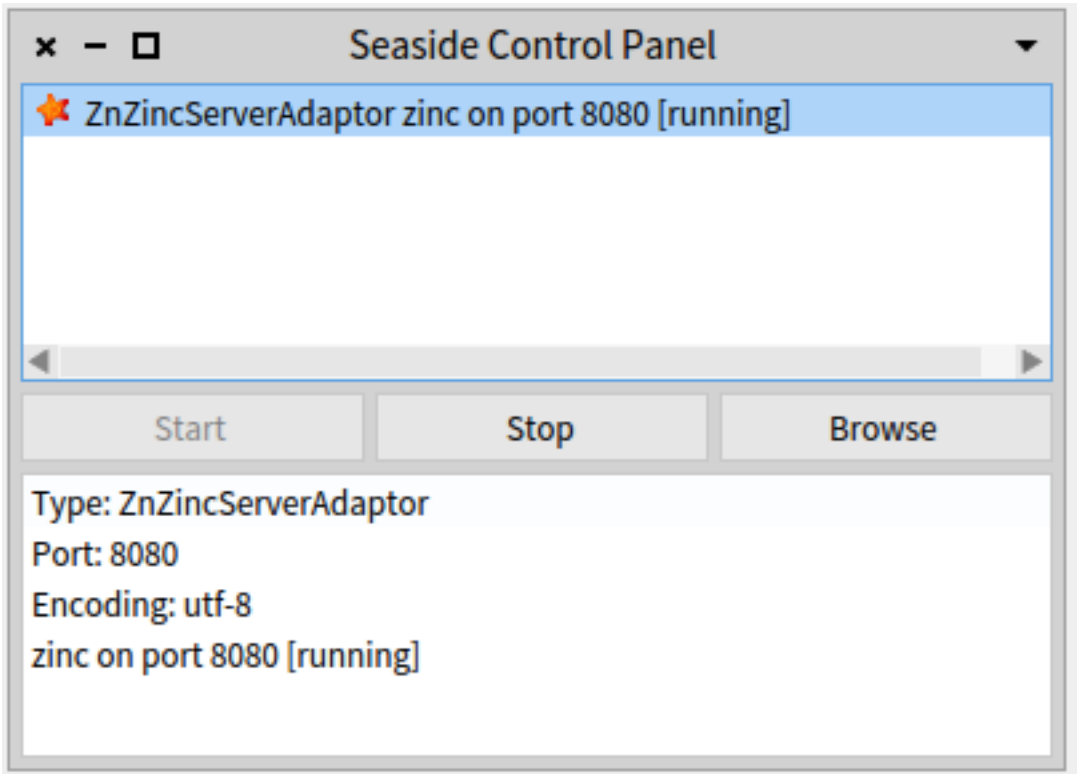
Here a single *keyword* message, 'between:and:', is being sent to the integer 5 with two integer arguments, 3 and 7, and it answers a Boolean. Smalltalk uses this keyword syntax rather than the more common position-based list of arguments, such as inRange(5,3,7), because the position-based list is ambiguous as to the meaning of each argument.

5. In addition to the *unary* message and the *keyword* message, Smalltalk defines a *binary* message that is generally used to handle what other languages do with operators. Like other messages, the format is a receiver followed by a message (with optional arguments), but this one uses punctuation characters for the message name and always has exactly one argument. In the Workspace add a new line, type the following, and ‘print-it’ (using <Ctrl>+<P>).

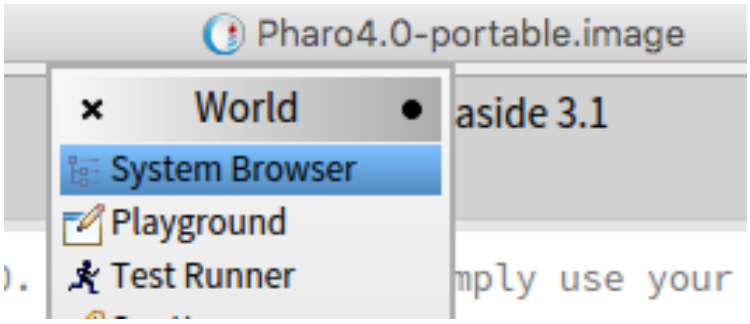
```
2 + 3.
```

   This should show 5 as the result of evaluating the expression.

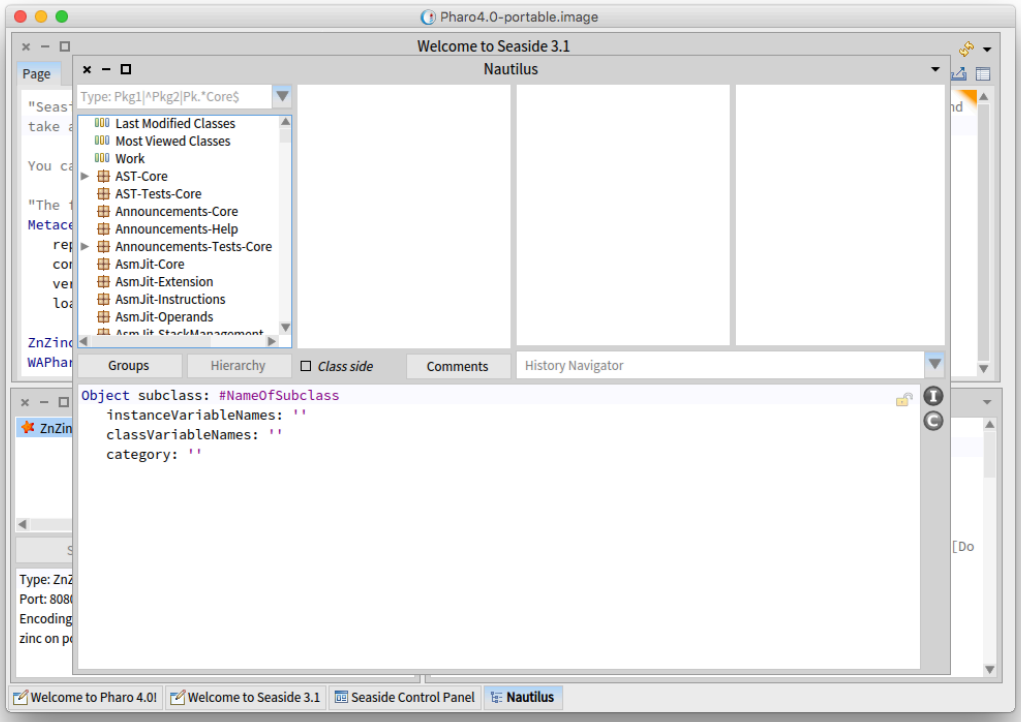
6. **The Seaside Control Panel** is used to manage the web server built into the one-click image. In it you can create, configure, and delete a server. You can select a server and inspect it.



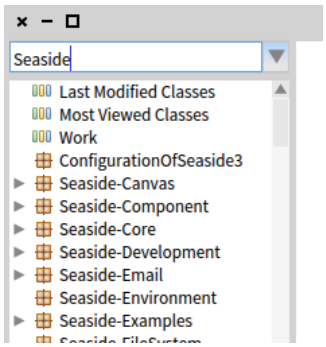
7. A **System Browser** is the tool typically used to browse code (classes and methods). To open one click on the background of the main window and select ‘System Browser’.



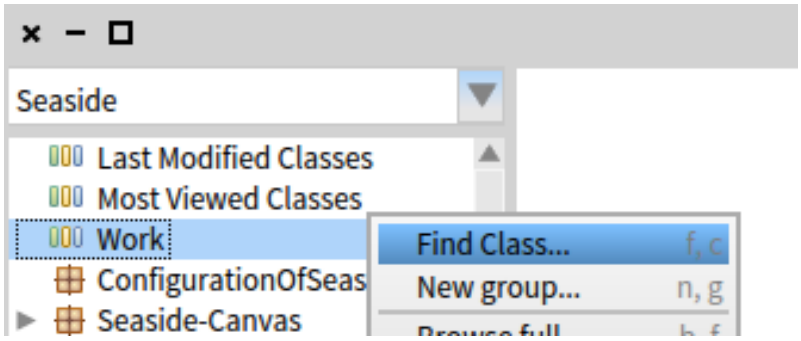
This should open a Nautilus window:



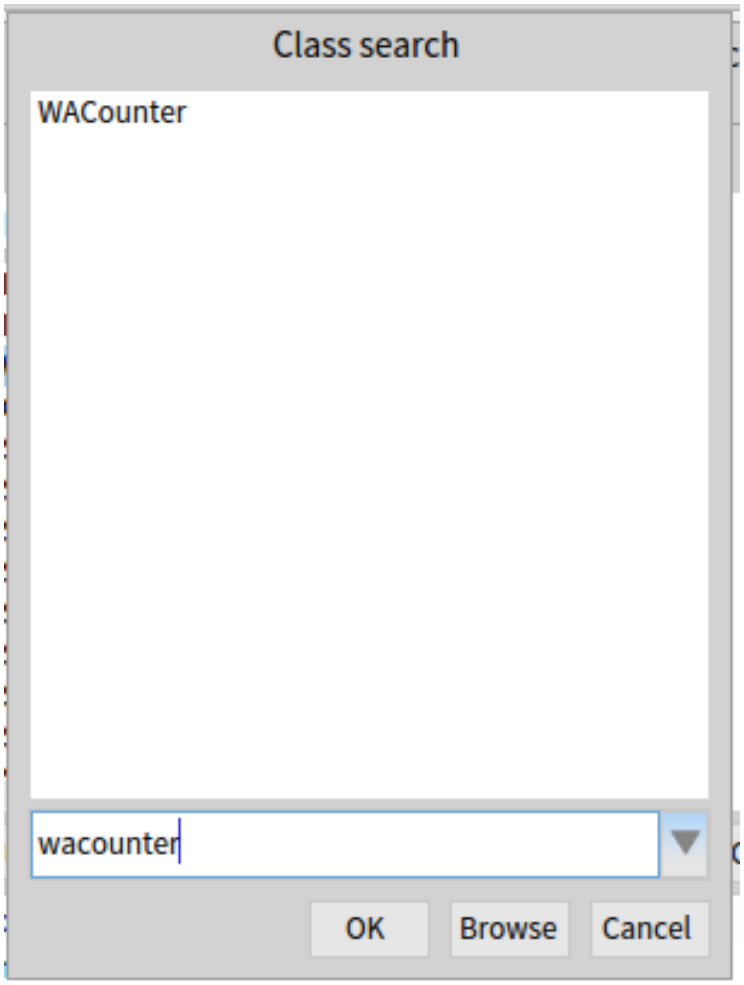
The first column contains a search drop-down text field above a list of packages containing classes. In the text field, type ‘Seaside’ (without the quotes) and notice how it filters the package list:



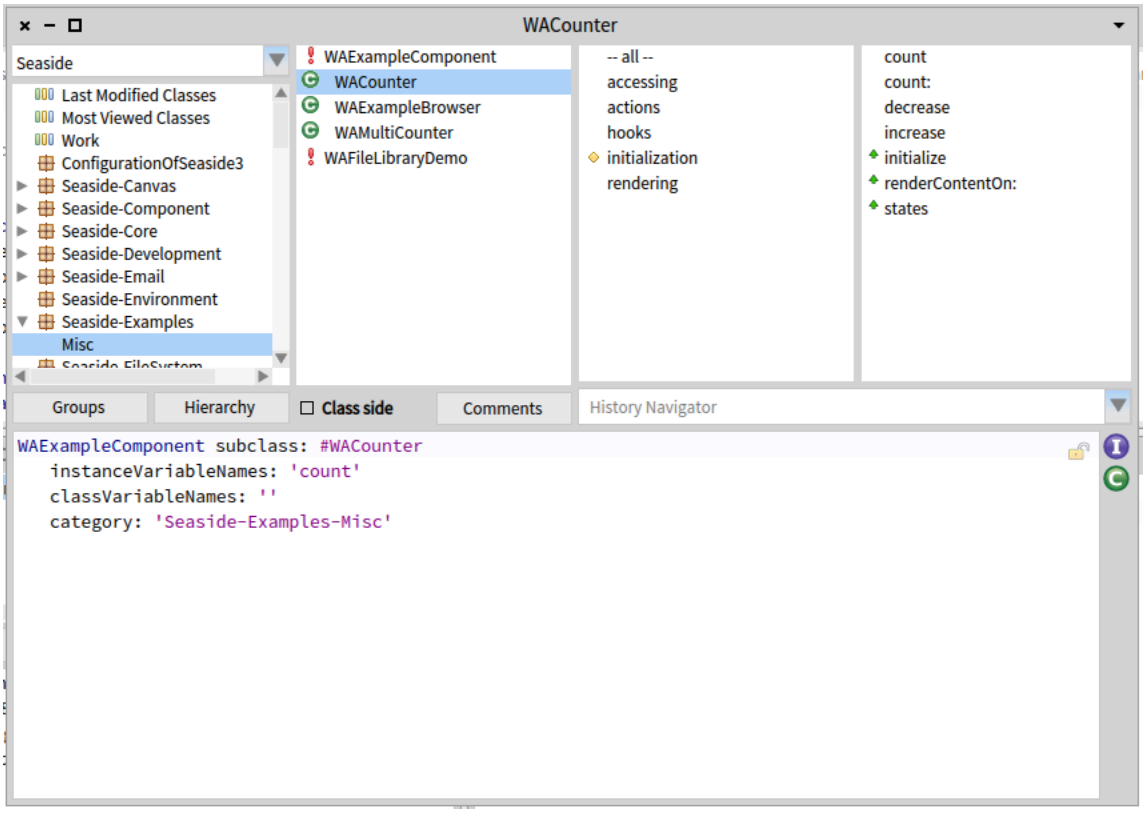
You can click the small triangle to the left of a package to expand its contents. You can also use the context menu (right or center mouse button?) to bring up a context menu. Select ‘Find Class…’ from this menu:



In the provided dialog enter ‘wacounter’ (all lowercase is fine) to filter the list:



If you click the ‘OK’ button the System Browser should show the WACounter class:



The second column lists classes that are in the selected package (‘Misc’ in ‘Seaside-Examples’) in a combined alphabetical/hierarchal order. That is, classes in the list that do not have their superclass in the list will have no indent (such as ‘WAFileLibraryDemo’). Subclasses are listed indented under their superclass if the superclass is in the same class category (‘WACounter’ is a subclass of ‘WAExampleComponent’). Selecting a class name will show the class definition in the text area in the bottom half of the window. (The red exclamation mark next to a class name indicates that it does not have a class comment. Click the Comments button to see this.)

The third column is a list of method categories for the selected class (with an extra ‘-- all --’ at the top of the list to show all methods). The fourth column is a list of methods in the class and method category selected. Selecting a method name in the fourth column will show that method’s definition in the text area in the bottom half of the window.

Below the second column (the class list) is a checkbox labeled ‘*class side*’ and a button labeled ‘Comments’. Clicking on the ‘Comments’ button will show a class comment (if available) for the selected class. The ‘*class side*’ checkbox is used to specify whether we are looking at methods that can be called on instances of the class or on the class itself. When the browser has selected the class ‘WACounter’ we see that it has five method categories (including ‘rendering’) and one method in the ‘rendering’ category. We generally refer to these methods as being ‘on the *instance side*.’

If you click on the 'class side' checkbox you will see three method categories and four methods. These methods are 'on the *class side*' and define code that will run when a message is sent to the class 'WACounter.' Class-side methods are what other languages might describe as 'static functions' and generally support instance creation and singleton management (where the nature of the class is that there should be only one instance).

In Smalltalk, a typical beginner's programming error involves putting methods on the wrong 'side' of a class. In some cases there might even be methods with the same name (e.g., 'initialize') on the instance side and on the class side of a class. In the instructions that follow, pay special attention when a method is added to the class side so that when you are done you remember return to the instance side. Also, if things don't work, go back and see if the methods have been added properly to the correct side of a class.

Again, as I've worked with students going through this tutorial the most common error has been creating methods on the wrong 'side' of a class. If you reread the preceding paragraph you will save yourself problems later.

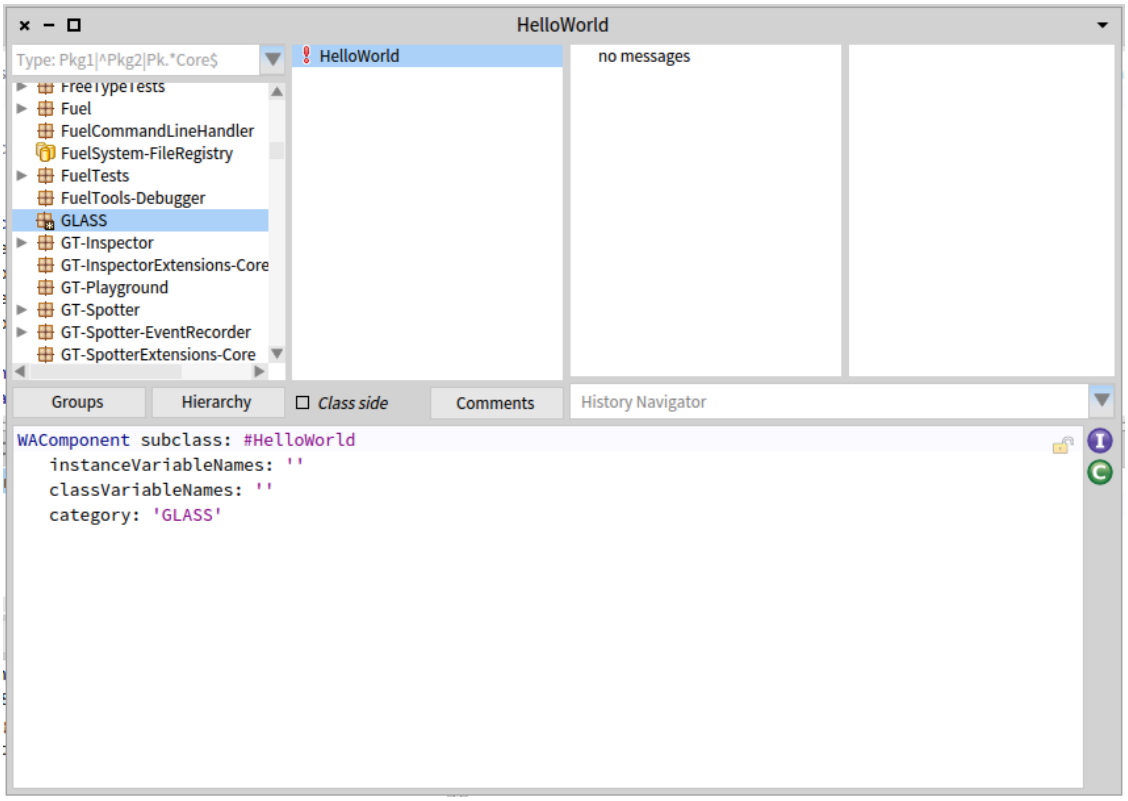
Now let's create a new class!

8. 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) and/or embedded in another component. **In the System Browser, click on any line in the first column to get a new class-creation template** (make sure that the 'class side' checkbox is unchecked)**.** In the text area replace the existing text (beginning with 'Object subclass: #NameOfSubclass') with the following and save it (<Ctrl>+<S>).

```
WAComponent subclass: #HelloWorld
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'GLASS'
```

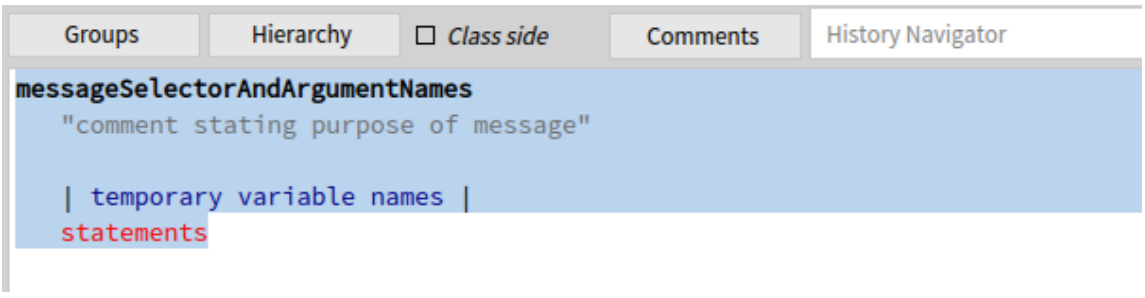
Unfortunately, copying from this document and pasting into Pharo will likely not preserve the formatting. If you want to copy/paste, you should use a simple text editor (such as Notepad on Windows, TextEdit on the Mac, or MousePad on Linux) as an intermediate paste/copy location.

Depending on your environment, you might find that <Alt>+<S> (or <Apple>+<S> on the Mac) can be used instead of <Ctrl>+<S>. This should result in a new ‘GLASS’ entry in the class category list (the first column) and the single line ‘HelloWorld’ in the second column.



The first thing to note here is that we have created a subclass of WAComponent by sending a message to the class WAComponent, not by editing a text file and then sending it through a compiler and then starting an application. *In Smalltalk we do not ‘program’ by editing text files, but by interacting with existing objects in an existing, active, object-based environment using tools that are in that environment.* If we save this modified object space as an image, and then restart from that image, the class would still exist.

9. We are now ready to give our new class some behavior. To add an instance method to our HelloWorld class, ensure that the instance side of the class is selected (uncheck ‘*Class side*’), then click in the third column (the method categories list). This will change the text area from a description of the class to a method template.

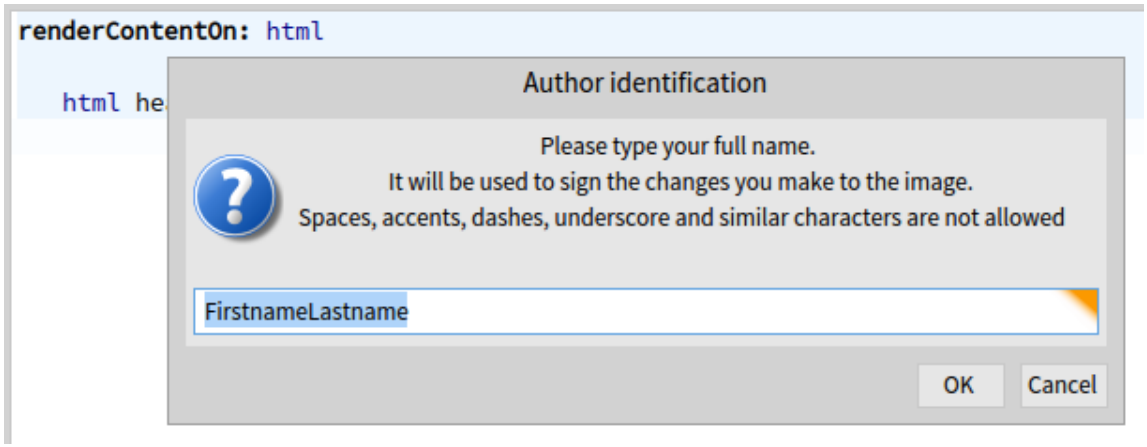


In Smalltalk, methods start with the name of the method (the message selector) and the name of any arguments. The remainder of a method consists of Smalltalk expressions that are evaluated when the method is called. Just as all expressions return an object, all methods return an object (which, of course, the caller is free to ignore). By default, the method returns the receiver, but any expression following the 'circumflex accent', usually referred to as 'up arrow' or 'caret' (^), will be returned as the expression's result. (This up arrow is one of Smalltalk's two operators. See step #15 for the assignment operator.)

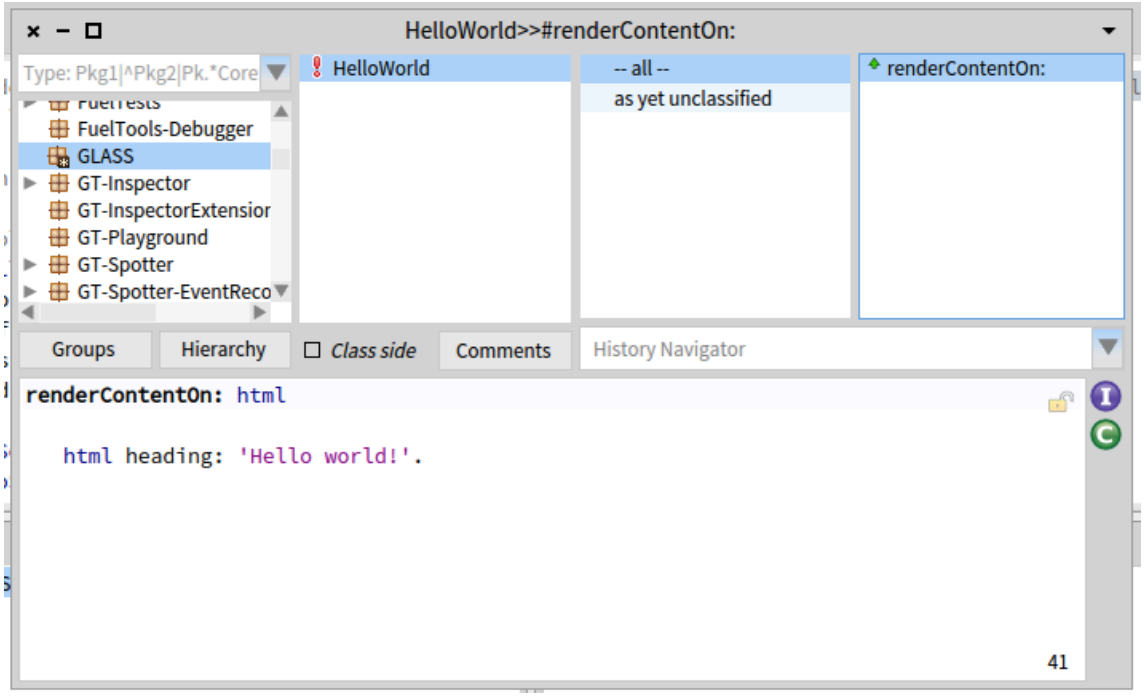
10. Perhaps the most important method in a Seaside component is 'renderContentOn:' which creates the HTML. In the spirit of the 'Hello world!' convention, replace the text in the text area with the following method.

```
renderContentOn: html

   html heading: 'Hello world!'.
```

11. When you save your first method (using <Ctrl>+<S>), Pharo asks for your name so it can keep a timestamp of who and when the method was last edited. In the following dialog box enter your name (with no punctuation characters) and press <Enter> or click the OK button:



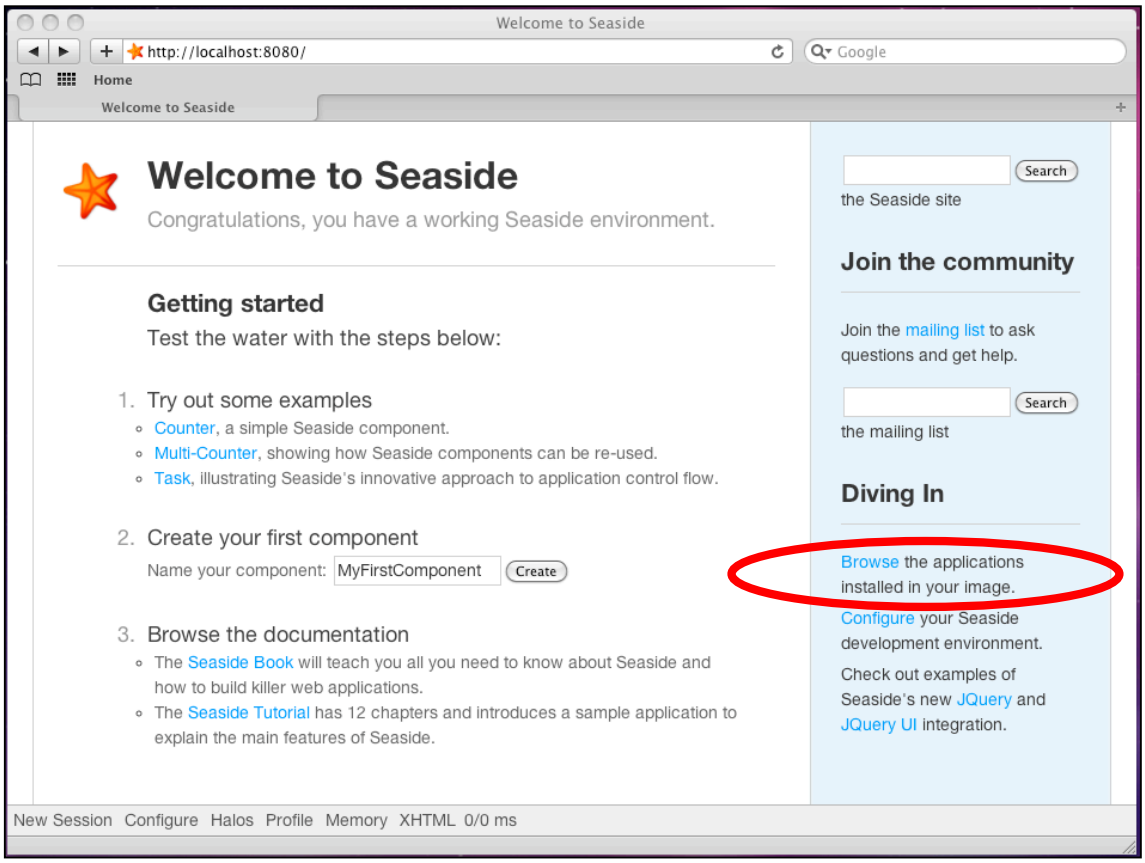
The result should be a System Browser that has a method category of ‘as yet unclassified’ and lists your method as ‘renderContentOn:’. The green arrow pointing up next to the method name in the fourth column is an indicator that you have overridden a superclass method.



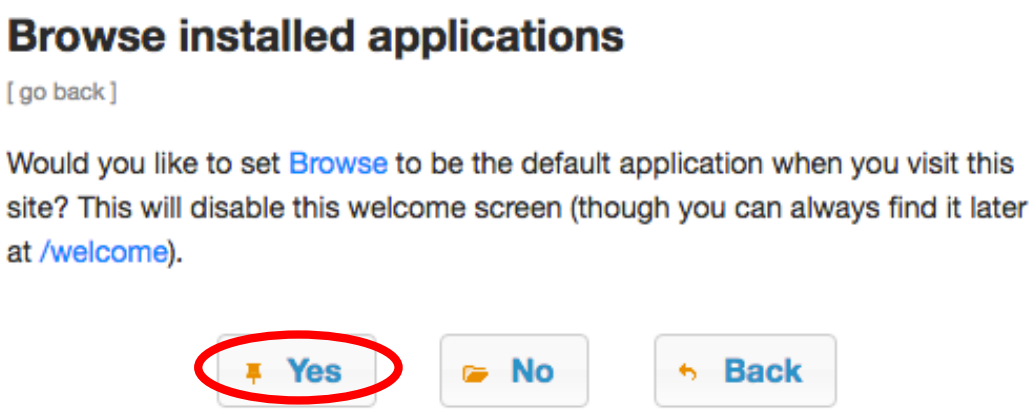
12. Now we need to inform Seaside that this new component can be used as a root component. In order to do this we need to tell the class what name to use when it registers itself as an application. Switch to the Workspace (the window titled ‘Welcome to Seaside 3.1) and evaluate the following text by clicking anywhere in the line and typing <Ctrl>+<D> (for ‘do-it’). If you have trouble, review step #3.

```
WAAdmin register: HelloWorld asApplicationAt: 'hello'.
```

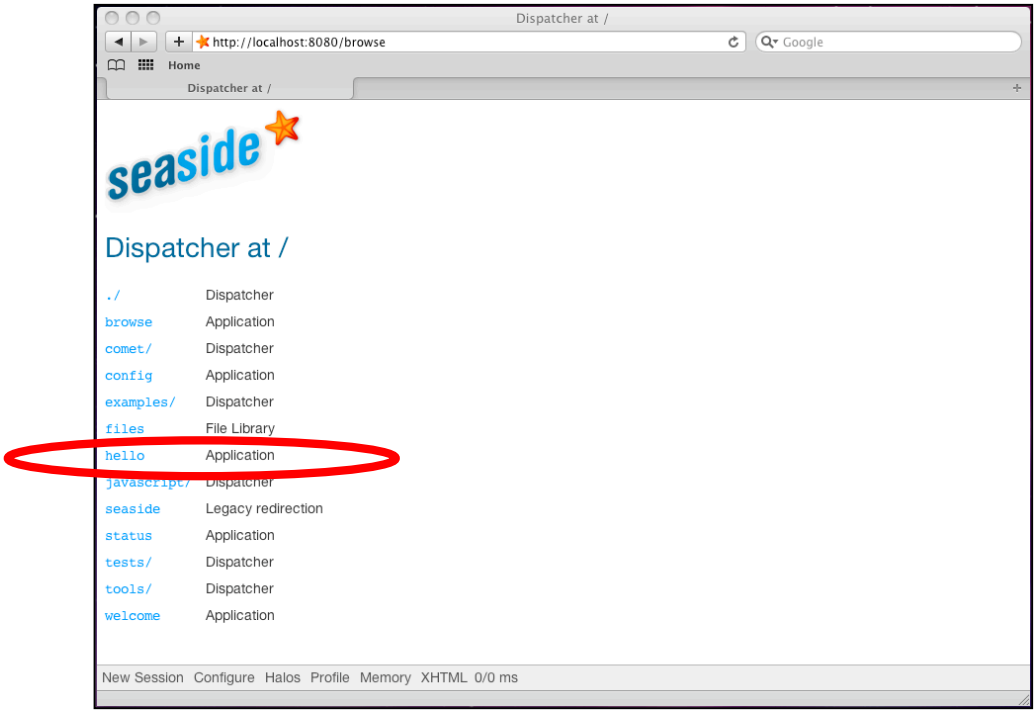

13. If the above steps were successful, you should be able to open a web browser on http://localhost:8080/ and click on the ‘Browse’ link.



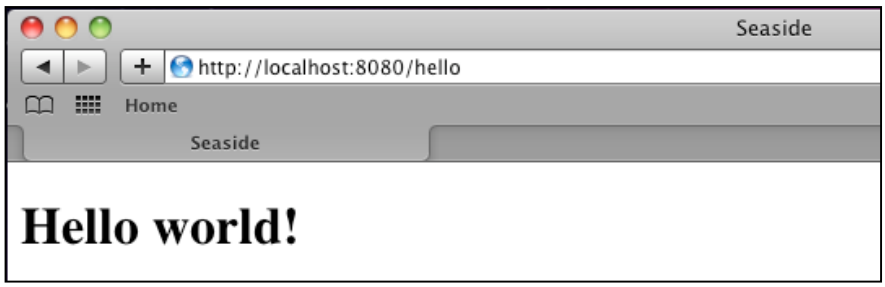
Before showing you a list of installed applications, Seaside prompts you to change your Seaside home page from the welcome page (above) to a page listing the installed applications. Click the ‘Yes’ button to change your home page.



The ‘Browse Installed Applications’ page gives you a list of the top-level links being served from by this web server, including our new ‘hello’ application.



14. Click on the ‘hello’ link and note that the expected message is displayed. While not very sophisticated, we have demonstrated that we can create a *Web Application* by adding one method to a subclass of WAComponent. (Can you guess what the “WA” in WAComponent stands for?)



15. In order to make the application a bit more dynamic, modify the 'renderContentOn:' method as follows (the changed lines have been highlighted). Note that the line numbers are in comments (the double quote character delineates a comment), so do not need to be entered.

```
renderContentOn: html

"3"      | now |
"4"      now := DateAndTime now.
"5"      html heading: 'Hello world!'.
"6"      html heading
"7"         level3;
"8"         with: now;
"9"         yourself.
```

This code introduces some new syntax. Line 3 defines a method temporary variable, 'now,' that is declared by placing it between vertical bars (or pipe characters). In Smalltalk (like Perl, Python, PHP, and others), variables are dynamically typed (as opposed to the static typing of languages like C and Java where types must be declared at compile time). This means that all we need to do is specify the name and this will reserve space for the object reference.

Line 4 is an expression that introduces the second of Smalltalk's two operators (the return operator was mentioned in step #9). The assignment operator, colon-equals (:=), is used to take the object returned by the expression on the right and place a reference to it in the variable defined on the left.

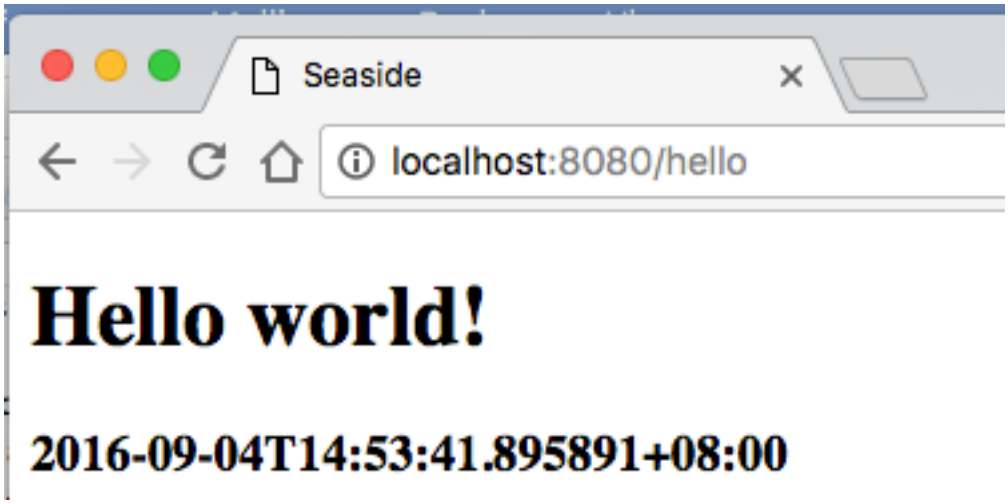
Line 5 remains from the original method and shows the greeting.

Lines 6-9 are a single expression. Since extra whitespace is ignored in Smalltalk you can add new lines and tabs for cosmetic styling. Next, recall that expressions are made up of receivers and messages and that there are three types of messages: unary, binary, and keyword (evaluated in that order). Before it was edited, the above method had only one message, the keyword message 'heading:' that was sent to 'html' with a String as an argument. The modified method has the same first message, but after it is a more complex expression. The expression starts with a unary message ('heading') that returns an instance of WAHeadingTag. This heading object is sent three messages, 'level3' (to set the level), 'with:' (to set the text contents), and 'yourself'.

This semicolon indicates a 'message cascade,' meaning that that what follows is not a full expression (which would require an object reference to designate the receiver), but another message to the receiver of the most recent message. The receiver of the last message was the object returned from the 'heading' message send (an instance of WAHeadingTag), so the next message will be sent to the same object.

Finally, we have the ‘yourself’ message that is sent to the receiver of the most recent message. The ‘yourself’ message calls a method that simply returns the receiver. In our case, since we ignore the returned object, this message send is nothing more than (a slightly inefficient) cosmetic filler so that the previous line can end with a semicolon rather than a period (which would probably be the more common approach by most Smalltalkers). This programming style allows you to come back later and add another line or rearrange the lines without having to change the line ending character.

16. Now return to your web browser and refresh a few times and watch the time change.



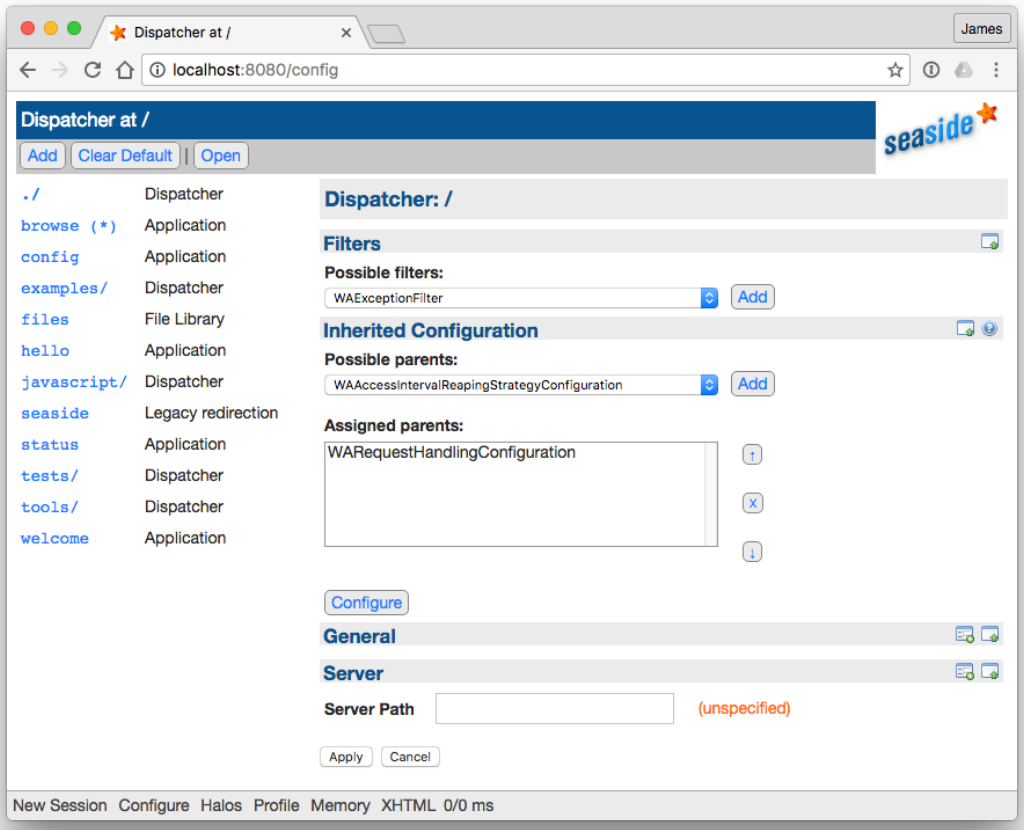
17. At this point you can return to Pharo and save the image and quit (as described in Chapter 1).

In this chapter we will explore some of the Seaside from the web browser's point of view, including examples available with the default installation of Seaside.

1. Launch the Seaside One-Click Experience (see Chapter 1 for details) and open a web browser on http://localhost:8080/browse.

   This page gives a list of the applications that have been registered with Seaside through the WADispatcher singleton (available by evaluating the Smalltalk expression 'WADispatcher default'). The dispatcher looks at the URL requested, and dispatches the request to one of these registered entry points. This is how the request from the client browser gets to the appropriate Smalltalk code (such as our 'HelloWorld' application).

2. Click on the config link to see the 'Dispatcher Editor.' This tool sets up the information for the Dispatcher Viewer we saw above.

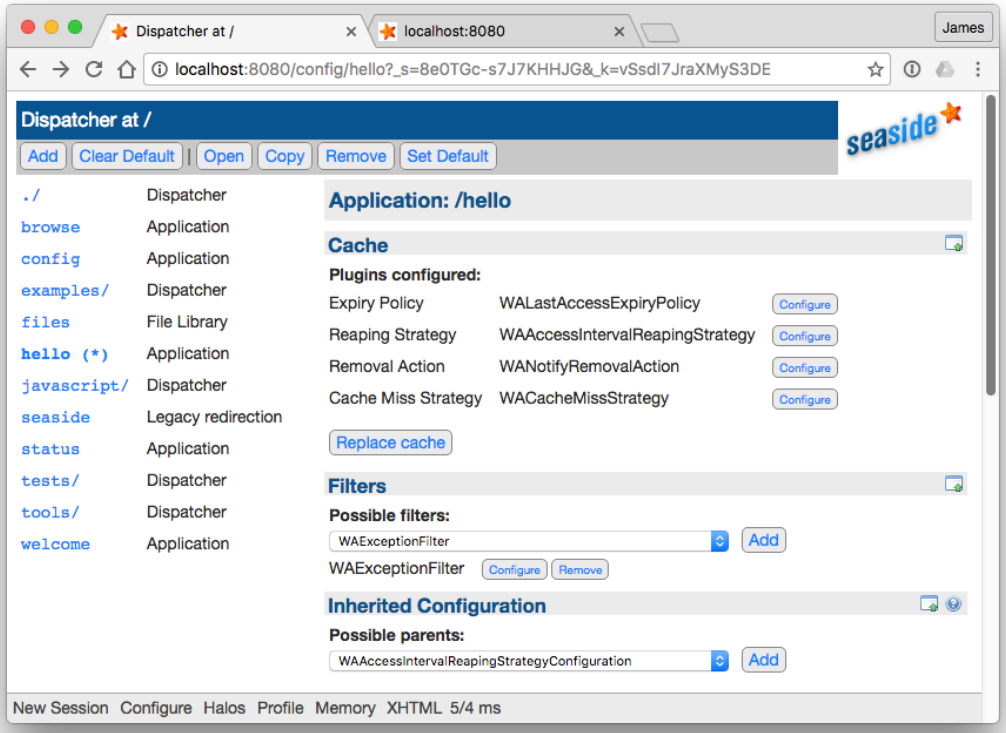


The left column lists each of the entry points and the initial selection is the root, which is just a dispatcher to some other web component. There are various types of components, including other dispatchers (examples/, javascript/, tests/, and tools/), applications (browse, config, hello, status, and welcome), a file library (files), and a legacy redirection (seaside). Selecting any component in the left column gives you some configuration information about that component.

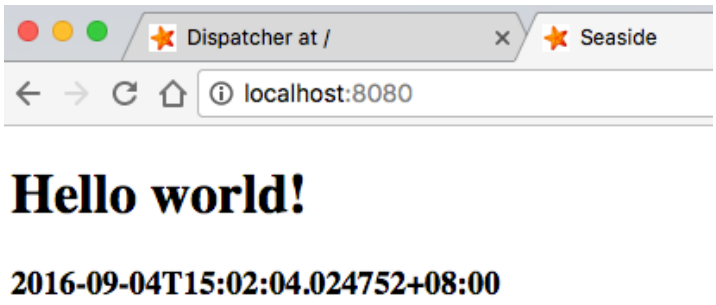
Note that there is a star next to the *default* component. The default component is the one provided by Seaside if you select the root (e.g., http://localhost:8080/). In chapter 2, step 13, we clicked 'Yes' when Seaside offered to change the default application from 'welcome' to 'browse'.

Note that near the top of the config page there is a ‘Clear Default’ button. Click the button then open a new web browser (window or tab) and navigate to the root. You should get an error.

Switch back to the config page (at http://localhost:8080/config if you did not save that page), select ‘hello’ from the links on the left, and then click the ‘Set Default’ button. This will make our Hello World application the default Seaside application.

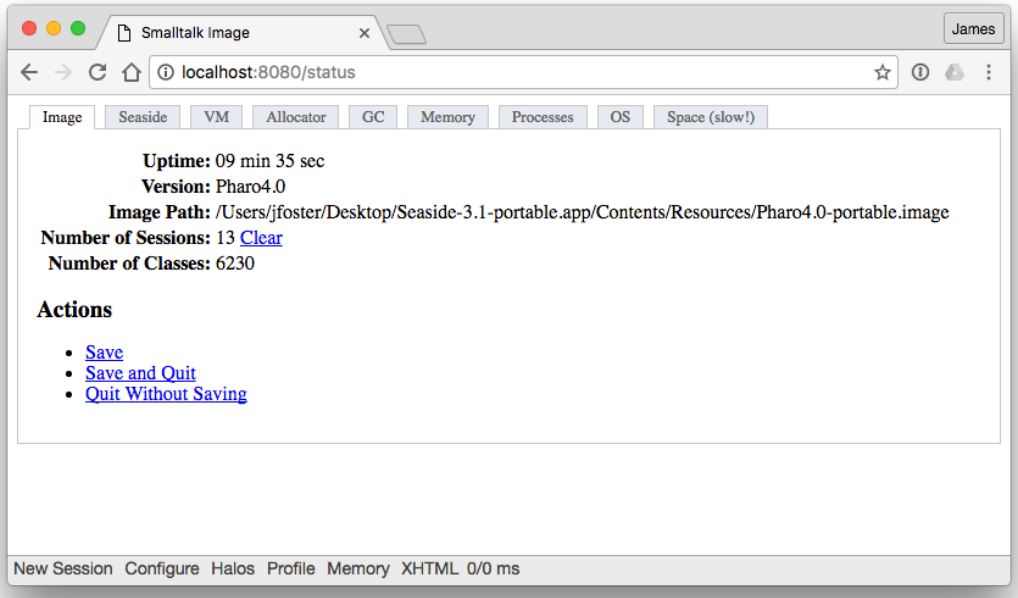


In your web browser, switch back to the tab or window with the error and refresh (or enter http://localhost:8080/). You should now see our Hello World application at the root.

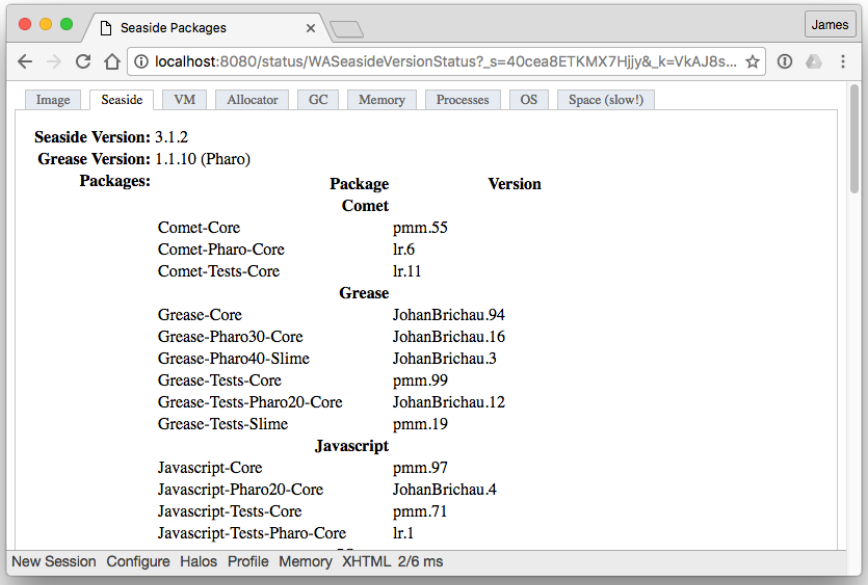


The config page provides a great deal of other configuration capabilities and demonstrates some of the capabilities of a Seaside component. We will not examine this area further at this time.

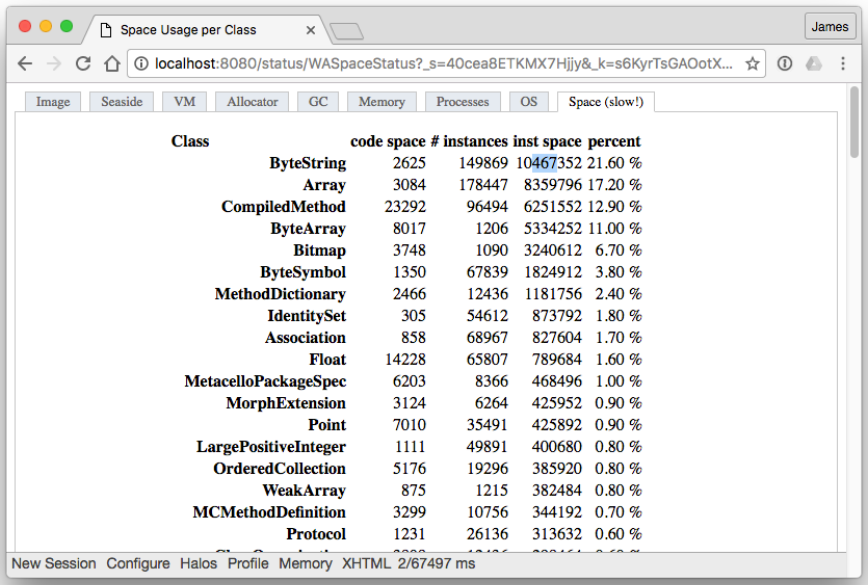
3. Enter http://localhost:8080/browse to return to the list of components and select 'Status' (or enter http://localhost:8080/status in your web browser's address bar). This will show you a list of tabs containing information about your Smalltalk environment. The first tab identifies the location of the file containing a snapshot (image) of the object space when it was last saved to disk. It also gives you the opportunity to make another snapshot of the current object space (by saving the image).



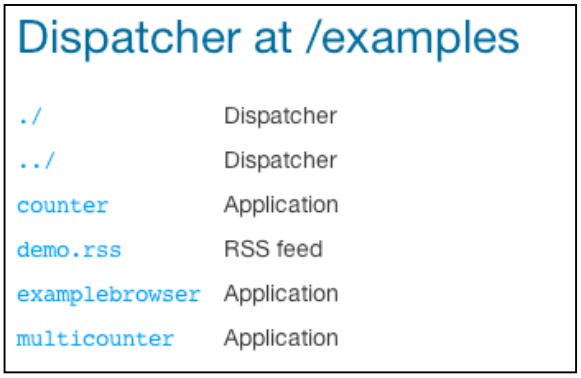
The ‘Seaside’ tab identifies the version of Seaside installed, and lists the version of various installed packages.



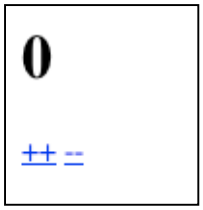
The final tab scans the entire object space and generates a report of objects in your object space. In this example, it took over 60 seconds to determine that there are almost 150 thousand strings in my environment, taking up about 10 MB of space.



4. Return to http://localhost:8080/browse and select 'examples' (a directory of other entry points). You should see the following:

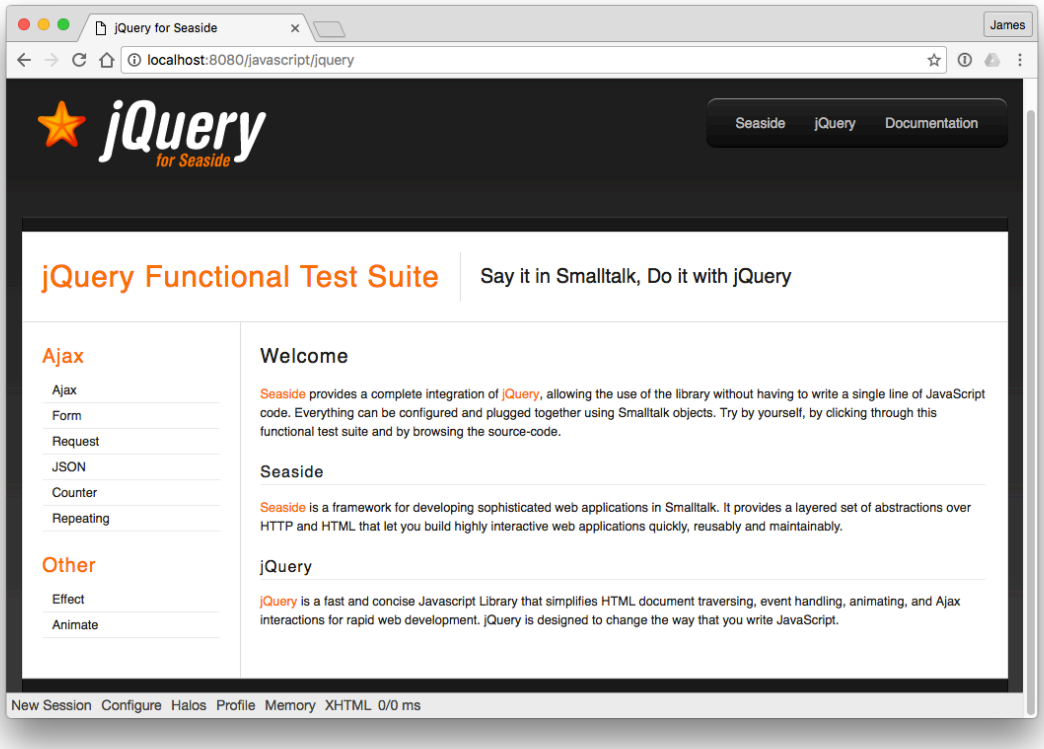


Click on the 'counter' and you will see the traditional Seaside example of an application that keeps state on the server. We will be using a slightly more complex example to explore some Seaside basics.



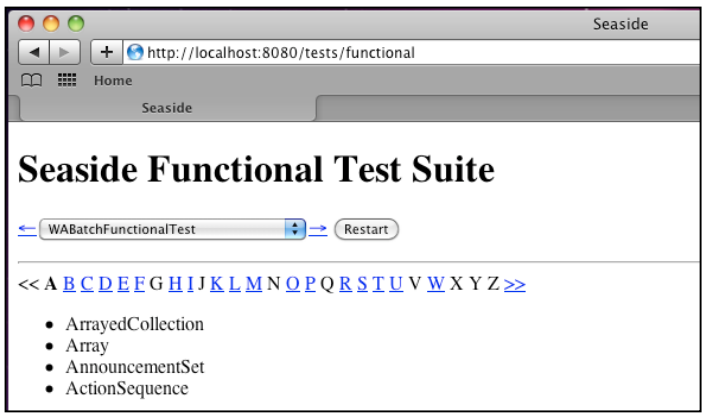
5. Use the web browser's back button to return to the examples page and then select the 'javascript/' link. Here you can find demos (with code) showing usage of jQuery and Scriptaculous.

Select the 'jquery' link to see a page describing Seaside support for JavaScript.

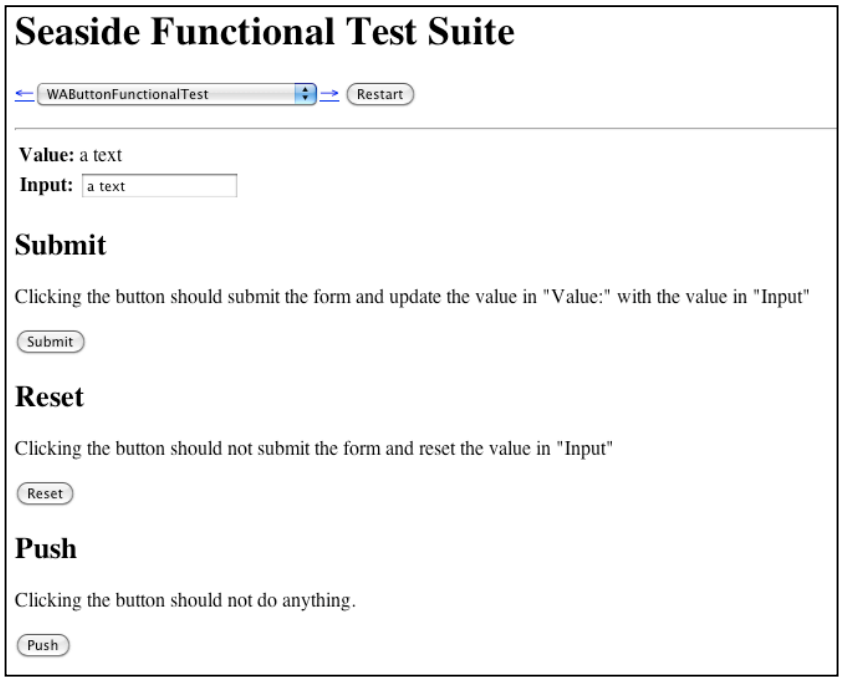


Select the 'Ajax' link in the left column. This gives a demonstration of how clicking on a button sends an Ajax request to Seaside to execute Smalltalk code on the server and send back a response (the current date and time).

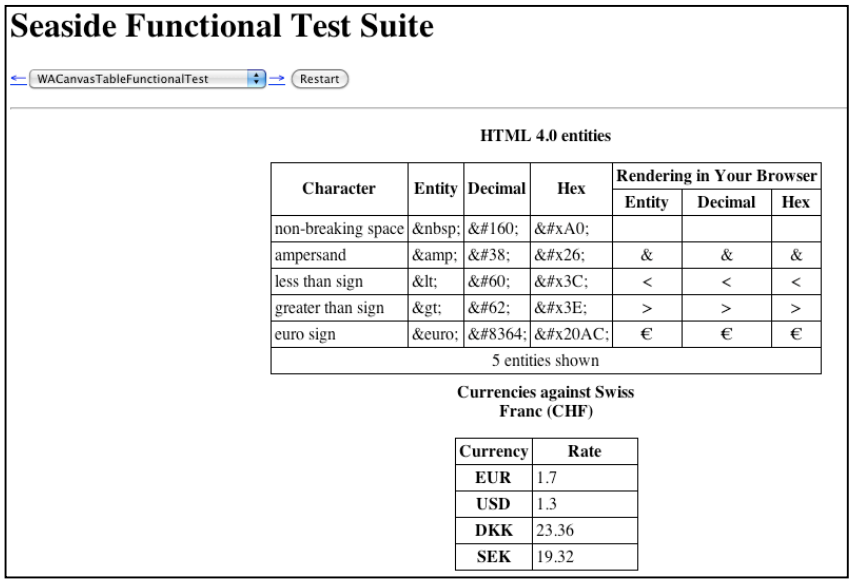
6. Return to http://localhost:8080/browse and select 'tests' and then 'functional.' The Functional Seaside Test Suite shows a drop-down list of various tests, with the test selected. The WABatchFunctionalTest encapsulates a list of class names and shows a horizontal list of the letters of the alphabet that can be used to jump to a particular place in the list. You can also use the previous ('<-') and next ('->') links to move through the list.



7. To reach the second test, click on the drop-down list showing 'WABatchFunctionalTest' and select the second item ('WAButtonFunctionalTest'). This will demonstrate the difference between various HTML button types ('submit', 'reset', and 'button'). If you enter text in the input field and click the Submit button, the new value will be returned to the server. If you enter text in the input field and click the reset button, the old value will be restored.

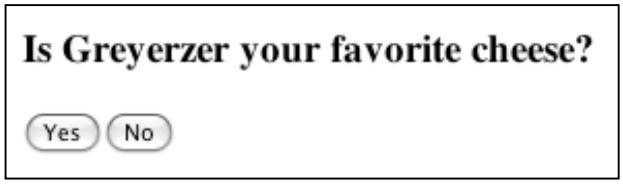


8. The WACanvasTableFunctionalTest demonstrates Seaside's ability to generate various table-related HTML elements, including <table>, <caption>, <colgroup>, <thead>, <tfoot>, <tbody>, <tr>, <th>, and <td>. This test gives you an example to use if you want to build a table in Seaside.

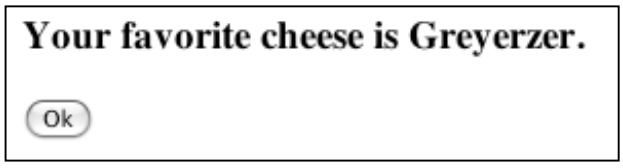


9. The WAFlowConvenienceFunctionalTest demonstrates three simple built-in components. The first component is the 'WAChoiceDialog' in which Seaside presents a list of options, add a prompt message, and then will return one of the items in the list of options or 'nil' if the user clicks 'Cancel'. (The object 'nil' is the single instance of UndefinedObject, an object that is the something that represents nothing in Smalltalk.) Any subclass of WAComponent (like our 'HelloWorld' class) can send 'chooseFrom:caption:' to self providing a list and a string and get back an answer.

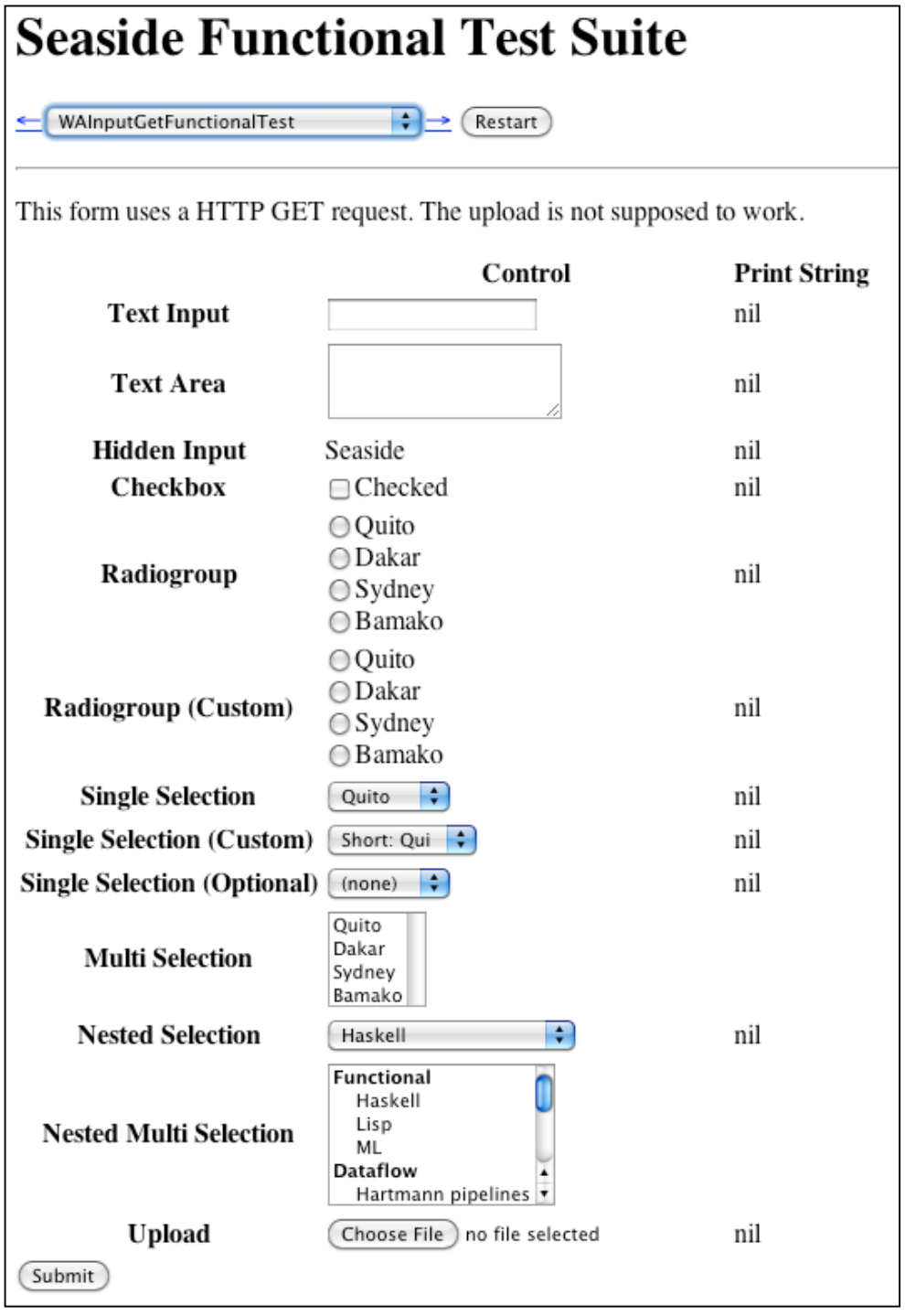
If you click 'Ok' on the previous component, Seaside immediately presents the 'WAYesOrNoDialog' in which a message is presented with two buttons, 'Yes' and 'No' that will be returned as 'true' or 'false' Booleans. Any subclass of WAComponent can send 'confirm:' to self with a string and get back a Boolean.



If you clicking on 'Yes' in the previous component, Seaside immediately presents 'WAFormDialog' in which a message is presented with a simple 'Ok' button. Any subclass of WAComponent can send 'inform:' to self with a string to create this page.



10. The WAInputGetFunctionalTest demonstrates a variety of HTML input tags. By reviewing these examples you can get ideas of what can be done and then go to the test class to look at sample code. (As you progress through this tutorial you will learn more about finding classes in Pharo. In this example you would right-click on the first column of a System Browser, select the 'Find Class' menu item, enter WAInputTest' when prompted for a class name fragment, and then select 'WAInputTest' from the list.)



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.
```


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.)

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.

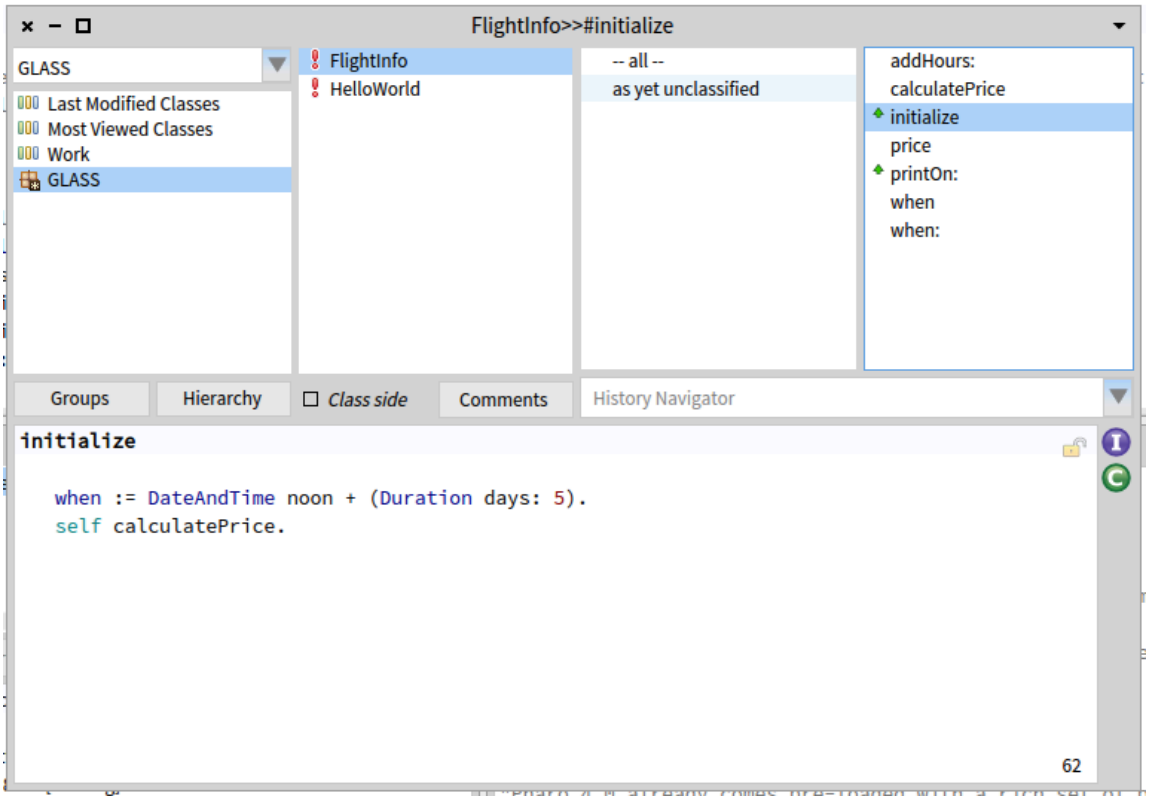
```
printOn: aStream

   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.

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.
```


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.

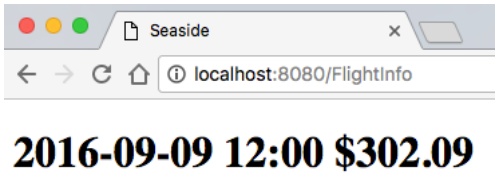
```
initialize

   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.

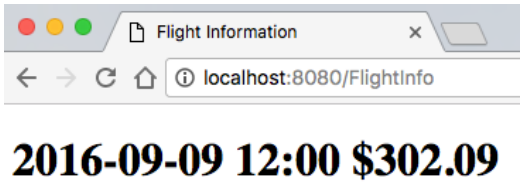


2016-09-09 12:00 $302.09

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.



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);
```

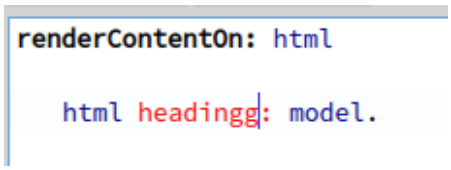
Seaside is a web framework in which the HTML is generated using regular Smalltalk programming. The power of Smalltalk comes from its ability to represent complex domain models as a rich interaction of simpler objects.

In this chapter we use the Flight Information application to learn how to use a debugger and see how Smalltalk provides uniquely powerful debugging opportunities. Then we will explore how code can be edited from a web browser and discuss code blocks, a powerful feature of Smalltalk.

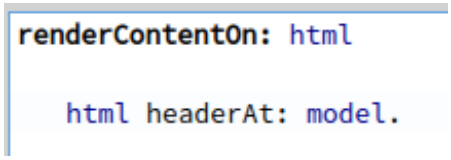
1. Start Pharo if it is not running and in the System Browser select the FlightInfoComponent and the 'renderContentOn:' method. We will modify this method in such a way as to introduce an error into the code.

```
renderContentOn: html

   html headingg: model.
```

This changes the message sent from 'heading:' to 'headingg:' and represents a typical 'typo' that could easily be made. Note that the message is in red type, indicating that there is no object in the entire object space that understands this message.

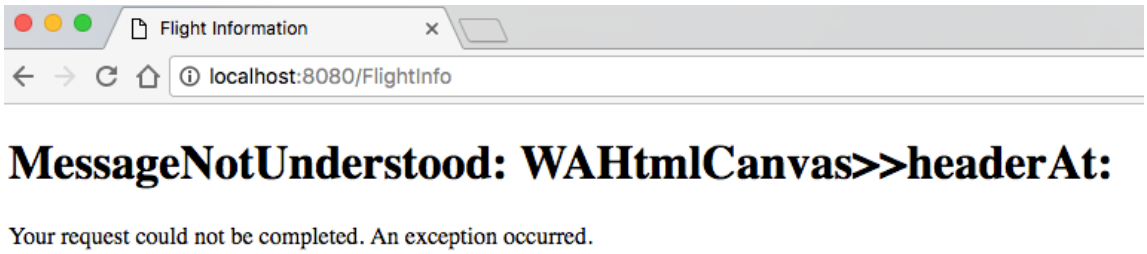


Recognizing that the method is wrong, we can change it to something else, 'headerAt:', that is recognized, but still wrong.

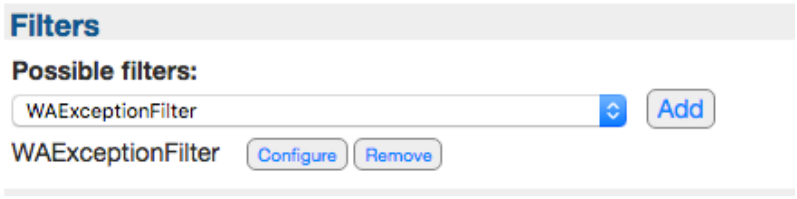


This demonstrates a problem with dynamically typed languages (such as Smalltalk). Because a variable can hold a reference to any object, we can't tell at compile time whether the receiver will understand the message we send to it. One of the advantages of a statically typed language is that this sort of error is substantially avoided. Suffice it to say that this is the topic of many language wars, and we'll let it go with the statement that Smalltalkers are almost universally happy with the flexibility offered by the dynamic environment. Of course, this means that we occasionally have to fix problems where we sent the wrong message, so here goes!

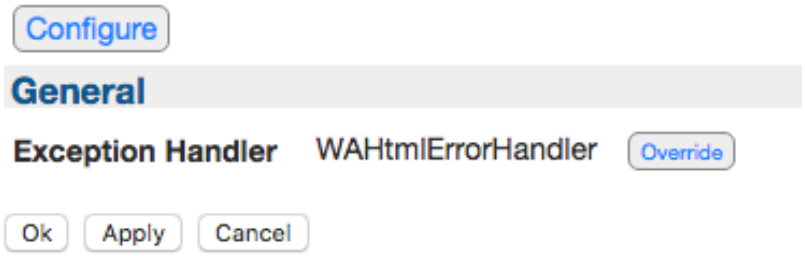
2. Return to your web browser and navigate to the 'FlightInfo' component (click the <Refresh> button if you are already there). Seaside returns an error.



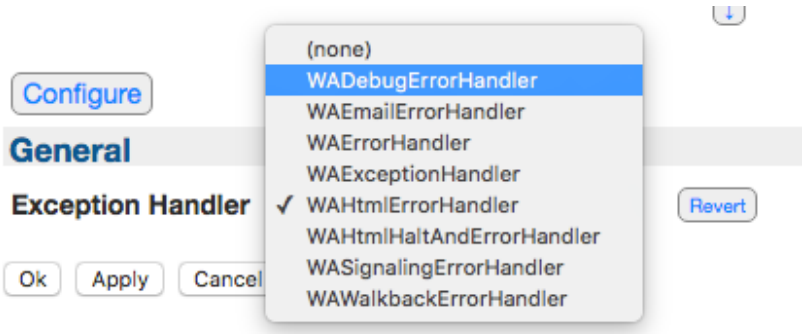
This is informative but not very helpful for debugging. In a new tab, open http://localhost:8080/config/FlightInfo and click the 'Configure' button next to 'WAExceptionFilter.'



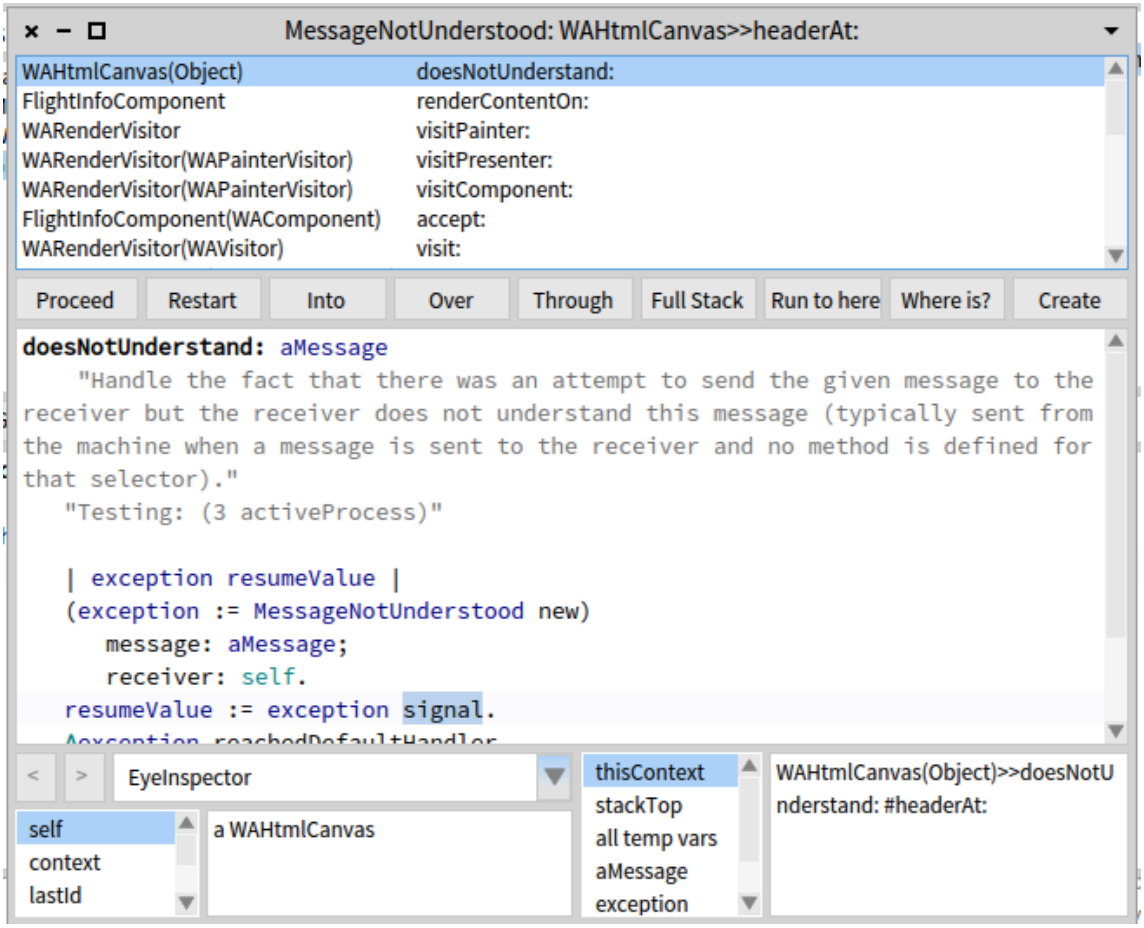
Next, click the 'Override' button next to 'WAHtmlErrorHandler'.



From the drop-down list of Exception Handlers, select 'WADebugErrorHandler' and then click the 'Apply' button.

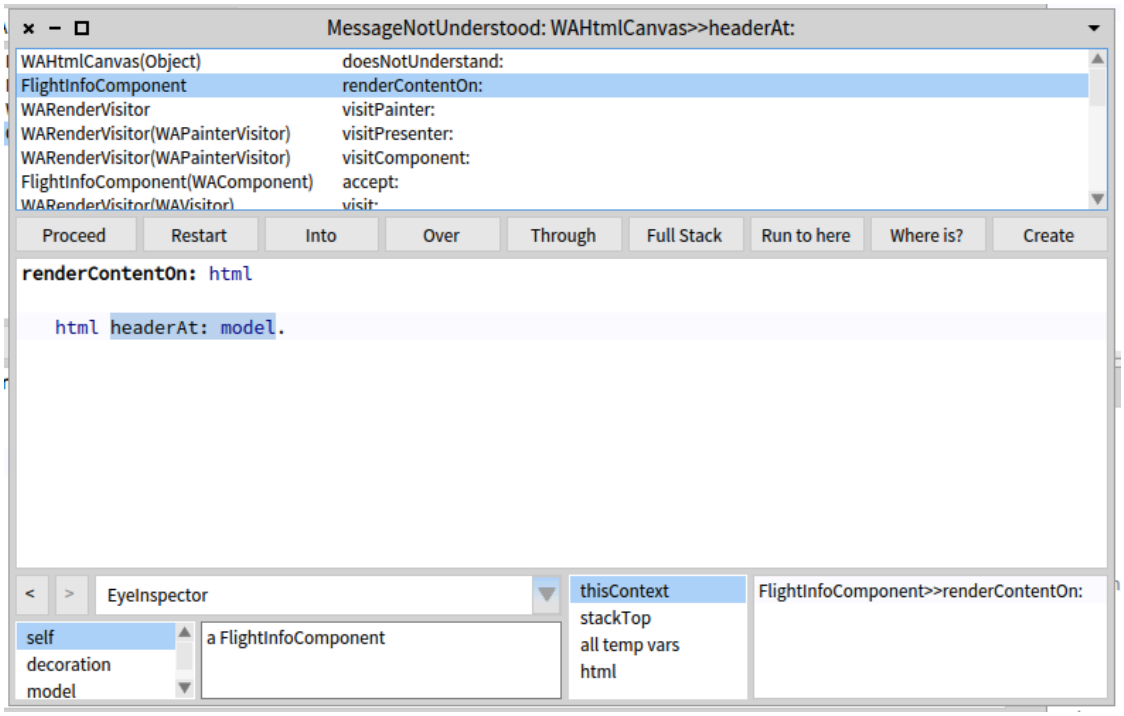


Now return to your web browser tab with the Flight Info error and click refresh. Note that the web browser just hangs. This is because Seaside got an error and has not returned any HTML to the client and it gives us an opportunity to look at Smalltalk's debugging capabilities. Switch back to Pharo and notice that a new window exists with the title 'MessageNotUnderstood'.



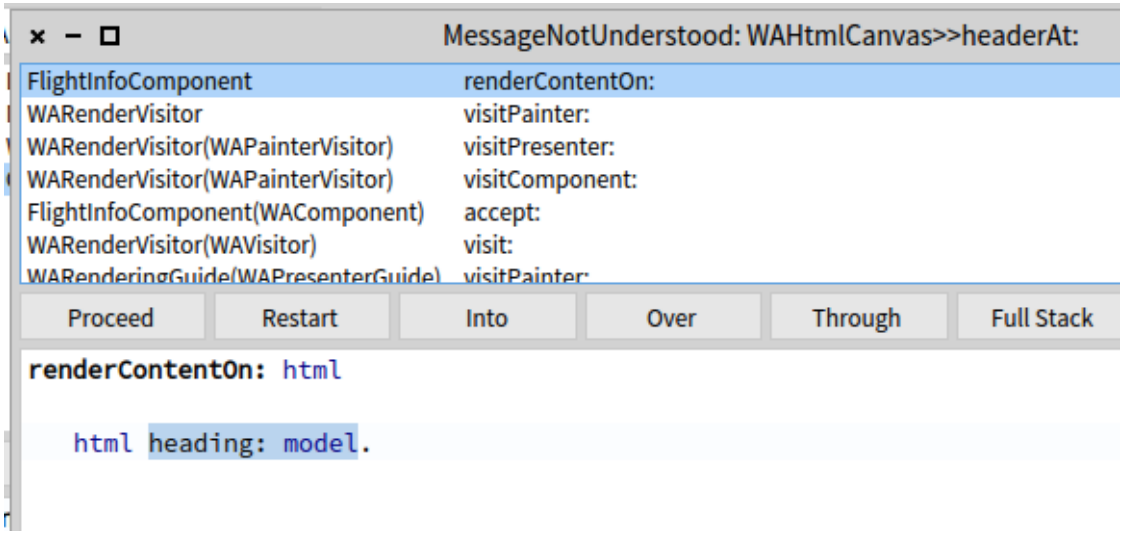
3. In the top third of this debugger window is a scrolling list of class and method names representing frames in the process stack. In the middle third we have a row of buttons followed by a text area. The buttons control the debugger and the text area shows us a method in a selected stack frame. The bottom third gives us information on the receiver and its instance variables, and the context and its temporary variables.

Click the second line in the process stack: ‘FlightInfoComponent>>renderContentOn:’.



4. At this point we can see the method is sending the message 'headerAt:' which is not understood. We should have sent the message 'heading:' to get the proper behavior.

   If this were a typical programming language, we would need to edit a text file containing the source, then recompile the code, restart the application, and apply it to the web server. Smalltalk gives us a much more powerful approach to fixing bugs. In the debugger, edit the method so that 'headerAt:' is replaced with 'heading:' and save the method (<Ctrl>+<S>). When you save the method the debugger trims the stack to this method (removing the frames above) and starts over at the beginning of this method.

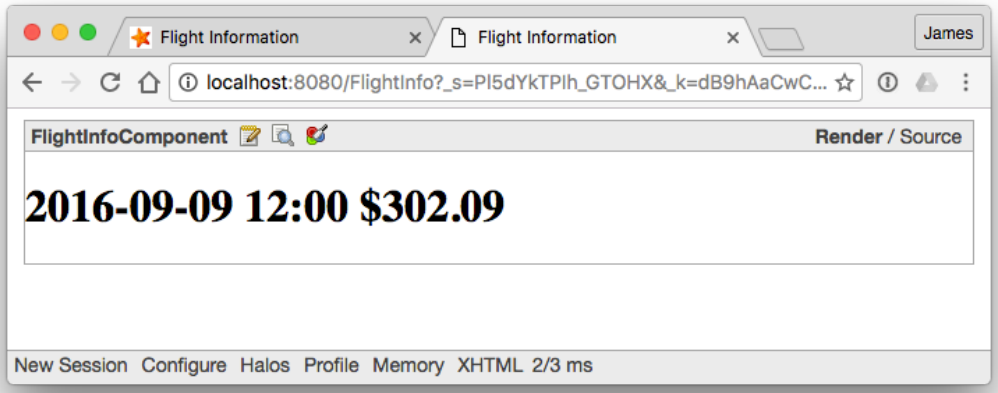


   You can click the 'Into' and 'Over' and 'Through' buttons a few times to watch the code being executed. You can also click on some of the lists at the bottom to see the values of various variables.

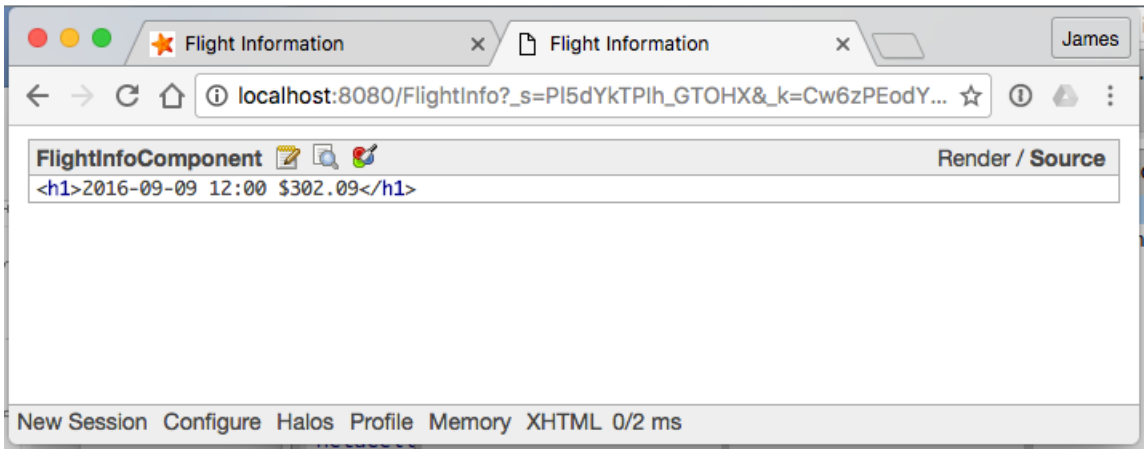
5. Once you have played a bit with the debugger, click the 'Proceed' button (the left-most of the buttons). The debugger window should close and then you can return to your web browser. The web browser should now show a page with a date, time, and dollar amount. This shows that we have fixed the problem introduced above in step #1.

   This is an example of Smalltalk's ability to do surgery on a sick patient when other environments would only give you an opportunity to do an autopsy on a dead patient.

6. So far we have been using Pharo's System Browser to edit code. Seaside provides an alternate method of editing code—using the web browser. To do this we first need to turn on some Seaside tools using the 'Halos' link at the bottom of the page. While we could do this in the current web browser's window, it will be helpful for what follows to keep the existing page and have a second page or tab open on the tools. How you do this will depend on your web browser, but most modern browsers support opening a link in a new tab (generally with a right-click context-sensitive menu). Following shows the result of opening 'Halos' in a new tab (click 'Halos' again if the box does not appear).



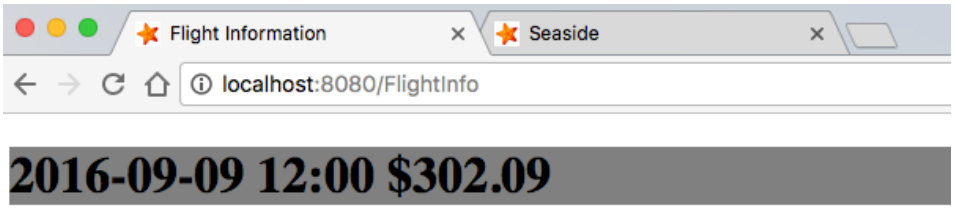
What Seaside has done here is add a box around our component that provides some information and tools (this is the 'halo'). The halo shows the component class name (FlightInfoComponent), three icons (that bring up a Class Browser, an Object Editor, and a CSS Style Editor), and links for Render and for Source). We will examine each of these.
Note that the 'Render' at the top right is bold. This lets us know that Seaside is 'rendering' the generated HTML. Click on the 'Source' to show the Source HTML. We can see that the 'heading:' message causes Seaside to render the model inside an <h1> element.



Click on the 'Render' to return to 'render' mode.

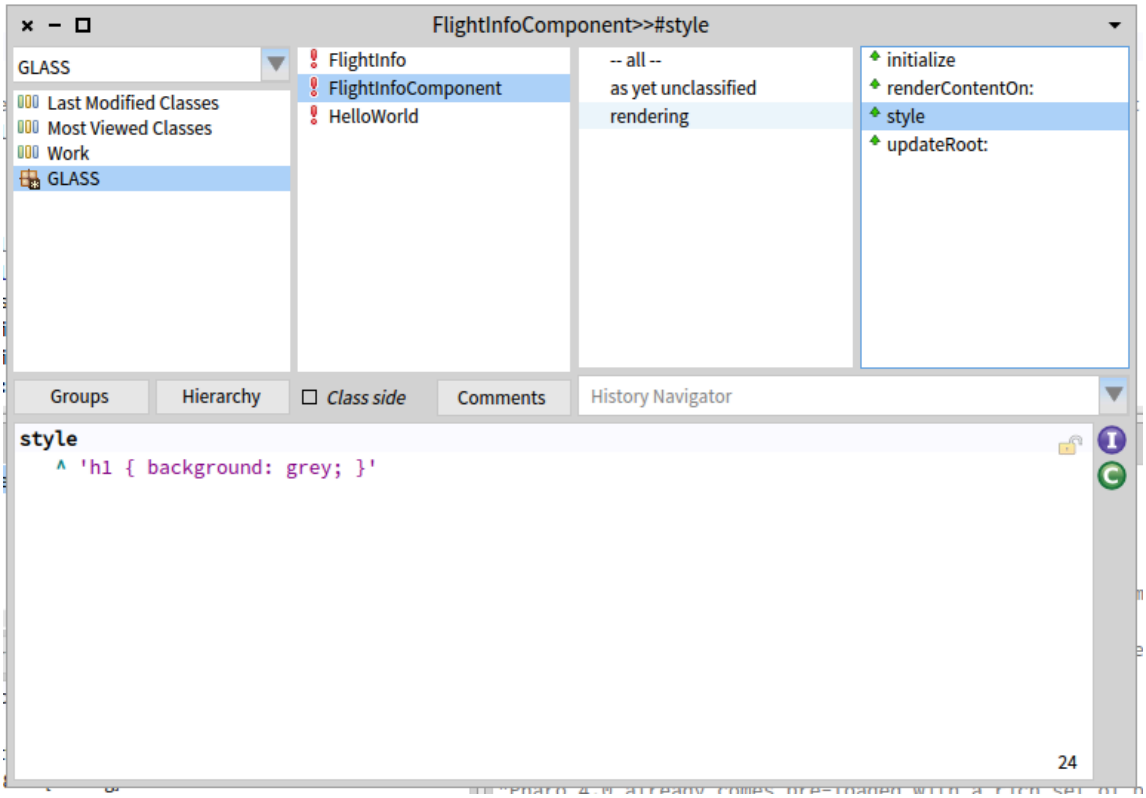
7. Next, click on the third icon that shows a tool over three colored circles. This changes the page to a CSS Style editor. Enter 'h1 { background: grey; }' in the text area and click the 'Save' button at the bottom (not shown here).

8. Now switch to the page (tab or window) with the original application and refresh. You should see a new background on the <h1> element.



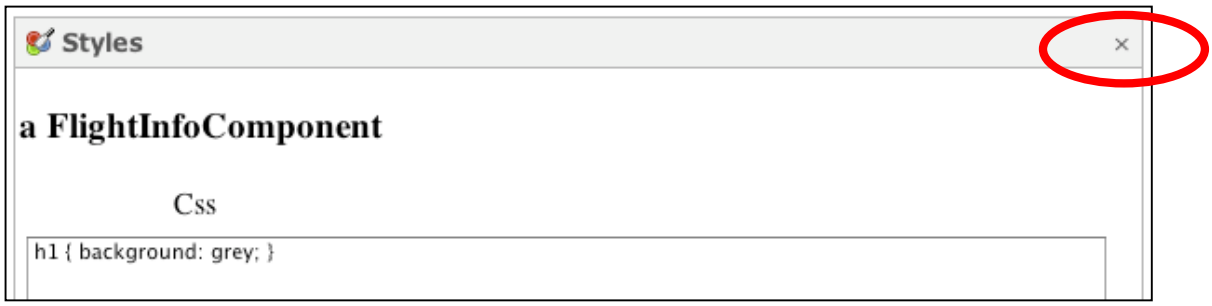
Depending on your environment, the grey might not show up well. If you don't see any change, go back to step #7 and try 'red' or 'yellow'.

9. Return to Pharo and click on 'FlightInfoComponent' in the System Browser, then on 'style' in the method list. You can see that Seaside has added a method to your component that provides CSS information.



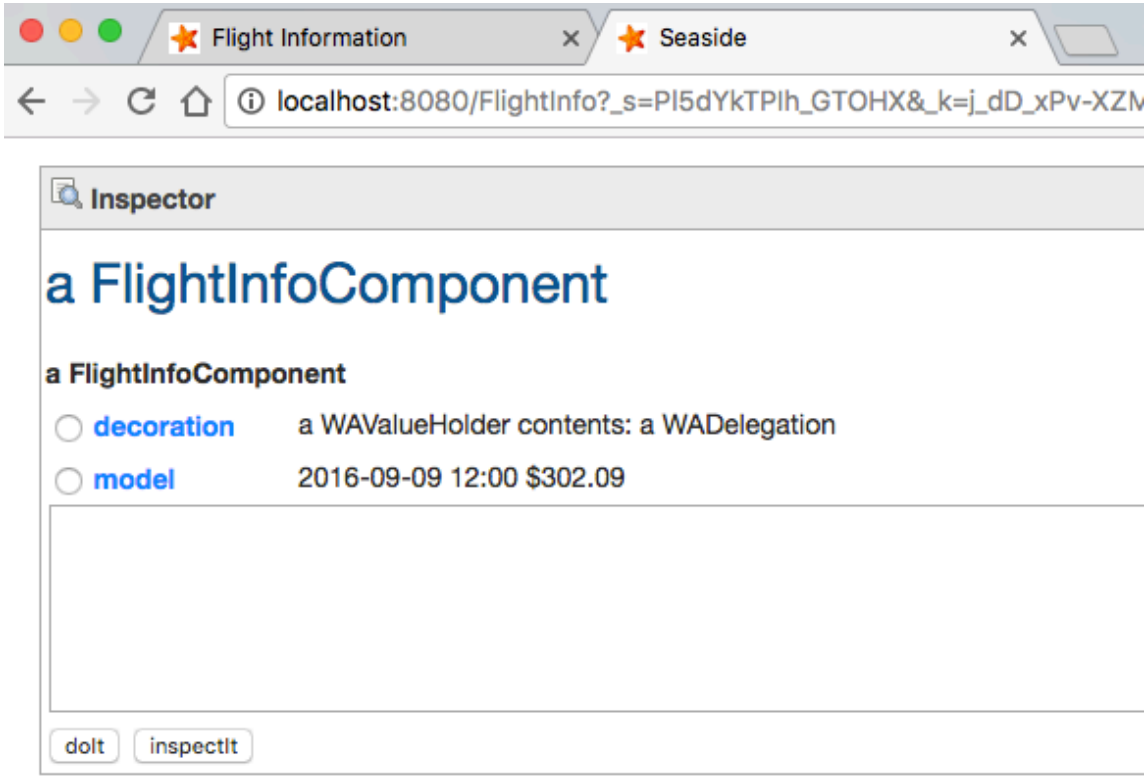
10. To reduce future confusion, edit this method so that the background for h1 is 'white'.

11. The next tool to explore is the Object Inspector. Return to the CSS Style Editor page in your web browser, and click the 'X' in the top right of the page.

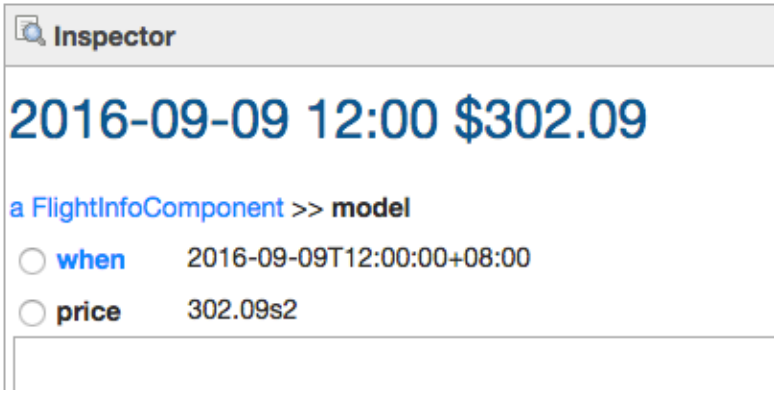


This should return you to the page shown above in step #6.

12. In the halos, the middle icon is a magnifying glass over a page. Click on that icon to open an Object Inspector.



13. In this Inspector you are first shown an inspector on the instance of FlightInfoComponent being presented by Seaside. Recall that we defined the class with one instance variable, 'model.' What we didn't notice is that one of our superclasses defined another instance variable, 'decoration.' Click on the 'model' link to inspect the instance of FlightInfo and see its instance variables.



14. In web browser, in the text area below price, type the following and click the 'do it' button.

```
self addHours: 2.
```

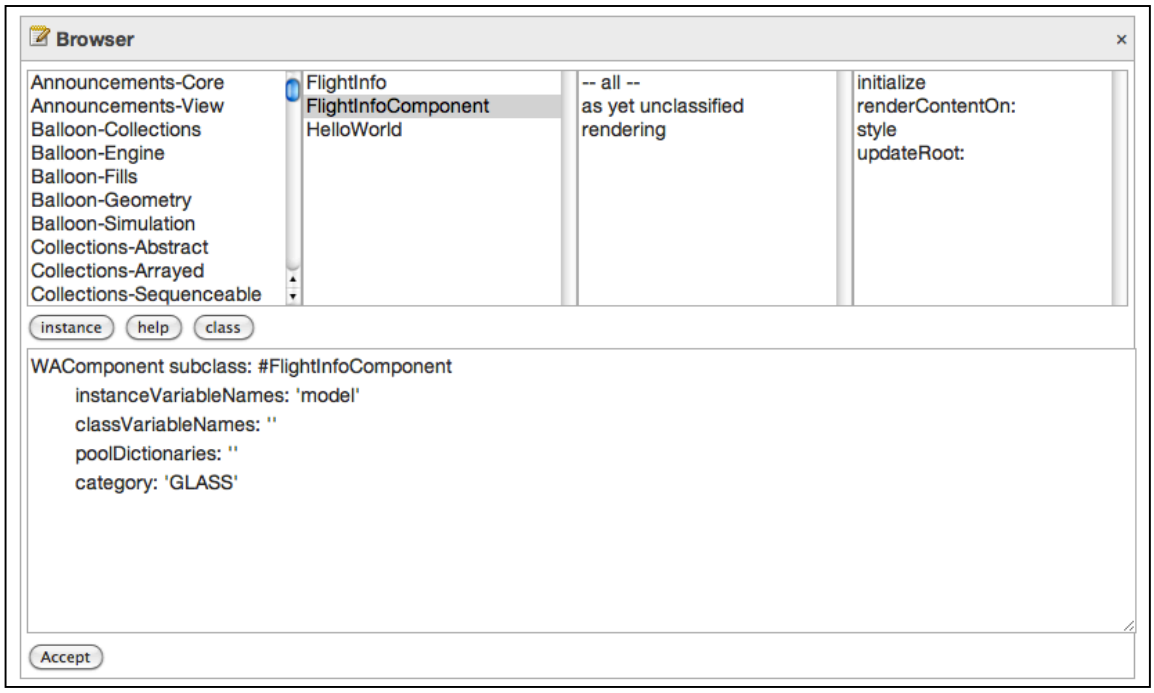
Notice how this changes the time (and the price, since the 'addHours:' method calls 'calculatePrice').

15. Again, in the text area, type the following and click on the 'inspect it' button. This will show you an inspector on 4.0, an instance of Float. From this we can see that we essentially have a Workspace available through the web.

```
16 sqrt.
```

When you are done inspecting objects, click the 'X' in the top right to close the Object Inspector. This should return you to the halos shown in step #6.

16. The final tool available from the halos is a Class Browser. Click on the first icon showing a spiral bound notepad with a pencil over it. This will open a Class Browser that should look a lot like the System Browser in Pharo. ~~We can use this browser (as an alternative to Pharo's System Browser) to add a few methods to our application component.~~



17. At this point our application simply initializes a model and displays it. By itself, this isn't a very sophisticated application and doesn't demonstrate much HTML functionality. The simplest addition that actually interacts with the data is a link. We will modify the render method to add an anchor, give it some code to execute when the link is clicked, and give it some text to display. In ~~the web browser~~ *Pharo*, click on 'renderContentOn:' in the last column and edit the text to match the following and click the 'Accept' button.

```
renderContentOn: html

   html heading: model.
   html anchor
      callback: [model addHours: -30];
      with: '<- Earlier'.
```


The added lines send one unary message ('anchor') and two keyword messages ('callback:' and 'with:'). As discussed in more detail below (at step #18), the 'addHours:' message will not be sent when this method is called. We have seen a message cascade before (in chapter 3), so we know that the receiver of the 'with:' message is the same as the receiver of the 'callback:' message. So which object is the receiver of the 'callback:' message? Since unary messages take precedence over keyword messages, the 'anchor' message will be sent to the passed-in html, and an object will be returned. The object is an instance of WAAnchorTag, a Seaside class that represents the <a> element in an HTML document.

The use of the cascade syntax here is important. We want to create one anchor and send it two messages. If we had written the code without the cascade, we might have been tempted to do the following:

```
html heading: model.
html anchor callback: [model addHours: -30].
html anchor with: '<- Earlier'.  "WRONG!"
```

This code would have created two anchors and sent the 'callback:' message to one and the 'with:' message to the other. This is not what we want! The correct way to do this (without using the cascade) would be to use a temporary variable, but it is more complex than the cascade approach. This, then, is an example of where the cascade makes the code simpler and more readable.

```
| myAnchor |
html heading: model.
myAnchor := html anchor.
myAnchor callback: [model addHours: -30].
myAnchor with: '<- Earlier'.
```

Now, let's discuss the two keyword messages sent to the new anchor. The simpler one comes last. The 'with:' message is being sent to the anchor tag with a single argument. The argument is a string literal, identified in Smalltalk with a straight single-quote character at the beginning and end. (If you need to insert a one single quote inside a string literal, put two in.) In Seaside, the 'with:' message should be the last one sent to a tag since it causes the content to actually be written to the HTML page. (Since it must be last, I don't use the 'yourself' at the end.)

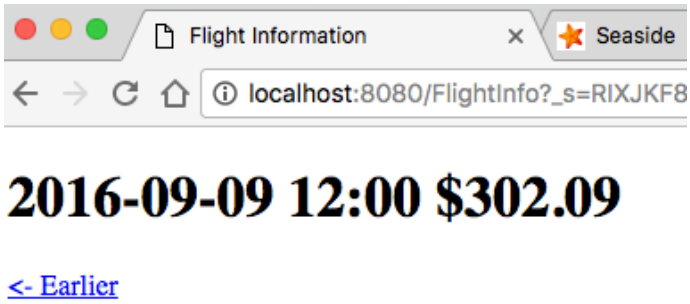
18. The 'callback:' keyword message introduces a new Smalltalk concept—a code block. In Smalltalk, any code inside square brackets ([]) is code that is *simply an object* that can be passed as an argument to any method that accepts an argument. Like any other object, it can be referenced by variables and messages can be sent to it at some later time. Thus, in our 'renderContentOn:' method we are not actually sending the 'addHours:' message; instead we are defining a block of code that *might* be executed later, at which time the 'addHours:' message will be sent.

A code block object knows its context—that is, the method in which it was defined. It also knows the method arguments and temporaries that existed at the time it was created. When the expressions contained in the block are later evaluated (by sending the message 'value' to the block), the expressions will properly reference instance variables, method arguments, and method temporaries.

By sending a code block to an anchor tag as the argument to a 'callback:' message, we are telling the anchor tag to hold onto this block, and when a user clicks on the anchor on the web page, Seaside will send the 'value' message to the code block, which will send the 'addHours:' message to the page's model with the argument of negative 30.

These Anonymous Functions (see http://en.wikipedia.org/wiki/Anonymous_function) give Smalltalk much of its power and flexibility. You might be familiar with some related approaches in other languages (JavaScript, Perl, Ruby, and of course Lisp have similar concepts). In C you can pass a pointer to a function as an argument to a function and in Java you can define inner classes, though these don't have the full capabilities of Smalltalk's blocks.

19. After saving the changes to the 'renderContentOn:' method, go back to your web browser and refresh the page. You should see a link below the heading and clicking on the link should cause the information to change.



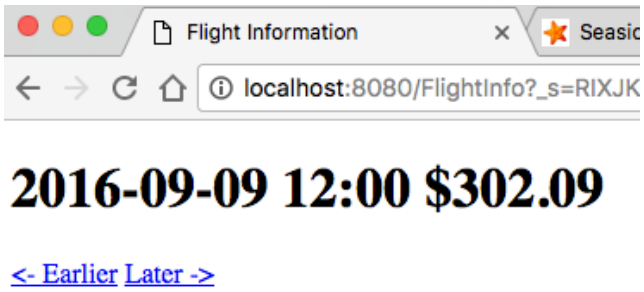
20. Next we will add a link to move to a later time. This time rather than doing the call directly in the block, we will call a local method that updates the model. Add the method 'later' to FlightInfoComponent, and then modify the 'renderContentOn:' method to call this new method. (These edits can be made from Pharo or from a web browser.) In this case we still have a code block being passed to the new anchor tag, but the code block calls a local method that sends a message to the model.

    Which approach you take is a matter of style and will generally be driven by the complexity of the callback operation. If you have any more that one message send, then the code should probably be put in its own method. A code block can be many lines long, but that would only clutter the 'renderContentOn:' method, which is already several lines long! In Smalltalk it is considered poor form to have a method longer than you can read in the code browser without scrolling, or about 6-8 lines long.

```
later

   model addHours: 30.
```

```
renderContentOn: html

   html heading: model.
   html anchor
      callback: [model addHours: -30];
      with: '<- Earlier'.
   html space.
   html anchor
      callback: [self later];
      with: 'Later ->'.
```

21. Try out the new page in your web browser. Click the links and note how the data changes.



22. Save your Pharo image before we go on to explore some other aspects of Seaside.

In this chapter we use the Flight Information application to learn how Seaside saves user data as part of a web application. But first, some background on the problem…

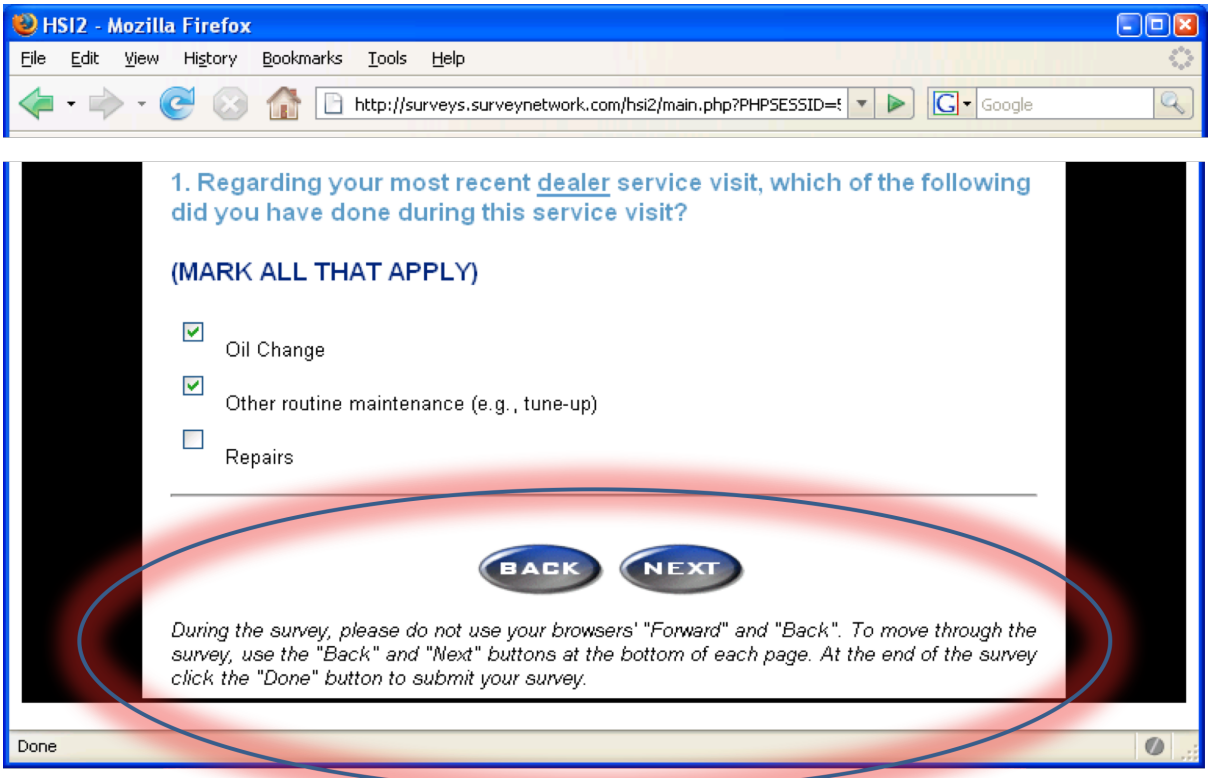
The HTTP protocol (which is used by web browsers to communicate with web servers) is by design very light-weight and intentionally avoids keeping any information that would tie one page request to another page request. This worked well when the web served static documents (the web was initially designed for the academic research community). Unfortunately, this does not work so well for complex applications such as e-commerce or travel reservation where we need to keep track of items added to a shopping cart or flights selected.

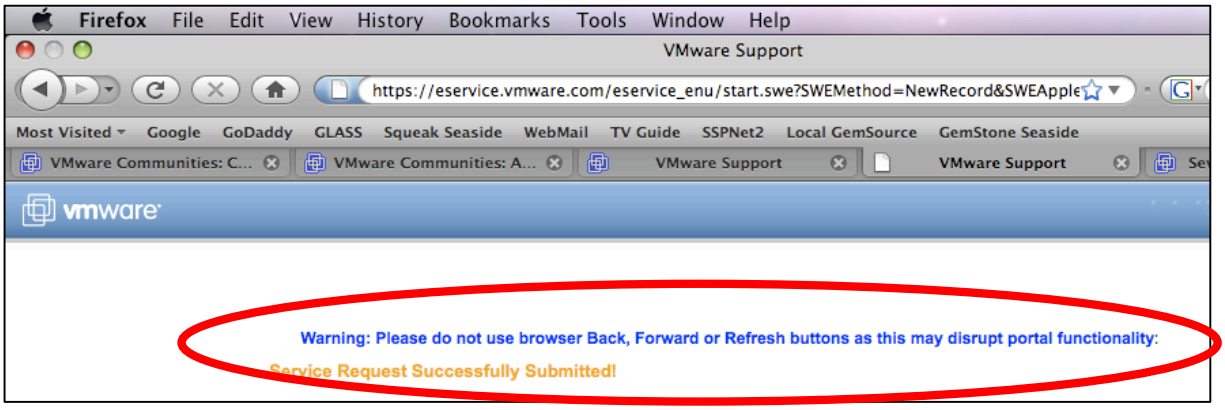
Various strategies are available to preserve state from one page to another. One approach is to add hidden fields to forms. When the form is submitted to the server, the hidden fields can contain saved data. This is okay if a form will be submitted, but doesn't work if the user clicks on a link (technically, the hidden field approach works for POST requests, but not for GET requests). Another approach is to use cookies. This works for both GET and POST as long as the user has not disabled cookies in the web browser. Also, the same cookie is sent for every request to the same server, so if the user has two tabs in the same browser, any changes in one will be visible to the other. A third approach is similar to "URL Rewriting" but instead of making the URL simpler or prettier, data is added to the URL (making it uglier).

In any case, one needs to consider what data is sent to the browser to be returned by the client (whether in a hidden field, in a cookie, or in the URL). If actual user data is encoded then nothing needs to be remembered on the server. This scales very nicely, but opens the application to hacking. (See the PayPal example below.)

The more secure alternative is to store user data on the server and send a *session key* to the client to be returned with subsequent requests to the server. With user data on the server it should be difficult for a hacker to modify server data other than through the web application. If the session key is hack resistant (say, a large random number), then there is little risk of someone hijacking another user's session. The down side of this approach is that the server must store data on each session and if there are a lot of sessions then there can be a lot of data. Further complicating this is that most users who start interacting with a web application will not complete the process (it is common to add a product to a shopping cart and not complete the purchase). The application then needs to decide when to expire stale sessions. In order to handle a lot of user sessions, the application needs to be able to store and quickly retrieve a lot of data.

One significant problem of storing user data on the server with a session key on the client is that the user could use the browser's back button to change the displayed data without communicating to the server that the user is looking at different data. Following are a couple screen shots of commercial web applications that instruct the user not to use the browser's back button. This seems like a rather clumsy solution to the problem.



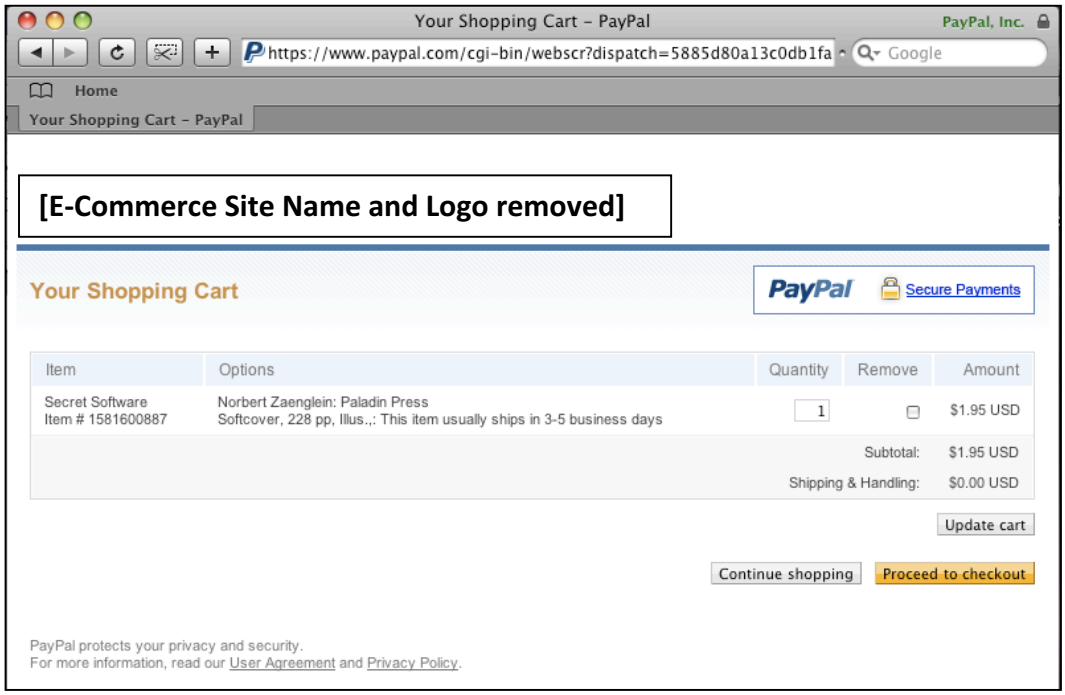


We will continue our Flight Information application to examine this problem and how Seaside addresses it.

A particularly striking example of sending critical data in an easily hackable manner is found in “Advanced Professional Web Design” by Clint Eccher (Charles River Media, 2007). Chapter 3 is titled “Understanding E-Commerce Functionality” and on page 72 describes how to submit a shopping cart to PayPal using a URL with encoded data. The example link is

```
<a href="https://www.paypal.com/cart/add=1&business=...&
item_name=...&item_number=...&amount=17.95&..."
target="_new"><img src=”add_to_cart.gif”></a>
```

When I first read Eccher’s book I tried the link with different values for the amount and there was every indication that PayPal would complete the order with the revised amount. Following is a screen shot of the shopping cart with a price of $1.95 instead of $17.95:



Note that the problem displayed here is not with Eccher’s description or his sample code, but with the credit card processing design he describes. (If using this approach it would be important for the seller to check the price paid by the customer before shipping the merchandise.)

1. At this point our Flight Information application allows a very trivial way of searching for flights but does not offer any way to select a displayed flight. Before we begin the following exercise, make sure you have the Seaside One-Click Experience running and that you can browse the Flight Information application in a web browser.

2. First, the method FlightInfoComponent>>#renderContentOn: (this is a typical Smalltalk way of identifying a class and method) is getting a bit long. We will start with copying most of the code to a new method and adding a line break after the earlier/later links. Next we will add a new anchor to report the selected flight.

```
renderChangeTimeLinksOn: html

   html anchor
      callback: [model addHours: -30];
      with: '<- Earlier'.
   html space.
   html anchor
      callback: [self later];
      with: 'Later ->'.
   html break.
```

```
renderContentOn: html

   html heading: model.
   self renderChangeTimeLinksOn: html.
   html anchor
      callback: [self inform: 'You selected ' , model printString];
      with: 'Book flight for ' , model printString.
```

3. In your web browser refresh the FlightInfo page to get the new elements, click the 'Later ->' link a few times, and then click on the 'Book flight' link. Your web browser should show a page showing the date, time, and price for the flight. Clicking the OK button takes you back to the flight information page.

4. Now click the 'Later ->' link a few times, note the price, click the browser's back button once, note the price, and then click the 'Book flight' link. Compare the price displayed on the inform page with the two prices. Because data is saved on the server, and because the server did not know that you clicked the back button, the 'Book Flight' link showed you the data saved on the server, not the data displayed in the browser.

5. We can see the same problem if we have two browsers or tabs. In your current web browser, right-click on the 'Later ->' link and select the menu option to open in a new window. In the second window click the 'Later ->' link a few times. Then switch back to the first window, note the displayed date/time/price, and then click the 'Book flight' link. Note that the 'wrong' flight was booked. Again, this happened because there is one FlightInfo instance that is kept in the 'model' instance variable of the one FlightInfoComponent being displayed on multiple windows.

6. To address this problem, Seaside gives you a way to identify objects for which the state at the time a page is rendered should be preserved and associated with that page. When a user interacts with that page in the future the state of the object is restored to how it was when the page was originally rendered. Thus, while there is still only one FlightInfo instance, its state (when & price) can be preserved by Seaside and restored if needed for a future request. To see this simply add a method to FlightInfoComponent:

```
states

   ^Array with: model.
```

7. After saving the new 'states' method, return to your web browser, click the 'Later ->' link a few times, click the browser's back button, note the displayed date/time/price, and then click the 'Book flight' link. Note that the correct flight is displayed.

   Seaside's approach of saving state on the server is powerful and easy, but it does come at a price. Now we are keeping information on the server not just for every user (or session), but for every page served to every user. This is a manifestation of the old adage that "There's no such thing as a free lunch." The nice thing is that it is available, and you can choose to use it if your application would benefit from this help. Also, keep in mind that until you measure performance (either space or speed) it is probably a mistake to be overly concerned.

8. Save your Pharo image before going on to the next chapter.

In this chapter we use the Flight Information application to learn about *continuations*, an often-cited but poorly understood feature of many Smalltalk dialects that allows Seaside applications to use subroutine calls to present user-interface components.

As we look at the advances in computer system technology, it is amusing to see how the pendulum swings back and forth. In the 1970s we had time-sharing systems in which multiple users could get character data on dumb terminals (initially teletypes then 'green screens'). The 'dumb' terminals became more sophisticated so that they could process more elaborate display attributes (bold, underline, blinking, reverse, etc.) culminating in the IBM 3270 terminal that was used to connect to mainframes. One of the features of the 3270 was that it reduced substantially the communication with the server. A screen full of data could be sent by the server to be displayed on the terminal, the user would enter data into pre-defined fields, and then press the <Enter> key to submit all the data back to the server in one chunk.

After a number of years in which computing became much more distributed and user interfaces became much more sophisticated (culminating in, say, the MacBook Air), the web era is taking us back to a model in which a (somewhat) less sophisticated terminal (the browser) displays chunks of text (though now with pictures and sound) and chunks of data returned to the central server when the user clicks a <Submit> button. While the browsers (thin clients) are closing the gap between them and the rich (or fat) client applications in terms of graphical user interface widgets, there are ways in which the programming models have gone in cycles as well.

One of the advances in software engineering was the introduction of the subroutine. As developers recognized the value of avoiding GOTO (as suggested by Edsger Dijkstra's letter "*Go To Statement Considered Harmful"* in 1968) they tended to write better code. Code could be more easily reused and the main (calling) code could be more abstract and better communicate intent. Instead of dealing with low-level details, a high-level program can describe what steps are being performed and rely on the subroutines to do the actual work.

The irony in this (for purposes of our discussion) is the dearth of true subroutine calling capability available in today's web frameworks. Yes, they have the ability to include other web components (like a page header or footer), but the process of writing a web application that sequences a series of web pages (like a shopping cart checkout) is not likely to include a program flow that looks like it would if the program were handling a rich (or fat) client application.

Note, for example, how much a web page link behaves like a GOTO statement. (This is the implication behind the blog title selected by Seaside's co-creator Avi Bryant: "*HREF Considered Harmful*.") Clicking on a link causes a request to be made for a new page—and the program that provided the link does not even know that you left its page! With the typical template framework, processing starts at the beginning of the page with each request. There is no easy way to call a subroutine that presents a web page and returns with the value retrieved from that web page.

Seaside, however, has such a capability built in as a trivial operation and we will use it to call a new component to select a date/time for our Flight Information application.

1. Launch the Seaside One-Click Experience and in the System Browser, click in the first column (say, on the 'GLASS' line) to get a class creation template. Create a new subclass of WAComponent that will present the user interface for selecting a date and time:

```
WAComponent subclass: #FlightInfoWhenComponent
   instanceVariableNames: 'dateSelector timeSelector'
   classVariableNames: ''
   category: 'GLASS'
```

2. Click in the third column to get a method creation template and add an initialize method that creates instances of two subclasses of WAComponent, WADateSelector and WATimeSelector:

```
initialize

   super initialize.
   dateSelector := WADateSelector new.
   timeSelector := WATimeSelector new.
```

3. When we are going to call this component, we need to tell it the current date/time so that it starts with a nice default:

```
when: aDateAndTime

   dateSelector date: aDateAndTime asDate.
   timeSelector time: aDateAndTime asTime.
```


4. As we've mentioned earlier, the one required method for a component is 'renderContentOn:'. In this example we introduce a form with a submit button. Furthermore, we use a table to lay out the various fields in the form. I'm aware that using tables for layout is frowned on by CSS purists, and better examples will come in later chapters, but I've chosen to use a table here because many people still use tables for layout and our purpose here is to teach Seaside (and save the religious wars for important questions, like how to use tabs to format code blocks ;-).

```
renderContentOn: html

"3"   html form: [
"4"      html table: [
"5"         html tableCaption: 'Select Date/Time for Flight'.
"6"         html tableRow: [
"7"            html tableData: 'Date:'.
"8"            html tableData: [html render: dateSelector].
"9"         ].
"10"        html tableRow: [
"11"           html tableData: 'Time:'.
"12"           html tableData: [html render: timeSelector].
"13"        ].
"14"        html tableRow: [
"15"           html tableData: ''.
"16"           html tableData: [
"17"              html submitButton
"18"                 callback: [self submit];
"19"                 with: 'Select'.
"20"           ].
"21"        ].
"22"     ].
"23"  ].
```

In the above method note that the code format matches how we might format an HTML document. At the top-level, we have a form that begins on line 3 and ends on line 23. Instead of using open and close tags (<form></form>) we use a left square bracket (to begin a code block) and a right square bracket (to end the code block). This is a way of telling the form element that everything in the block is inside the form. There is only one element in the form, a table element that begins on line 4 and ends on line 22. Inside the table are four elements, a table caption (line 5) and three table rows (lines 6-21).

Again, the line numbers are just comments and can be left out.

The three table rows each have two table data elements. In each row the first data element is a label (though in the third row the label is blank), and the second element is some other construct. The third row contains a submit button (rows 17-19) that should look a lot like our earlier anchor examples—it has a callback and a text label. The callback contains a block of code

that is set aside and not evaluated until the user clicks on the submit button. At that point the form data is submitted and the 'submit' method is invoked.

The more unusual code is on lines 8 and 12. Here we are simply rendering other components, the ones created in the 'initialize' method. Here are examples of true 'components' in which a small class does a simple thing and we can reuse it in a variety of places with minimal effort.

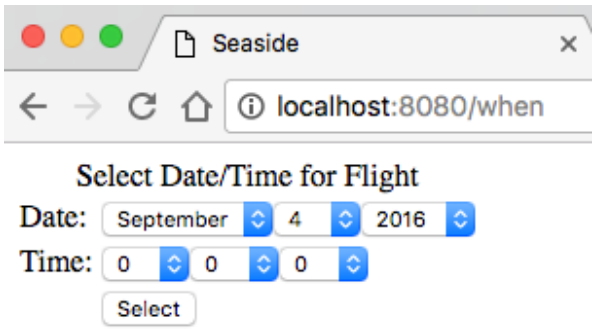
5. Finally, we need the 'submit' method that will be called when the user clicks on the 'Select' button. This code extracts the date and time from the child components, and constructs an instance of DateAndTime. The interesting piece here is the last line which is where the new DateAndTime is used as an argument to 'answer:' to return a value to the caller of the component. We will look at that shortly.

```
submit

    | date time dateAndTime |
    date := dateSelector date.
    time := timeSelector time.
    dateAndTime := DateAndTime date: date time: time.
    self answer: dateAndTime.
```

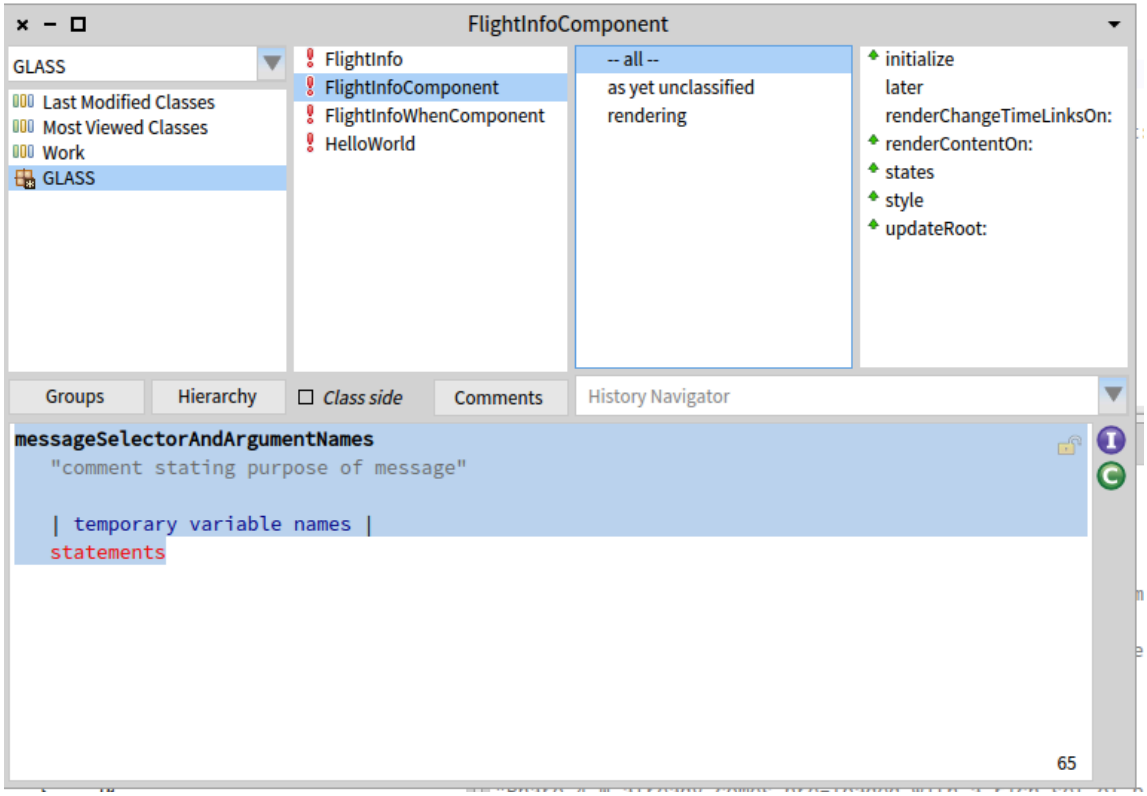
6. All this gives us a new component, but no clear UI for it. If you want to see the component by itself, you can register the component as an application and then view it using a web browser. Evaluate the following in a workspace.

```
WAAdmin register: FlightInfoWhenComponent asApplicationAt: 'when'.
```

7. With a web browser, navigate to http://localhost:8080/when and you should see the following:



8. In the System Browser in Pharo, switch back to the FlightInfoComponent (we are done with the FlightInfoWhenComponent for the moment) and click in the third column to get a method creation template.



Now we will add a method to FlightInfoComponent to call the new component (if you get an error it is likely because you are still on the FlightInfoWhenComponent; make sure to select the FlightInfoComponent class):

```
selectWhen

"3"    | component when |
"4"    component := FlightInfoWhenComponent new
"5"       when: model when;
"6"       yourself.
"7"    when := self call: component.
"8"    model when: when.
```

This method asks for a new instance of the FlightInfoWhenComponent class and assigns into it the current model's date/time. To review some Smalltalk dealing with lines 4-6:

a. The 'new' message is sent to the FlightInfoWhenComponent class which returns a new instance of the class (this is a unary message). The new instance will have its 'initialize' method called, so its two instance variables will have new components in them. The new instance is held on the execution stack temporarily.

b. The unary message ‘when’ is sent to the object in the ‘model’ instance variable and it returns a DateAndTime instance. (This message is sent before the ‘when:’ message because unary messages have precedence over keyword messages.)

c. The keyword message ‘when:’ is sent to the instance of FlightInfoWhenComponent created in (a) and this sets the default values of the WADateSelector and the WATimeSelector.

d. The message ‘yourself’ is sent to the receiver of the previous message (‘when:’), which is our new instance of FlightInfoWhenComponent created in (a) above. This message simply returns itself so is a slight performance overhead but serves a couple purposes. First, it means that the new instance of FlightInfoWhenComponent is the object returned by the final message send. Without it the returned object would be the result of the ‘when:’ message send, which might be our new component, but could be something else (like the DateAndTime argument); without going to read the code we couldn’t be sure. Second, it means that we could add another message send after line 5 without having to edit line 5. That is, if line 5 ended with a period instead of a semicolon, then we couldn’t add another message send without changing the period to a semicolon. Having to edit a ‘when:’ message send to add a ‘where:’ message send (for example), implies that something has changed about the ‘when:’ message, when it really hasn’t.

e. Finally, the reference to the object returned by the ‘yourself’ message is placed in the method temporary ‘component.’

Line 7 is where we send ‘call:’ with the new component. This tells Seaside to suspend the current method, show the new component (a FlightInfoWhenComponent) in place of the current component (a FlightInfoComponent), wait for the new component to send ‘answer:’, and take the object sent as an argument to ‘answer:’ and put a reference to it in the method temporary named ‘when.’ The DateAndTime returned by the new component is passed with the ‘when:’ message as an argument to the model.

The magic here is that the execution was interrupted in the middle of line 7, another web component was sent to the client browser, the user interacted with that component, and then finally when the second component decided it was done (by sending ‘answer:’), execution continued (hence the name *continuation*) where it paused in line 7.

The ability to treat a web page as a subroutine is something that few web frameworks allow. This is because each page request needs to finish execution by returning a page. At this point the web application framework is essentially done and waiting for another request. Each request starts at the beginning, creates a page, and finishes by returning the page. To stop in the middle of processing a page request means that the page is not returned. Thus, the call stack is unwound as part of the ‘return a new page’ action. This is particularly evident in web frameworks that use templates (and most do). The application logic has to start fresh at the top

of each page and if some conditional is required it can check for session data left over from previous interactions with the client (or data submitted by a POST request).

In Smalltalk, when the application reaches a point where it wants to be able to suspend execution of the current stack (as implied by a subroutine call) but still return from execution of the current stack (as required by the need to return a page to the client), the application can create a *continuation*. A continuation is a copy of the current stack, including all temporary variables and method arguments, with the property that it can be returned to later at the point where it was suspended. Execution can continue with a passed-in value that will (in Seaside) be used as the object returned after sending the 'call:' message.

9. Now that we have a 'selectWhen' method, we need to create something to call the method. Modify the 'renderContentOn:' method to add the last four lines:

```
renderContentOn: html

   html heading: model.
   self renderChangeTimeLinksOn: html.
   html anchor
      callback: [self inform: 'You selected ' , model printString];
      with: 'Book flight for ' , model printString.
   html horizontalRule.
   html anchor
      callback: [self selectWhen];
      with: 'Select Date/Time'.
```

The 'horizontalRule' is there to show you how to generate an <hr /> element. The new anchor uses the typical format of a callback and a label.

10. Try out the new functionality in your web browser and see how easy it is to enter a date/time and see the impact on the price.

11. We have completed the Flight Information application. Be sure to save your image.

Smalltalk

While we have discussed Smalltalk, this chapter will give a more focused description of the language.

Smalltalk is one of the original Object-Oriented Programming (OOP) languages and came from Xerox's Palo Alto Research Center (PARC) in the 1970s. It inspired much of today's graphical user interface design (Steve Jobs was given a demo of Smalltalk at PARC and went on to build the Macintosh) and influenced many of today's programming languages (especially Objective C and Ruby). While not as well-known as many subsequent languages, it has a vibrant community of users who continue to provide leadership in the software industry.

Like its influential predecessor Lisp, but unlike most of today's better-known languages, Smalltalk is **image-based** in that "programming" consists of modifying an existing system rather than creating a new system from scratch. That is, while the typical C program is created by compiling text files into an executable, a Smalltalk program is created by cloning an existing system and modifying it to provide new capabilities. The program itself is represented internally by objects (including classes and methods), and changes are made (i.e., code is written) by sending messages to existing objects.

An "image" is a snapshot (using a camera metaphor) of a live object space written to a binary file. An object space saved in this way can be exactly recreated by reading the image back into memory and continuing execution from the point of the snapshot (much like a fork() operation in C creates a new process with a copy of the existing data). This process of writing an environment to disk and reading it back is similar to the hibernate operation on a modern laptop computer—when the image is restored all the windows are open to the same place on the screen and they have the same content.

A typical running Smalltalk system consists of an operating system process (a virtual machine to interpret and/or compile source code) and an object space containing all the code and data (typically copied into RAM from the disk-based image). The virtual machine (VM) is also responsible for automatic garbage collection—reclaiming space used by unreferenced objects. If the VM terminates without saving an image of the object space, then changes made in the object space are thrown away and restarting from an earlier image will restore the object space to the prior state. Most Smalltalk dialects record programmer activity to a separate log from which code changes can be selectively reapplied so that programmers can experiment with minimal risk of losing work.

Because a Smalltalk program (classes and methods) is represented by objects in the object space, the act of writing a program involves manipulating the object space. You create a new class (a subclass) by sending a message to an existing class with arguments that describe the new class's schema and you create a new method by sending a message to a class with the source string for the new method. These operations are typically done using tools built into the environment. Because the tools and the code live together in the same object space, there is a rich tradition of extensions, such as refactoring tools (which originated in Smalltalk).

Smalltalk Syntax

Sending Messages (Instead of Issuing Commands)

Where other languages might have an elaborate syntax (including many keywords and operators), Smalltalk takes a simple idea—message passing—to an extreme. Instead of issuing a series of commands for the computer to follow, the programmer is scripting a series of polite requests that objects make of one another (and the receiver responds by sending other messages). The syntax of message sending is to name a variable (which always holds a reference to an existing object) and then specify the message with any arguments. This placement of the object first is backwards from the approach of most languages where the procedure is first. For example, to close a file, most languages would use a construct like the following (where the semicolon ends an expression):

```
Close(myFileHandle);
```

In contrast, Smalltalk puts the object first rather than the command first (and expressions end with a period or dot):

```
myFileHandle close.
```

Three Message Types

There are three types of messages: (1) **unary** messages (like the 'close' example above), (2) **binary** messages that take exactly one argument (like '+' and '*' in the example below), and (3) **keyword** messages where one or more arguments follow a word ending with a colon (like 'between:and:' below). The precedence is unary, binary, and keyword, with left-to-right within a particular message type. While elegant and consistent, this creates some confusion for people coming from other languages. The following expression (with two binary messages) evaluates to 20 in Smalltalk (using strict left-to-right evaluation for binary messages), while other languages would give 14 (treating '*' as a language-defined operator with higher precedence than '+'):

```
2 + 3 * 4.
```

As in other languages, parenthesis can be used to change the order of evaluation. If you want the above expression to evaluate to 14, use the following:

```
2 + (3 * 4).
```

Keyword messages use words rather than just the position to identify the argument. For example, to check for a value in a certain range a traditional language would use something like the following:

```
between(x, y, z);
```


Smalltalk makes more explicit which is the value being tested, which is the lower bound, and which is the upper bound. While this requires some more typing, it makes the code easier to read:

```
x between: y and: z.
```

Five Reserved Words

In Smalltalk there are only five reserved words: *self*, *super*, *true*, *false*, and *nil*. The first two, *self* and *super*, are "pseudo-variables" that are used in a method to reference the receiver (an object) of the current message. When a message is sent to *self*, method lookup starts in the class of the receiver (without regard to the class where the method being executed is implemented). When a message is sent to *super*, method lookup starts in the superclass of the class where the method being executed is implemented. The remaining reserved words, *true*, *false*, and *nil*, are actually globals that always reference the single instance of the classes True, False, and UndefinedObject respectively.

Control Flow (Loops and Conditionals)

The things that are typically done in other languages using reserved words, including class definition and flow-of-control, are done in Smalltalk by sending a message to an object. Thus, while a traditional language would use a construct like the following for a loop (where *for* is a command):

```
for (i = 0; ++i; i < sizeof(array)) { DoSomething(array(i)); };
```

Smalltalk would use a message ('to:do:') sent to an integer to create a similar loop:

```
0 to: array size do: [:i | (array at: i) doSomething].
```

Here the 'to:do:' message is sent to an instance of SmallInteger with two arguments, another SmallInteger and a Block (code that can be executed later). In fact, because this sort of iteration is extremely common, another message, 'do:,' is used to handle iteration without explicitly managing the counter (allowing the programmer and the code to focus on a higher level):

```
array do: [:each | each doSomething].
```

In the above example, the code block (inside the square brackets) is an object passed to the 'do:' method and is evaluated repeatedly, once for each object in the array. Code blocks can have arguments, like 'each' in the above example (the preceding colon is syntax for naming a block argument).

Code blocks are also used for conditionals. In the following example, two code blocks are provided as the arguments to the message 'ifTrue:ifFalse:', where the receiver is a Boolean expression:

```
(x < 0) ifTrue: [self doThis] ifFalse: [self doThat].
```

Compare the above Smalltalk code with the following from a more traditional language where *if* and *else* are reserved words in the language syntax rather than simply messages to objects:

```
if (x < 0) then {doThis()} else {doThat()};
```


Assignment and Return

Smalltalk has a couple built-in operators that are not message sends. The first is variable assignment:

```
   x := 3.
```

The second built-in operator specifies an immediate return from a method with a particular value:

```
   ^x.
```

All methods return an object—either explicitly or implicitly. By default, the object returned is the receiver, but using the up-arrow (as above), a method can return an explicit value. If the goal is to return early from a method that would not otherwise return a value, then the return value is typically (by convention) the default return value, *self*:

```
   (x < 0) ifTrue: [^self].
```

Method Definitions

A method is always defined in the context of a class and starts with a template or prototype that includes the method name and names for the arguments. Temporary variables are defined in a method or block by enclosing the names in vertical bars before any expressions:

```
add: anObject
   | sum |
   sum := self + anObject.
   ^sum.
```

Message Cascades

Because it is often useful to send a series of messages to the same object, Smalltalk provides a shortcut so that you don't have to repeat the receiver in a method. In the following example, two messages are being sent to *self*, but the receiver is only identified once (note the semicolon):

```
   self doThis; doThat.
```

Classes are Objects

In Smalltalk, all values are objects that are instances of some class. Unlike many other OO languages, Smalltalk implements the classes themselves as objects. Methods can be defined in the metaclass for a class, so that messages sent to the class will find and evaluate the methods. This provides what in other languages might be characterized as a *static function*. In Smalltalk, we speak of having methods "on the instance side" of a class (where they will be evaluated if a message is sent to an instance of the class) and having methods "on the class side" (where they will be evaluated if a message is sent to the class). The tools typically make this easy to manage.

Constants

The Smalltalk language syntax provides for creating (or referencing) various constants in a method.

A **Number** is generally defined in a familiar manner. A series of digits without punctuation (and with an optional preceding minus sign) is interpreted as an instance of the class Integer. If a decimal point (or dot) is included, then the number will be interpreted as an instance of the class Float. Floating point numbers are also permitted to have an exponent component (with an optional sign). A number that ends with 's' followed by one or more digits will be treated as an instance of the class ScaledDecimal.

A **Character** can be one or more bytes long, depending on the dialect. The dollar symbol introduces the character constant. The following identifies a Character with the code point (ASCII value) of 65.

```
x := $A.
```

A **String** is defined as a series of characters enclosed in a single-quote or apostrophe character ('). To include a single-quote inside a string, simple double it. Thus, 'ab''cd' is a five-character string where the third character is a single-quote character.

A **Symbol** is a subclass of String in which the system guarantees that only one instance with the specified sequence of characters will exist in the object space. This allows for more efficient equality comparison because rather than doing a character-by-character test, the system only needs to compare the object IDs of two symbols to see if they are the same. Symbols are often used as flags in the way that an enumeration or #define constant might be used in C. It is more efficient than a string and more readable than a number. To define a constant symbol, precede a string constant with a hash symbol. In many common situations (a string with only alphabetic characters), the quotes may be excluded. The following example shows two constant symbols:

```
x := #'this is a symbol'.
y := #thisIsAlsoASymbol.
```

A constant **Array** can be defined in a method using a hash and left parenthesis to begin the list and a right parenthesis to end the list. Inside the list may be any other constant (including Arrays). The following example shows a constant Array containing six objects, an Integer, a Float, a Character, a String, a Symbol, and another Array with three objects (elements are separated with whitespace):

```
x := #(42 -1.25e-13 $A 'hello' #world #(1 2 3))
```

Comments

Comments are included in a method by enclosing them with double-quote characters ("). To include a double-quote character in a comment, double it. The following line has a comment:

```
x := Float pi. "3.14159265358979"
```


Scope for Name Lookup

As code (generally a method) is compiled, names are looked up using a succession of six (6) scopes, listed here from innermost (found first) to outermost (found last):

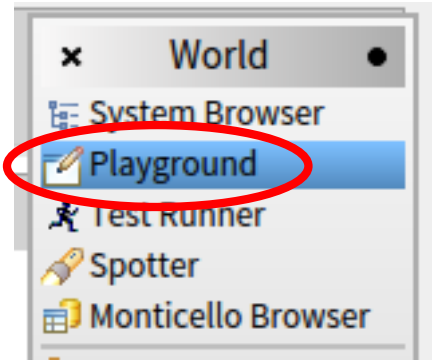
1. **Block arguments and temporaries**
    a. Block temps are initialized to nil (the sole instance of the UndefinedObject class) each time the block is evaluated
    b. Visible only within the block
2. **Method arguments and temporaries**
    a. Method temps are initialized to nil each time the method is called, even if called recursively
    b. Visible only to code within the method
3. **Instance variables**
    a. Initialized to nil when the object is instantiated
    b. References are preserved during the life of the object
    c. Values are visible only to methods defined in the class where the variable is defined and its subclasses
    d. No direct access is permitted in standard Smalltalk; the only way to get the value of an instance variable from outside the object is to send a message to the object
4. **Class variables**
    a. Initialized to nil when the class is first defined
    b. References are preserved during the life of the class
    c. Values are visible to all methods in the metaclass, in the class, and in all subclasses
5. **Pool Dictionaries**
    a. Each class definition can have zero or more key/value collections (called a Dictionary in Smalltalk, but typically called a Hash in other languages)
    b. A single dictionary can be shared among multiple classes
    c. Values are visible and shared among methods in all classes that reference the pool
    d. These are typically used for shared constants (e.g., map a color name to a number)
6. **Globals**
    a. Smalltalk provides a global namespace that is visible and shared by all methods
    b. These are generally used to hold classes

By convention, variables in scopes 1-3 are named with an initial lowercase letter and variables in scopes 4-6 have names that begin with an initial capital letter. As in other languages, it is considered a good engineering practice to use the narrowest scope that will work correctly. That is, a global is generally avoided when something like a class instance variable would do. And while a block can reference (read and write) method temporary variables, if the variable is only used within the block and does not need to preserve its value between block evaluations, it would likely be more efficient to move the variable to inside the block (as the compiler can generate simpler code).

Note that a class (an instance of Metaclass) may have instance variables. The values will be visible only to methods on the metaclass itself (i.e., class-side methods), not to instances, and not to subclasses.

The Workspace

In Chapter 2 we looked briefly at the three windows that are part of the Seaside One-Click Experience. The Workspace is a text area where Smalltalk expressions can be typed and evaluated. To get another Workspace you may left-click on the desktop to get a World menu and select 'Playground' from that menu.

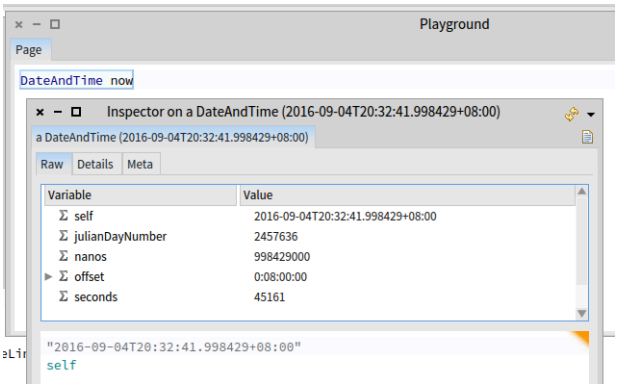


After you have the new Workspace, try entering a Smalltalk expression (like '100 factorial') and then type <Ctrl>+<P> (for 'print-it'). You should see a very large integer highlighted. Press <Backspace> or <Delete> while the text is selected to delete it.

If you type <Ctrl>+<D> (for 'do-it'), the expression will be evaluated but the resulting object will be ignored. This is useful when we want to evaluate an expression to cause some side-effect to occur (like registering a new Seaside application) and are less interested in seeing a printout of the result.

Sometimes we want to investigate a returned object in some detail. Most Smalltalk dialects provide an *inspector* to look at objects in more detail that simply printing some text in a workspace. In a Workspace enter the following and press <Ctrl>+<I> (for 'inspect-it'; on a Mac you might need to use <Apple/Command>+<I>).

```
DateAndTime now.
```



The window in front of the Workspace is an inspector and it has as its title the name of the class. The inspector has three tabs and the first tab has two panes: (1) a scrolling list of the object and its instance variables, (2) a text area with a printed representation of the selected instance variable and a place where expressions can be entered and evaluated.

If you right-click in the Workspace you will get a context menu of commands that apply to the Workspace. Some of these are similar to what you would expect in a text area (cut/copy/paste/etc). Others provide commands for executing Smalltalk code.

The System Browser

As we saw with the Workspace, you can get a new System Browser by clicking on the Browser icon on the desktop and dragging it to a convenient location. The System Browser gives you a way of looking at the classes and methods defined in the current object space (usually called the *image*). The first column lists categories used to group classes. Scroll through the list to see a sampling of categories:

- Kernel: Forms the innermost foundation for the Smalltalk library
- Collections: Classes like Array and Set; used to aggregate objects
- Files: Classes used to communicate with the native operating system's files
- Balloon, Graphics, Morphic: Classes used to create Pharo's Graphical User Interface (GUI)
- Network: Classes used to communicate with the native operating system's networking layer
- SUnit: The original XUnit testing framework that inspired various other unit testing frameworks
- Compiler, System: Classes that support the internal operation of Pharo Smalltalk
- Monticello: A source code management and version control system
- Refactoring: Classes to support automated refactoring of code
- Seaside: A framework for developing web applications in Smalltalk
- Scriptaculous: An add-on to Seaside to support this popular JavaScript library

If you right-click in the class category list you get a context menu that allows various category-related operations (add/delete/etc.). Perhaps most useful is the 'find class…' menu command. This will offer a dialog box into which you can enter a fragment of a class name. Pharo will then present a list of classes with a matching name. When you select one the browser will select the category and class.

The second column shows a list of classes in the selected class category. The list is organized in a hierarchy (and alphabetically when classes have a shared superclass). Right-click to get a context menu, including the following:

- new class template: Replace the text area at the bottom with a generic class definition
- subclass template: Provide a template for creating a subclass of the selected class
- browse: Open a new System Browser with this class selected
- browse hierarchy: Open a class hierarchy browser showing all superclasses (even those not in the current class category)
- browse references: Show a list of every method in the system that references this class
- chase variables: Open a browser showing instance variables and methods that reference a selected instance variable

Below the second column is a checkbox that allow you to view methods on the instance side and class side of the class. A frequent error is creating a method on the wrong side of a class. Most of the time you should be on the instance side; when you go to the class side be sure to come back.

The third column gives a list of method categories and has a context menu that allows you to manage method categories (add, rename, remove, etc.) and has other tools similar to the class context menu. The fourth column gives a list of the methods in the selected method categories for the selected class.

This list has a context menu that allows management of the methods and has other tools similar to the class context menu.

Below the four columns are a series of buttons that mostly open new browsers that are also available from the context menus. For example, when a method is selected it is often interesting to browse senders of that message or other implementers of that message.

Most Pharo windows (that you will interact with in the context of writing a Seaside Application) have various standard window manipulation capabilities. You can close a window by clicking in the red circle at the top left. You can move a window by clicking on the title bar and dragging. You can resize a window by dragging one of the four corners.

The Changes Browser

One very valuable tool is the Changes Browser. Left-click on the desktop to get a World menu, select 'Tools…' then 'Recover lost changes…'. This will present you with a list of events in reverse chronological order (the most recent will be at the top). The 'SNAPSHOT' event refers to saving an image of your object space and the 'QUIT' event refers to quitting Pharo without saving an image. If your Pharo image crashes (or you quit without saving), you can get back all of your work. Selecting one of the events (generally the most recent one) will show you a list of your activity including class definitions, method definitions, and expressions evaluated in a workspace. You can replay any or all of these actions to get your object space back to the point where you lost your work.

The Transcript

The Transcript is a special workspace that is associated with the global named 'Transcript.' From the World menu (left-click on the desktop) select 'Tools…' and 'Transcript.' From another workspace evaluate the following expression with <Ctrl>+<D> (for 'do-it').

```
Transcript cr; show: 100 factorial printString.
```

The string should appear on the Transcript. This is quite handy for debugging as an alternative to interrupting the code execution. You can dump a string to the Transcript at various places in your code to see what is happening. This is similar to what you would do with 'printf()' and standard output in other languages.

In this chapter we start a new Seaside application that will incorporate more of the complexities normally found in a sophisticated web application. The primary focus of this chapter is how to incorporate CSS (for layout and style) and images in your web application.

Imagine that your child has been recruited to play on the best youth football (*soccer* for you Americans) team in the region, *Los Boquit*as.[*] All the parents are expected to be involved and instead of coaching on the field, you have agreed to create a web site to manage some team information. After a review of various technologies, you have decided to use Seaside.

Following is a screen shot of the basic structure, showing various pieces of a web site (header, sidebar, main, image, footer), that we will build.

This layout is, of course, completely arbitrary. Web page layout is best done by a designer and that designer should provide sample HTML and the CSS to go with the HTML. As a programmer, your job is to translate the HTML into Smalltalk code and reference the supplied CSS.

[*] The first version of this tutorial was in 2007 and was held in Buenos Aires, the home of *Club Atlético Boca Juniors*. *Boca* is Spanish for 'river mouth' and the club is in a neighborhood on the mouth of the Matanza River. The idea of naming a children's team 'Little Mouths' seemed appropriate.

1. First we will define a home page for the site. We will use 'LB' (for *Los Boquitas*) as the prefix for our classes. While some Smalltalk dialects (Cincom and GemStone) support namespaces, this is a convention that helps avoid name conflict when porting code across dialects. Launch Pharo and enter the following class definition:

```
WAComponent subclass: #LBMain
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LosBoquitas'
```

   a. Previously we have registered our application by evaluating an expression in a workspace. As the expression gets more complex, it becomes desirable to put the expression in a method and then just call it from a workspace. This allows us to manage the code with our source code management tools (tracking versions, etc.). An added benefit is that when a class is first loaded using the code management tools, the message 'initialize' is sent to the class. This means that our application can be automatically registered when it is loaded into a new Smalltalk object space.

      Click on the 'Class' button under the class list to specify that we are adding a *class-side* method. In this method we register the name (boquitas) and also specify that we want to use a WASession as the session class. This is the default and it expires a session that is inactive for a configurable period (the default is 10 minutes).

```
initialize
"
   LBMain initialize.
"
   super initialize.
   (WAAdmin register: self asApplicationAt: 'boquitas')
      preferenceAt: #sessionClass put: WASession;
      yourself.
```

   b. Now we are ready to add some content. Add this *instance*-side method to LBMain (uncheck the '*Class side*' checkbox to switch from the class side back to the instance side):

```
renderContentOn: html

   html heading
      level: 1;
      with: 'Los Boquitas Soccer Team'.
```

2. Now initialize your component by executing the following in a workspace:

```
LBMain initialize.
```

   Alternatively, you can switch back to the class side, select the initialize method, click anywhere on the third line, and press <Ctrl>+<D>. Don't forget to switch back to the instance side.

3. Finally, start at the dispatcher page, http://localhost:8080/browse, and follow the link to boquitas. If your application is not visible, then go back and see if you put the proper methods on the class side of LBMain. If they are on the instance side, then things will not work.

4. Now we will start the process of adding an image.

    a. Modify the render method in LBMain so that there is an image tag but it points to a file that does not exist:

```
renderContentOn: html

   html heading
      level: 1;
      with: 'Los Boquitas Soccer Team'.
   html image
      url: 'boquitas.jpg';
      yourself.
```

    b. Try viewing the page and notice that the image does not display. Depending on your browser, a placeholder might be displayed. At a minimum we need some alternate text to be displayed when the image is missing. Modify the render method again:

```
renderContentOn: html

   html heading
      level: 1;
      with: 'Los Boquitas Soccer Team'.
   html image
      altText: 'children playing soccer';
      url: '/images/boquitas.jpg';
      yourself.
```

    c. Try viewing the page and see if the alternate text is displayed. In Safari the text is not displayed, but this is a feature, not a bug! Technically, the official definition of the alt attribute is that it is for user agents (browsers) that are unable to display images (such as screen readers for visually impaired users). Since Safari is capable of displaying images, the alt attribute is ignored.

d. Now we could update the link to reference a site that does have the picture. (If you are not connected to the Internet, skip to step #5.) Modify the render method to point to an external server with the file. (If you are running a local server, you can download the file from seaside.gemstone.com and install it in your local images directory.)

```
renderContentOn: html

   html heading
      level: 1;
      with: 'Los Boquitas Soccer Team'.
   html image
      altText: 'children playing soccer';
      url: 'http://seaside.gemtalksystems.com/images/boquitas.jpg';
      yourself.
```

e. View the page and verify that the image shows.

5. While having an external server provide static data (such as images) is efficient (reducing the load on your local server—in our case the Pharo VM), it does mean that your application is not completely self-contained. Sometimes for development, testing, or demos it is nice to have even static data served from Smalltalk (and as long as you can't measure the performance impact, why not?). To support serving static data, Seaside provides an abstract class, WAFileLibrary.

   a. Save a copy of the image to a local directory. You can do this from your web browser by right-clicking on the image and selecting 'Save image as…' (the exact menu command will differ based on your web browser). Alternatively, copy the file from a CD/DVD if included.

   b. Create a subclass of WAFileLibrary to handle static data for our application (click in the :

```
WAFileLibrary subclass: #LBFileLibrary
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LosBoquitas'
```

   c. Add the downloaded image to the file library by evaluating an expression in a workspace that includes the path to the file (this will, of course, differ based on your machine).

```
LBFileLibrary addFileAt: 'C:\temp\boquitas.jpg'
```


d. Now using the System Browser, note that a new method (‘boquitasJpg’) has been added to the instance side of LBFileLibrary. Modify the render method to reference the new library. Notice how the class LBFileLibrary understands the ‘/’ message and returns a reference to the static file.

```
renderContentOn: html

   html heading
      level: 1;
      with: 'Los Boquitas Soccer Team'.
   html image
      altText: 'children playing soccer';
      url: LBFileLibrary / 'boquitas.jpg';
      yourself.
```

e. If you view source in your web browser, you can see that Seaside generated the following element for the page:

```
<img alt="children playing soccer" src="/files/LBFileLibrary/boquitas.jpg">
```

6. Adding a page title.

a. Note that the page is simply titled "Seaside" rather than something more descriptive. Add the following instance-side method to LBMain:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: 'Los Boquitas'.
```

b. View the page again and note that it now has a title.

7. Creating multiple areas.

    a. The typical web site has a header, a side-bar, a main content area, and a footer. We will now add these pieces by modifying the render method again. Note that the #'with:' message is the last one sent to each element since this causes the HTML tag to be written and closed.

```
renderContentOn: html

   html div
      id: 'allcontent';
      with: [
         html div
            id: 'header';
            class: 'section';
            with: [
               html heading
                  level1;
                  with: 'Los Boquitas Soccer Team'.
            ].
         html div
            id: 'main';
            class: 'section';
            with: [
               html image
                  altText: 'children playing soccer';
                  url: LBFileLibrary / 'boquitas.jpg';
                  yourself.
            ].
         html div
            id: 'sidebar';
            class: 'section';
            with: [
               html heading
                  level2;
                  with: 'Sidebar'.
            ].
         html div
            id: 'footer';
            class: 'section';
            with: [
               html text: 'Copyright (c) ' , Date today year printString.
            ].
      ].
```

    b. View the page and confirm that the various pieces exist. Note, however, that they do not have any formatting.

8. Until recently, the typical way of doing page layout was to use a table. This approach is no longer recommended and the better approach is to use Cascading Style Sheets (CSS). In chapter 5 we saw that a 'style' method on our component could provide style. This is useful for simple experiments, but Seaside provides alternatives.

    a. Add the following method to the instance side of LBFileLibrary (not LBMain!):

```
boquitasCss

^'body {
    font-family:        Georgia, "Times New Roman", Times, serif;
    font-size:          small;
}

#allcontent {
    width:              770px;
    padding-top:        5px;
    padding-bottom:     5px;
    background-color:   #fff0d0;
    margin-left:        auto;
    margin-right:       auto;
}

*.section {
    background-color:   #ffeabf;
}

#header {
    margin:             10px;
}

#main {
    padding:            15px;
    margin:             0px 10px 10px 10px;
    width:              550px;
    float:              right;
}

#sidebar {
    padding:            15px;
    margin:             0px 600px 10px 10px;
}

#footer {
    color:              #204670;
    text-align:         center;
    padding:            15px;
    margin:             10px;
    clear:              right;
}'
```


b. Modify the root component to reference the style sheet (back to LBMain!):

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: 'Los Boquitas'.
   anHtmlRoot link
      type: 'text/css';
      beStylesheet;
      addAll;
      url: LBFileLibrary / 'boquitas.css';
      yourself.
```

9. Let's refactor the render code so that it is more modular.

a. Add the following four methods:

```
renderHeaderOn: html

   html div
      id: 'header';
      class: 'section';
      with: [
         html heading
            level1;
            with: 'Los Boquitas Soccer Team';
            yourself.
      ];
      yourself.
```

```
renderMainOn: html

   html div
      id: 'main';
      class: 'section';
      with: [
         html image
            altText: 'children playing soccer';
            url: LBFileLibrary / 'boquitas.jpg';
            yourself.
      ];
      yourself.
```



```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Sidebar';
            yourself.
      ];
      yourself.
```

```
renderFooterOn: html

   html div
      id: 'footer';
      class: 'section';
      with: [
         html text: 'Copyright (c) ' , Date today year printString.
      ];
      yourself.
```

b. And now modify the renderContentOn: method to call these new methods:

```
renderContentOn: html

   html div
      id: 'allcontent';
      with: [
         self
            renderHeaderOn: html;
            renderMainOn: html;
            renderSidebarOn: html;
            renderFooterOn: html;
            yourself.
      ].
```

c. View the application in a web browser to confirm that it displays the page shown at the beginning of this chapter.

10. Save your Pharo image.

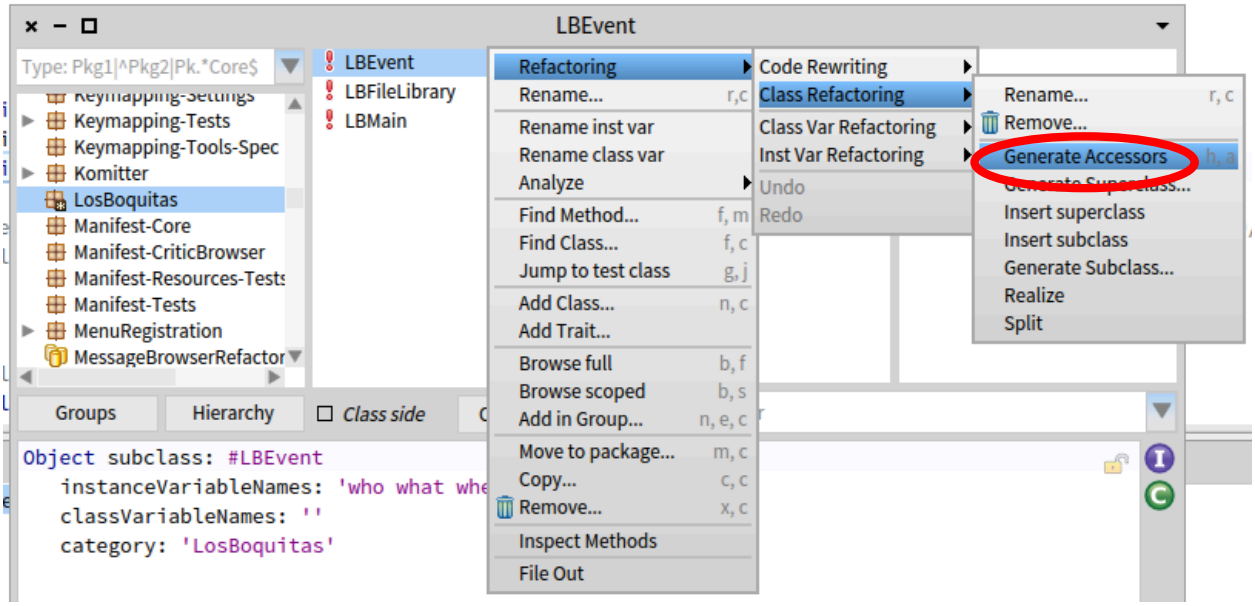
In this chapter we enhance the Los Boquitas application with a new component showing upcoming events in a table.

1. First, we need to have some events to display.

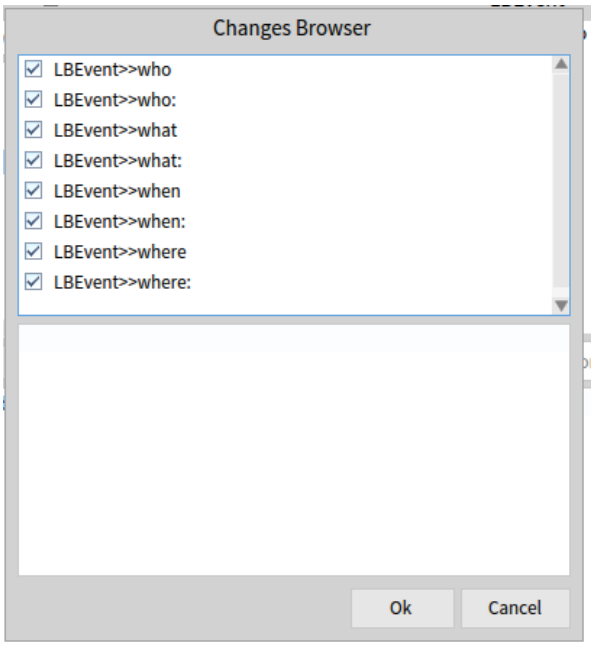
    a. We will start by defining an event class.

```
Object subclass: #LBEvent
   instanceVariableNames: 'who what when where'
   classVariableNames: ''
   category: 'LosBoquitas'
```

    b. Next we will create accessors for the instance variables. Rather than creating the methods one at a time, you can use some of Pharo's refactoring tools to create the methods. Select LBEvent, right-click and select 'refactor class' then 'accessors'.



c. The refactoring tool will show you the proposed new methods and give you a chance to accept or cancel before the changes are installed. Click the 'Ok' button.



d. Add a method to support sorting the events.

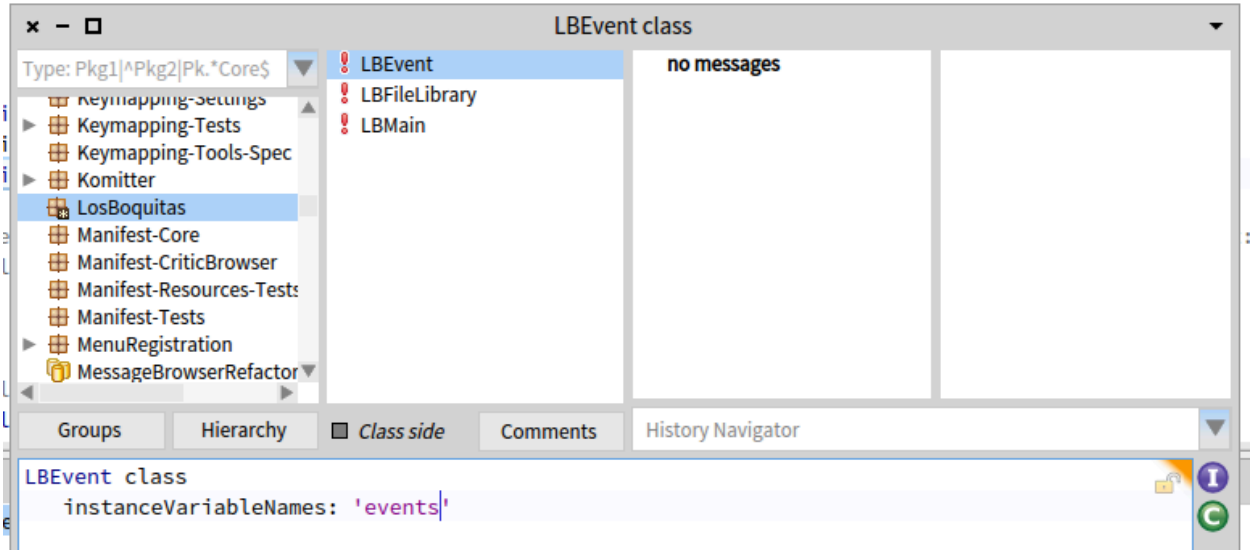
```
<= anEvent

   ^self when <= anEvent when.
```

e. Add an initialize method to ensure that something is in each instance variable.

```
initialize

   super initialize.
   who := 'players'.
   what := 'practice'.
   when := DateAndTime noon.
   where := 'field'.
```


2. Now we need a place to put the events. In most web frameworks and languages we would now start a discussion of setting up a relational database. In Smalltalk, however, we prefer to avoid the 'object-relational impedance mismatch' problem (see http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch) as long as possible. Instead of an external database that must be configured and mapped to, we will save our event objects in a *class instance variable* on the LBEvent class.

    a. In the Pharo System Browser, select LBEvent in the class list and then click on the '*Class side*' checkbox below the class list. This will change the class definition to a definition for the class instance variable. Edit the text area to add an 'events' class instance variable and save the text.



    b. Now click in the method category list to get a method template. Add a *class-side* method to access the events.

```
events

   events isNil ifTrue: [events := IdentitySet new].
   ^events.
```


c. Add a *class-side* method to create some sample events.

```
createEvents
"
   LBEvent createEvents.
"
   events := nil.
   self events
      add: (self new
         who: 'family';
         what: 'registration';
         when: DateAndTime noon;
         where: 'Clubhouse';
         yourself);
      add: (self new
         who: 'players';
         what: 'practice';
         when: (DateAndTime noon + (Duration days: 1));
         where: 'field';
         yourself);
      add: (self new
         who: 'guests';
         what: 'game';
         when: (DateAndTime noon + (Duration days: 2));
         where: 'Memorial Park';
         yourself);
      yourself.
```

d. In the SystemBrowser, click anywhere on the third line of the method (the one sending the 'createEvents' message) and press <Ctrl>+<D> (for 'do-it'). By adding the expression to the method as a comment, we can evaluate it without having to go to a workspace.

3. Now we will define a component to display the schedule.

```
WAComponent subclass: #LBScheduleComponent
   instanceVariableNames: 'listComponent'
   classVariableNames: ''
   category: 'LosBoquitas'
```

a. The goal is to embed this component into the main application, but for purposes of development and testing we will treat this as a stand-alone component (or application). Register the application by evaluating the following in a workspace.

```
WAAdmin register: LBScheduleComponent asApplicationAt: 'boquitas-schedule'.
```

b. Add a place-holder render method.

```
renderContentOn: html

   html heading: self class name.
```


c. In a web browser, navigate to the dispatcher (http://localhost:8080/browse) and confirm that the new component is in the list and that it displays the class name.

4. Now we will add a real display capability to the component.

a. Add four methods to define report columns and an initialize method to create a table report using those columns.

```
whoReportColumn

   ^WAReportColumn new
      title: 'Who';
      selector: #who;
      clickBlock: nil;
      yourself.
```

```
whatReportColumn

   ^WAReportColumn new
      title: 'What';
      selector: #what;
      clickBlock: nil;
      yourself.
```

```
whenReportColumn

   ^WAReportColumn new
      title: 'When';
      selector: #when;
      clickBlock: nil;
      yourself.
```

```
whereReportColumn

   ^WAReportColumn new
      title: 'Where';
      selector: #where;
      clickBlock: nil;
      yourself.
```



```
initialize

   | columns |
   super initialize.
   columns := Array
      with: self whoReportColumn
      with: self whatReportColumn
      with: self whenReportColumn
      with: self whereReportColumn.
   listComponent := WATableReport new
      columns: columns;
      rowPeriod: 1;
      yourself.
```

b. Now modify the render method to show the table.

```
renderContentOn: html

   listComponent rows: LBEvent events asSortedCollection.
   html render: listComponent.
```

c. Starting from the dispatcher (http://localhost:8080/browse) in a web browser, view the schedule component and confirm that it shows three rows of four columns. (Don't click on the anchors yet!)

5. Now we will update our main application to make room for a child component.

a. Change the class schema for LBMain to add an instance variable to hold the component being displayed in the main region.

```
WAComponent subclass: #LBMain
   instanceVariableNames: 'mainArea'
   classVariableNames: ''
   category: 'LosBoquitas'
```


b. Modify LBMain>>#renderSidebarOn: to change the heading.

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
      ].
```

c. Return to your web browser and display the home page (http://localhost:8080/boquitas). It should have the new text now ('Menu' instead of 'Sidebar').

6. Add a menu to the sidebar.

a. Modify the sidebar render method as follows:

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         html anchor
            callback: [mainArea := LBScheduleComponent new];
            with: 'Events'.
      ].
```

b. View the home page in a browser and confirm that the <Events> link is present. Clicking on it does not have any impact, but it is there!

c. We want the render method to use the mainArea component if it exists; otherwise, the image will be displayed. Modify the renderMainOn: method as follows.

```
renderMainOn: html

   html div
      id: 'main';
      class: 'section';
      with: [
         mainArea notNil ifTrue: [
            html render: mainArea.
         ] ifFalse: [
            html image
               altText: 'children playing soccer';
               url: LBFileLibrary / 'boquitas.jpg';
               yourself.
         ].
      ].
```

d. View the home page in a web browser and confirm that the event list displays when you click on the 'Events' link in the sidebar.

e. We now want a way to return to the home page. Modify the sidebar render method as follows:

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         html anchor
            callback: [mainArea := nil];
            with: 'Home'.
         html break.
         html anchor
            callback: [mainArea := LBScheduleComponent new];
            with: 'Events'.
      ].
```

f. View the application in a web browser and confirm that you can switch between the image and the schedule.

7. The ‘renderMainOn:’ method in LBMain includes a conditional that hints for the need of a refactoring. Now that we have one subcomponent, we might as well have more.

    a. Create a new component:

```
WAComponent subclass: #LBHome
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'LosBoquitas'
```

    b. Add a render method to LBHome with code from LBMain>>#’renderMainOn:’.

```
renderContentOn: html

   html image
      altText: 'children playing soccer';
      url: LBFileLibrary / 'boquitas.jpg';
      yourself.
```

    c. LBHome is done. Now we will go back and add an initialize method to LBMain to use our new component.

```
initialize

   super initialize.
   mainArea := LBHome new.
```

    d. Now we can simplify LBMain>>#’renderMainOn:’ considerably by always rendering a subcomponent instead of having conditional code:

```
renderMainOn: html

   html div
      id: 'main';
      class: 'section';
      with: [html render: mainArea].
```


e. Finally, we need to modify one line of the sidebar menu creation to use our new component.

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         html anchor
            callback: [mainArea := LBHome new];
            with: 'Home'.
         html break.
         html anchor
            callback: [mainArea := LBScheduleComponent new];
            with: 'Events'.
      ].
```

f. Return to a web browser and start the application over from http://localhost:8080/boquitas. You should be able to switch back and forth between the home page and the schedule.

8. We would like to be able to edit events. We will start with deleting an event.

   a. Add LBScheduleComponent>>#'actionReportColumn'.

```
actionReportColumn

   ^WAReportColumn new
      title: 'Action';
      valueBlock: [:anEvent | 'delete'];
      clickBlock: [:anEvent | self delete: anEvent];
      yourself.
```


b. Edit LBScheduleComponent>>#'initialize' to use the new column. Note that instead of using the instance creation message 'with:with:with:with:' on Array, we are using an OrderedCollection and adding items to it. This is because once you reach more than four items, there might not be a class-side method that accepts enough arguments. Also note that this is an example of where the 'yourself' message at the end of the cascade is not just cosmetic but is necessary since the 'add:' method returns the argument (a column) rather than the receiver (an OrderedCollection).

```
initialize

   | columns |
   super initialize.
   columns := OrderedCollection new
      add: self whoReportColumn;
      add: self whatReportColumn;
      add: self whenReportColumn;
      add: self whereReportColumn;
      add: self actionReportColumn;
      yourself.
   listComponent := WATableReport new
      columns: columns;
      rowPeriod: 1;
      yourself.
```

c. If you return to your web browser and refresh, the new column will likely not appear. This is because the component is still holding an instance of WATableReport that was initialize with only four columns. To see the new table you need to click on the 'Events' link to install a new component.

d. If you click on a <delete> link now, you should get a MessageNotUnderstood error because we have not implemented the #delete: method in LBScheduleComponent. Add the following method (and note how we are not doing any SQL or other database related activity):

```
delete: anEvent

   LBEvent events remove: anEvent.
```

e. Try refreshing your web browser and note that one of the events has been removed.

9. Before you delete all the events, we will add a confirm dialog using JavaScript. Edit LBScheduleComponent>>#'actionReportColumn' to change how the 'Delete' anchor is generated. Note that having the column definition in its own method means that we don't have to edit a big method to make this change. The definition of the column is encapsulated in a single method and does not share the method with another definition.

```
actionReportColumn

   ^WAReportColumn new
      title: 'Action';
      valueBlock: [:each :html |
         html anchor
            onClick: 'return confirm(''Are you sure?'')';
            callback: [self delete: each];
            with: 'delete'.
      ];
      yourself.
```

   Here we are creating an anchor and giving it JavaScript for the 'onclick' event. The JavaScript code will run before the link is followed, and if the JavaScript returns false the new page is not requested.

   In order for this change to be visible, you must recreate the component. A simple way to do this is to click the 'Home' link and then click the 'Events' link.

   After deleting events, recreate them by evaluating 'LBEvent createEvents' in a workspace or using the instructions at step 2d above.

10. Save your Pharo image and quit.

In this chapter we look at creating an HTML form element to collect information from the user.

1. Define a new component to edit events.

```
WAComponent subclass: #LBEventEditor
   instanceVariableNames: 'model'
   classVariableNames: ''
   category: 'LosBoquitas'
```

   a. Add a render method to show that we are displaying the component.

```
renderContentOn: html

   html heading: self class name.
```

   b. Create an initialize method on the instance side.

```
initialize: anEvent

   self initialize.
   model := anEvent.
```

   c. Now create an instance creation method *on the class side*. Note that we are calling 'basicNew' rather than 'new.' This is because 'new' would call 'initialize' on the instance, and we want to call 'initialize:' explicitly and let it call 'initialize.'

```
on: anEvent

   ^self basicNew
      initialize: anEvent;
      yourself.
```

2. Now we can return to the LBScheduleComponent and arrange to call our new editor.

   a. Add an 'edit:' method to the instance side of LBScheduleComponent.

```
edit: anEvent

   | editor answer |
   editor := LBEventEditor on: anEvent.
   answer := self call: editor.
   answer
      ifTrue: [self inform: 'Edits were saved']
      ifFalse: [self inform: 'Edits were cancelled'].
```


b. Modify LBScheduleComponent >>#'whatReportColumn' so that the column has a click block. Note that since we have the column definition in its own method we don't have to modify a large initialize method.

```
whatReportColumn

   ^WAReportColumn new
      title: 'What';
      selector: #what;
      clickBlock: [:each | self edit: each];
      yourself.
```

c. In your web browser, click on the <Events> link to show that the 'what' field as a link. Click on any row and see that the schedule list is replaced with the event editor component (which simply displays some text).

3. Add true editing to the editor.

a. Modify LBEventEditor>>#renderContentOn: to lay out a table with headings and input fields. (Yes, we are using a table for formatting; the next step will refactor this method to avoid using a table.) Note that for this first round we are using only text fields so have set the 'when' field to be read only.

```
renderContentOn: html

   html form: [
      html table: [
         html tableBody: [
            html tableRow: [
               html tableHeading: 'Who:'.
               html tableData: [
                  html textInput
                     value: model who;
                     callback: [:value | model who: value].
               ].
            ].
            html tableRow: [
               html tableHeading: 'What:'.
               html tableData: [
                  html textInput
                     value: model what;
                     callback: [:value | model what: value].
               ].
            ].
            html tableRow: [
               html tableHeading: 'When:'.
               html tableData: [
                  html textInput
                     value: model when printString;
```



```
                    yourself.
                ].
            ].
            html tableRow: [
                html tableHeading: 'Where:'.
                html tableData: [
                    html textInput
                        value: model where;
                        callback: [:value | model where: value].
                ].
            ].
            html tableRow: [
                html tableData: [
                    html cancelButton
                        callback: [self answer: false];
                        with: 'Cancel'.
                ].
                html tableData: [
                    html submitButton
                        callback: [self answer: true];
                        with: 'Save'.
                ].
            ].
        ].
    ].
].
```

b. Try this component in your web browser. It should be possible to edit the fields and save or cancel the edits. Cancelled edits should not be persisted.

c. Note how a table is used to lay out the form. Historically, this was a fairly typical approach because it allows labels and data entry fields to be positioned relatively nicely. More recently the recommendation has been to use CSS rather than tables to handle layout (see http://en.wikipedia.org/wiki/Tableless_web_design). We tackle that challenge next.

4. Refactor the editing to avoid the use of a table.

    a. Add small methods to edit each field.

```
renderWhoOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'Who:'.
      html textInput
         id: tagID;
         value: model who;
         callback: [:value | model who: value].
   ].
```

```
renderWhatOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'What:'.
      html textInput
         id: tagID;
         value: model what;
         callback: [:value | model what: value].
   ].
```

```
renderWhenOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'When:'.
      html textInput
         id: tagID;
         value: model when;
         yourself.
   ].
```



```
renderWhereOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'Where:'.
      html textInput
         id: tagID;
         value: model where;
         callback: [:value | model where: value].
   ].
```

```
renderButtonsOn: html

   html div: [
      html cancelButton
         callback: [self answer: false];
         with: 'Cancel'.
      html submitButton
         callback: [self answer: true];
         with: 'Save'.
   ].
```

b. Modify 'renderContentOn:' to call the new methods.

```
renderContentOn: html

   html form
      class: 'eventEditor';
      with: [
         self
            renderWhoOn: html;
            renderWhatOn: html;
            renderWhenOn: html;
            renderWhereOn: html;
            renderButtonsOn: html;
            yourself.
      ].
```

c. View this in a browser and observe that the layout has each element on a new line. Now we can edit the CSS to make this a bit more fancy. Add the following lines to the text in LBFileLibrary>>#'boquitasCss' inside the existing string (i.e., after the first single quote character and before the last single quote character).

```
.eventEditor { display: table; }
.eventEditor > div { display: table-row; }
.eventEditor > div > * { display: table-cell; }
```


d. Refresh the page in your web browser, and note that the positioning is now controlled by the CSS. We have separated the text markup (HTML) from the style (CSS). This is considered a much better way to build web sites, but does rely on some CSS features that might not be supported in older browsers. For example, Internet Explorer 7 (and earlier) does not recognize table formatting.

   Of course, once we start down the path of separating content from style we need to learn CSS and how it interacts with HTML.

e. Notice that the label width is unusually wide. It turns out that this is because the buttons in the fifth row are lumped together in the first column.

f. To move the buttons to the second column, add an empty label before the buttons.

```
renderButtonsOn: html

   html div: [
      html label: [html space].
      html cancelButton
         callback: [self answer: false];
         with: 'Cancel'.
      html submitButton
         callback: [self answer: true];
         with: 'Save'.
   ].
```

g. We intended that each element inside a div inside the eventEditor would be treated as a table-cell. It turns out that the CSS specification (see http://www.w3.org/TR/CSS21/conform.html#conformance) allows browsers to ignore

CSS properties for form controls (including input fields like buttons): “CSS 2.1 does not define which properties apply to form controls and frames, or how CSS can be used to style them.”

h. If you want the buttons to be in separate columns, enclose them in another element, such as span.

```
renderButtonsOn: html

   html div: [
      html span: [
         html cancelButton
            callback: [self answer: false];
            with: 'Cancel'.
      ].
      html span: [
         html cancelButton
            callback: [self answer: true];
            with: 'Save'.
      ].
   ].
```

5. Adding a new event.

   a. Modify LBScheduleComponent>>#renderContentOn: to add an <Add> link.

```
renderContentOn: html

   listComponent rows: LBEvent events asSortedCollection.
   html render: listComponent.
   html anchor
      callback: [self add];
      with: 'Add'.
```

   b. Try it out and note that you get a walkback because the add method is not implemented.

   c. Add the following method:

```
add

   | event editor |
   event := LBEvent new.
   editor := LBEventEditor on: event.
   (self call: editor) ifTrue: [
      LBEvent events add: event.
   ].
```

   d. Refresh your browser and try adding an event. Try opening the editor but cancelling the new event.

e. Note how we are reusing a component—to add and to edit. The component doesn't know how it is being used which provides for good encapsulation.

f. Note also that the answer is useful in this case. If the user pressed the Cancel button, we don't want to add the new event. Before going further let's cleanup LBScheduleComponent>>#'edit:' so that we don't alert the user to whether the 'Cancel' or 'Save' button was clicked. Note how much simpler the method is now.

```
edit: anEvent

   self call: (LBEventEditor on: anEvent).
```

6. Edit 'when' with WADateTimeSelector. We left the 'when' field as read only since we are storing an instance of DateAndTime (rather than an instance of String). Let's give this editor some more usability.

   a. As we discovered in Chapter 7, Seaside provides a number of sample components that can be used to present typical information on a web page. Change the schema for LBEventEditor to add another instance variable.

```
WAComponent subclass: #LBEventEditor
   instanceVariableNames: 'model dateTimeSelector'
   classVariableNames: ''
   category: 'LosBoquitas'
```

   b. Modify the 'initialize:' method to obtain a new component and set its initial value.

```
initialize: anEvent

   self initialize.
   model := anEvent.
   dateTimeSelector := WADateTimeSelector new
      dateAndTime: model when;
      yourself.
```

   c. Modify the 'renderWhenOn:' method to use the new component. The new component is enclosed in a span element so that the label can be associated with the component.

```
renderWhenOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'When:'.
      html span
         id: tagID;
         with: [html render: dateTimeSelector].
   ].
```


d. Now we need some way to get the value out of the component in case the user changed the value. Since we are simply rendering a subcomponent, we don't have a 'callback:' that we can add to it. Instead, we need to do something when the 'Save' button is clicked. Modify the 'renderButtonsOn:' method to call a new 'save' method.

```
renderButtonsOn: html

   html div: [
      html span: [
         html cancelButton
            callback: [self answer: false];
            with: 'Cancel'.
      ].
      html span: [
         html submitButton
            callback: [self save];
            with: 'Save'.
      ].
   ].
```

e. Add the new 'save' method.

```
save

   model when: dateTimeSelector dateAndTime.
   self answer: true.
```

7. Edit 'who' with a drop-down list. Often we want to constrain the value of a field to something taken from a list. This will demonstrate how to do that.

a. Add a method to LBEvent (a different class) that returns a list of allows values for 'who.'

```
whoList

   ^#('players' 'family' 'guests' 'staff').
```

b. Modify LBEvent>>#'initialize' to use the new list.

```
initialize

   super initialize.
   who := self whoList first.
   what := 'practice'.
   when := DateAndTime noon.
   where := 'field'.
```


c. Now return to LBEventEditor and edit 'renderWhoOn:' so that we create a <select> element with a series of <option> elements (view the source if you are curious).

```
renderWhoOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'Who:'.
      html select
         id: tagID;
         selected: model who;
         list: model whoList;
         callback: [:value | model who: value].
   ].
```

8. Edit 'what' with a single-select list.

   a. Add a method to LBEvent that returns a list of allows values for 'what.'

```
whatList

   ^#('practice' 'registration' 'game' 'staff meeting' 'party').
```

   b. Modify LBEvent>>#'initialize' to use the new list.

```
initialize

   super initialize.
   who := self whoList first.
   what := self whatList first.
   when := DateAndTime noon.
   where := 'field'.
```


c. Now return to LBEventEditor and edit 'renderWhatOn:' so that we create a <select> element with a series of <option> elements (view the source if you are curious). Note that the only difference from a drop-down list is that the size is specified and is greater than one.

```
renderWhatOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'What:'.
      html select
         id: tagID;
         selected: model what;
         list: model whatList;
         size: 4;
         callback: [:value | model what: value].
   ].
```

9. Edit 'where' with a multi-line text area.

   a. Modify 'renderWhereOn:' to replace the textInput with a textArea.

```
renderWhereOn: html

   | tagID |
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'Where:'.
      html textArea
         id: tagID;
         value: model where;
         callback: [:value | model where: value];
         yourself.
   ].
```

   b. Modify the CSS to make the field larger. Edit LBFileLibrary>>#'boquitasCss' to add the following line. (The width will vary depending on your browser's selection of a font for the text area. Because of this it might be better to use a pixel width.)

```
.eventEditor textarea { height: 4em; width: 30em; }
```


10. Make ‘when’ more readable in the table.

    a. Add a method to LBEvent to return a more readable version of the when value.

```
whenString

   ^when asDate printString , ' ' , when asTime printString.
```

    c. Modify LBScheduleComponent>>#’whenReportColumn’ to use the new method.

```
whenReportColumn

   ^WAReportColumn new
      title: 'When';
      selector: #when String;
      clickBlock: nil;
      yourself.
```

11. In order to demonstrate checkboxes, radio buttons, and some JavaScript interaction with CSS, we will add an attribute to LBEvent to keep track of whether a game is home or away.

    a. Add ‘gameType’ as an instance variable to LBEvent. We will treat this value as a three-state flag: nil (for ‘not a game’), #‘home’ (a Symbol, or string singleton), and #‘away’ (also a Symbol). The initial value is nil.

```
Object subclass: #LBEvent
   instanceVariableNames: 'who what when where gameType'
   classVariableNames: ''
   category: 'LosBoquitas'
```

    b. Using the class refactoring menu, add an accessor for the new variable.

    c. We will have three form elements (a checkbox on one line and two radio buttons on another line) to capture this data (three radio buttons would be more efficient, but this gives a good demo!). Because of the way Seaside processes the callbacks associated with these fields, we won’t simply assign a value to the model during any one callback. Instead we will have two instance variables in the editor that capture various pieces of state that we will merge as part of the save process. To do that, add ‘isGame’ and ‘gameType’ to the definition of LBEventEditor.

```
WAComponent subclass: #LBEventEditor
   instanceVariableNames: 'model dateTimeSelector isGame gameType'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LosBoquitas'
```

    d. Modify LBFileLibrary>>#’boquitasCss’ to add a line allowing a element to be hidden if it has a class attribute of ‘hidden.’

```
.eventEditor div.hidden { display: none; }
```


e. Return to LBEventEditor and modify 'renderContentOn:' to call a couple new methods (the new messages will show as red since the methods have not been defined yet).

```
renderContentOn: html

   html form
      class: 'eventEditor';
      with: [
         self
            renderWhoOn: html;
            renderWhatOn: html;
            renderWhenOn: html;
            renderWhereOn: html;
            renderIsGameOn: html;
            renderGameTypeOn: html;
            renderButtonsOn: html;
            yourself.
      ].
```

f. Add 'renderIsGameOn:' to LBEventEditor. Note that this method has JavaScript code that is added to the checkbox. The JavaScript finds the element created below and changes its class depending on whether the checkbox is checked or not. Based on the class, the CSS defined above will be applied.

```
renderIsGameOn: html

   | script tagID |
   script := "Workaround for IE bug (thanks to Stephan Eggermont)"
      'document.getElementById("idGameType").setAttribute("class",' ,
         'this.checked? "":"hidden");' ,
      'document.getElementById("idGameType").setAttribute("className",' ,
         'this.checked? "":"hidden");'.
   html div: [
      html label
         for: (tagID := html nextId);
         with: 'Is Game:'.
      html checkbox
         id: tagID;
         value: model gameType notNil;
         callback: [:value | isGame := value];
         onClick: script;
         yourself.
   ].
```


g. Add 'renderGameTypeOn:' to LBEventEditor. Note that the HTML class attribute of the div is set to 'hidden' or nil depending on whether gameType is nil. This div element will have its class changed by the JavaScript code above.

```
renderGameTypeOn: html

   | tagID group |
   html div
      id: 'idGameType';
      class: (model gameType isNil ifTrue: ['hidden'] ifFalse: [nil]);
      with: [
         html label
            for: (tagID := html nextId);
            with: 'Type:'.
         html span
            id: tagID;
            with: [
               group := html radioGroup.
               html radioButton
                  id: (tagID := html nextId);
                  group: group;
                  selected: model gameType ~= #'away';
                  callback: [gameType := #'home'].
               html label
                  for: tagID;
                  with: 'Home'.
               html radioButton
                  id: (tagID := html nextId);
                  group: group;
                  selected: model gameType = #'away';
                  callback: [gameType := #'away'].
               html label
                  for: tagID;
                  with: 'Away'.
            ]].
```

12. Try out the various combinations and note how the game type is displayed and hidden based on the checkbox value (you may need to refresh your browser a few times to reload the CSS). This demonstrates the use of JavaScript in Seaside. Note, however, that the value is not saved (thanks to Stephan Eggermont for noticing this and providing the fix). Edit LBEventEditor>>#'save' as follows.

```
save

   model when: dateTimeSelector dateAndTime.
   model gameType: (isGame == true ifTrue: [gameType] ifFalse: [nil]).
   self answer: true.
```

13. Try out the various combinations and then save your Pharo image.

Many web sites and web applications limit access based on a user. We will demonstrate this functionality by identifying some functionality that is available only to registered users.

1. Create a class to model the user.

    a. Create the LBUser class with an id, a name, and a password.

```
Object subclass: #LBUser
   instanceVariableNames: 'id name password'
   classVariableNames: ''
   category: 'LosBoquitas'
```

    b. Create accessors using the class refactoring menu (accepting all the proposed new methods). Note that one of the created methods is 'name1' (to return the 'name' instance variable). This happened because there was already a 'name' method (here it happens to be in a superclass), and the refactoring tool did not want to override the existing method. We do want to override the method, so add a 'name' method.

```
name

   ^ name.
```

    c. To remove the 'name1' method, select 'name1' in the method list, right-click, and select 'remove method…'. If there are any senders of 'name1' you will be asked to confirm the delete.

    d. In general, it is considered a poor practice to store passwords. Instead, applications should store some sort of one-way encryption of the password. To do this, modify the 'password:' method that was generated so that instead of storing the passed-in string, we store a hash of the string. This is (only) a little bit more secure than a free-text string.

```
password: anObject

   password := anObject hash.
```

    e. Add a method to verify passwords.

```
verifyPassword: aString

   ^aString hash = password.
```

    f. Add a method to initialize the values of the instance variables.

```
initialize

   super initialize.
   id := ''.
   name := ''.
   password := 0.
```


g. Add a method to support sorting.

```
<= aUser

   ^self id <= aUser id.
```

h. Define a class instance variable to hold a cache of users (this is similar to what we did on LBEvent to cache events). Make sure that LBUser is selected, and then click on the 'class' button below the class list. This will replace the class definition with a place to define class instance variables.

```
LBUser class
   instanceVariableNames: 'users'
```

i. Add the following *class-side* method to return the user list.

```
users

   users isNil ifTrue: [
      users := IdentitySet with: (self new
         id: 'admin';
         name: 'Site Administrator';
         password: 'passwd';
         yourself).
   ].
   ^users.
```

j. Add the following *class-side* method to lookup a user.

```
userWithID: idString password: passwordString

   ^self users
      detect: [:each | each id = idString and:
         [each verifyPassword: passwordString]]
      ifNone: [nil].
```

2. In LBMain class>>#'initialize' (the initialize method on the class-side of LBMain) we define our application to use WASession. This is an object that holds various session information that is available to all components. We would like to hold a user as part of the session, so we will create a subclass that has an additional instance variable.

a. Define LBSession with 'user' as an instance variable.

```
WASession subclass: #LBSession
   instanceVariableNames: 'user'
   classVariableNames: ''
   category: 'LosBoquitas'
```

b. Create instance variable accessors using the class refactoring menu.

c. Modify LBMain class>>initialize to use this new session class.


```
initialize
"
   LBMain initialize.
"
   super initialize.
   (self registerAsApplication: 'boquitas')
      preferenceAt: #sessionClass put: LBSession;
      yourself.
```

d. Initialize LBMain so that it uses the new class. You can click anywhere on the third line of the method which contains the LBMain initialize comment and press <Ctrl>+<d> (for 'do-it').

3. Create a login component that we can use.

a. Create a class 'LBLoginComponent' with two instance variables, 'userID' and 'password.'

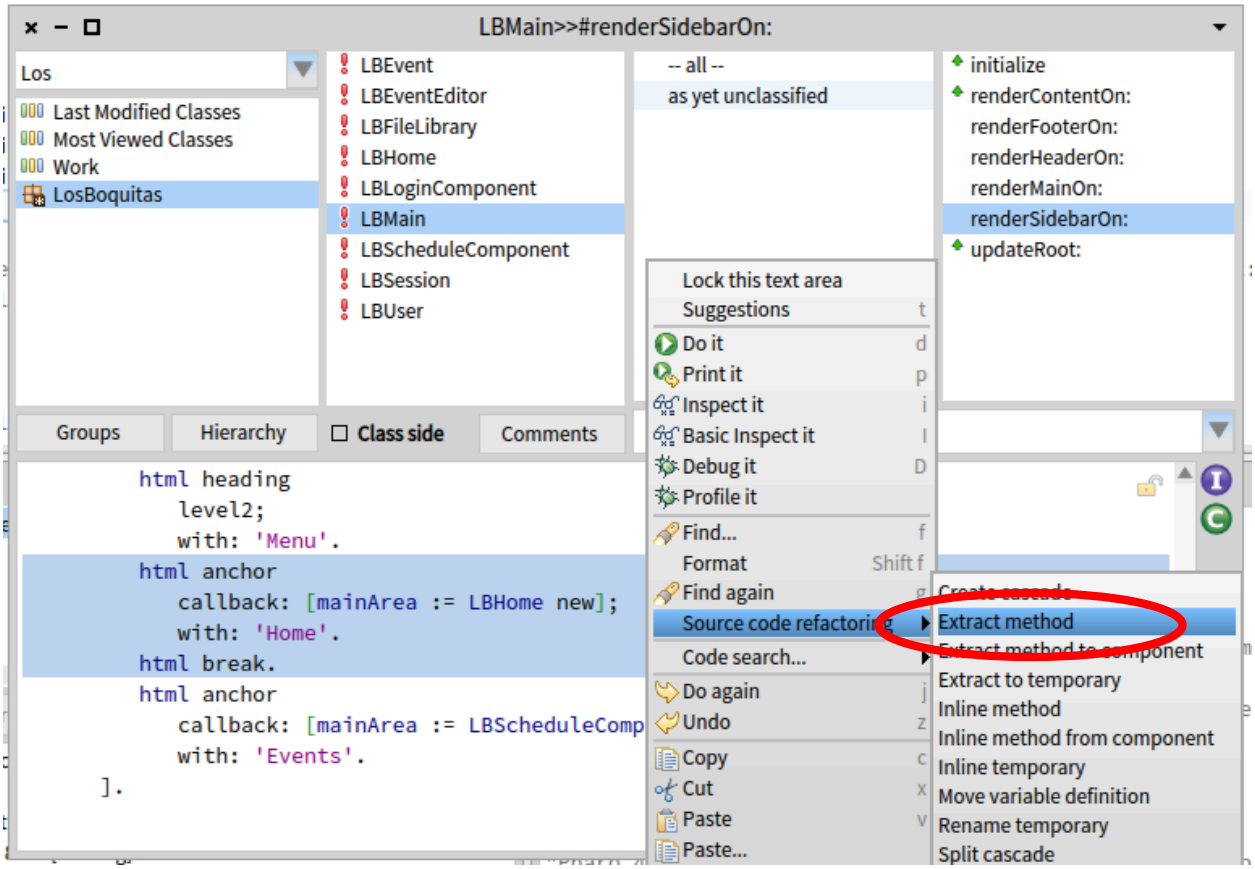
```
WAComponent subclass: #LBLoginComponent
   instanceVariableNames: 'userID password'
   classVariableNames: ''
   category: 'LosBoquitas'
```

b. Add a render method to show that it is being called.

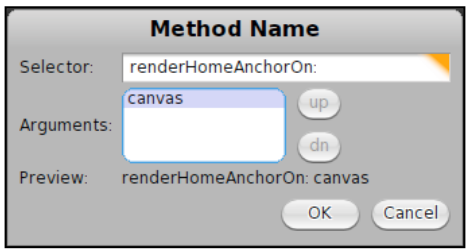
```
renderContentOn: html

   html heading: self class name.
```


4. Refactor LBMain>>#'renderSidebarOn:' so that we have more small methods rather than a few large methods. This is a much-favored practice in the Smalltalk community.

    a. Select the four lines of code defining the home anchor in the class 'LBMain' and the method #'renderSidebarOn:'. Right-click after selecting the code and select 'refactor source' and then 'extract method.'

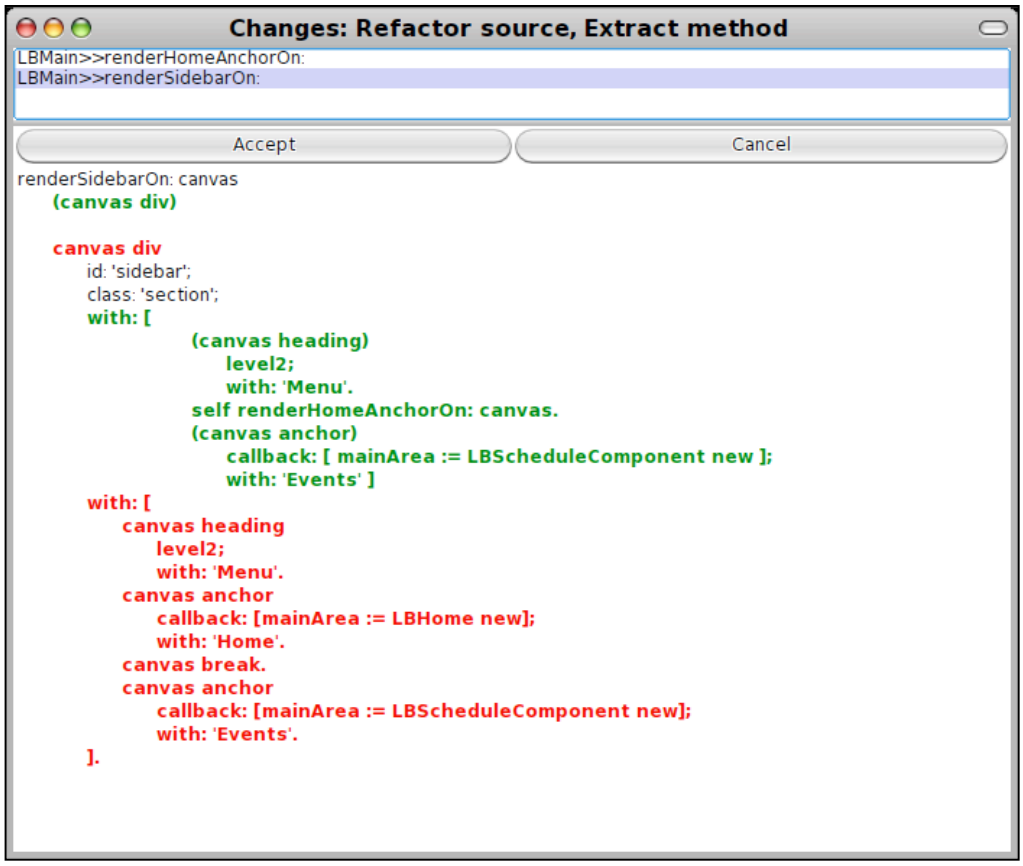


    b. This will pop up a dialog asking for a new name for the method. Enter 'renderHomeAnchorOn: html' as the text and click 'OK' or press <Enter>.



    c. This will show a Changes dialog on the ExtractMethodRefactoring where you can see two methods involved. One is a new method ('renderHomeAnchorOn:') and the other is a modified method ('renderSideBarOn:'). If you select the modified method you can see the current code (in red type) and code that will be installed if you click the 'accept'

button (in green). Note that the refactoring will change the formatting somewhat and overstates the extent of the changes. Go ahead and click ‘accept’.



d. In a similar manner, extract the ‘Events’-link creation code. Note that the refactoring changed the placement of the square brackets in the method. This means that we can no longer select full lines, but must select through the ‘yourself’ but not the closing square bracket. Extract this code into a new method named ‘renderEventsAnchorOn: html’ and accept the changes.

```
renderSidebarOn: html
   html div
      id: 'sidebar';
      class: 'section';
      with: [
               html heading
                  level2;
                  with: 'Menu'.
               self renderHomeAnchorOn: html.
               html anchor
                  callback: [ mainArea := LBScheduleComponent new ];
                  with: 'Events' ]
```


5. Add a 'Login' anchor.

    a. First, edit 'renderEventsAnchorOn:' to add a break at the end so that subsequent elements are on a new line.

```
renderEventsAnchorOn: html

   html anchor
      callback: [mainArea := LBScheduleComponent new];
      with: 'Events'.
   html break.
```

    b. Next, create a new method very similar to the above to render the login link.

```
renderLoginAnchorOn: html

   html anchor
      callback: [mainArea := LBLoginComponent new];
      with: 'Login'.
   html break.
```

    c. Finally, modify 'renderSidebarOn:' to call our new method (and get back our formatting).

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         self
            renderHomeAnchorOn: html;
            renderEventsAnchorOn: html;
            renderLoginAnchorOn: html;
            yourself.
      ].
```

    d. In a web browser, navigate to the application and note the new link. Clicking on the link should show the login component.

6. Add a login form to the login component.

    a. Add various render methods to LBLoginComponent. Note that we can set focus to a particular field using explicit JavaScript. It would be more elegant to use a library (such as jQuery), but this demonstrates the general capability.

```
renderUserOn: html

   | htmlID |
   html div: [
      html label
         for: (htmlID := html nextId);
         with: 'User:'.
      html textInput
         id: htmlID;
         value: userID;
         callback: [:value | userID := value];
         script: 'document.getElementById(' ,
            htmlID printString , ').focus()';
         yourself.
   ].
```

```
renderPasswordOn: html

   | htmlID |
   html div: [
      html label
         for: (htmlID := html nextId);
         with: 'Password:'.
      html passwordInput
         id: htmlID;
         value: password;
         callback: [:value | password := value];
         yourself.
   ].
```

```
warning

   self session user notNil ifTrue: [
      ^'Logged in as ' , self session user name.
   ].
   (userID isNil or: [userID isEmpty]) ifTrue: [
      ^'Please enter User ID and Password'.
   ].

   ^'Login failed!'.
```



```
renderWarningOn: html

   html div: [
      html
         span: '';
         span: self warning;
         yourself.
   ].
```

```
renderSubmitOn: html

   html div: [
      html submitButton
         callback: [self login];
         with: 'Login'.
   ].
```

```
renderFormOn: html

   html form
      class: 'loginForm';
      with: [
         self
            renderUserOn: html;
            renderPasswordOn: html;
            renderWarningOn: html;
            renderSubmitOn: html;
            yourself.
      ].
```

```
renderContentOn: html

   self renderFormOn: html.
```

b. Return to the web browser and click the 'Login' link. Note that the formatting is not quite right since the text entry fields are not aligned.

c. Edit LBFileLibrary>>#'boquitasCss' so that the lines beginning with '.eventEditor' now begin with 'form' (referring to the element rather than the class). In this way CSS applies to both forms.

```
form { display: table; }
form > div { display: table-row; }
form > div > * { display: table-cell; }
form textarea { height: 4em; width: 40em; }
form div.hidden { display: none; }
```

d. Refresh the browser and note that the fields are now aligned in a table layout.

e. Clicking the 'Login' button should give an error since the 'login' method is not yet implemented. Add the following method to LBLoginComponent.

```
login

   | user |
   user := LBUser
      userWithID: userID
      password: password.
   self session user: user.
   user notNil ifTrue: [
      userID := nil.
      password := nil.
   ].
```

f. Now try the application again. If you give a wrong user ID/password, you should get a message displayed with that information. If you give a correct user ID/password ('admin' and 'passwd'), you should get a message identifying the logged in user.

g. Modify 'renderContentOn:' so that if a user is logged in we do not display the login form.

```
renderContentOn: html

   self session user isNil ifTrue: [
      self renderFormOn: html.
   ] ifFalse: [
      html heading: 'Welcome, ' , self session user name.
   ].
```


7. Modify LBMain to handle the presence of a session user.

    a. Instead of always rendering a login link, we need an alternative. Add a 'renderLogoutOn:' method.

```
renderLogoutAnchorOn: html

   html anchor
      callback: [self session user: nil];
      with: 'Logout ' , self session user name.
```

    b. Modify the 'renderLoginAnchorOn:' method to remove the break at the end.

```
renderLoginAnchorOn: html

   html anchor
      callback: [mainArea := LBLoginComponent new];
      with: 'Login'.
```

    c. Create a 'renderUserOn:' method that will call one or the other of the above methods.

```
renderUserOn: html

   self session user isNil ifTrue: [
      self renderLoginAnchorOn: html.
   ] ifFalse: [
      self renderLogoutAnchorOn: html.
   ].
   html break.
```

    d. Modify the 'renderSidebarOn:' method so that it calls the 'renderUserOn:' method instead of the 'renderLoginAnchorOn:' method.

```
renderSidebarOn: html

   html div
      id: 'sidebar';
      class: 'section';
      with: [
         html heading
            level2;
            with: 'Menu'.
         self
            renderHomeAnchorOn: html;
            renderEventsAnchorOn: html;
            renderUserOn: html;
            yourself.
      ].
```

    e. Try the application with various combinations of correct and incorrect passwords.

8. Restrict some features to logged-in users.

    a. Modify LBScheduleComponent>>#renderContentOn: as follows:

```
renderContentOn: html

   listComponent rows: LBEvent events asSortedCollection.
   html render: listComponent.
   self session user notNil ifTrue: [
      html anchor
         callback: [self add];
         with: 'Add'.
   ].
```

    b. Try the application when logged in and when not logged in. The ‘Add’ link should appear and disappear based on the user state.

    c. Modify LBScheduleComponent>>#initialize as follows:

```
initialize

   | columns |
   super initialize.
   columns := OrderedCollection new
      add: self whoReportColumn;
      add: self whatReportColumn;
      add: self whenReportColumn;
      add: self whereReportColumn;
      yourself.
   self session user notNil ifTrue: [
      columns add: self actionReportColumn.
   ].
   listComponent := WATableReport new
      columns: columns;
      rowPeriod: 1;
      yourself.
```

    d. Try the application when logged in and when not logged in. The ‘delete’ link should appear and disappear based on the user state.

9. Change the window title based on the subcomponent being viewed.

    a. As we discovered in chapter 4, the #updateRoot method provides a way to set the title of the web browser window (and/or tab) to something appropriate for the page being displayed. Now we have a main component (LBMain) and three subcomponents (LBHome, LBLoginComponent, and LBScheduleComponent). Add LBHome>>#updateRoot as follows:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: anHtmlRoot title , ' -- Home'.
```

    b. Add LBLoginComponent>>#updateRoot as follows:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: anHtmlRoot title , ' -- Login'.
```

    c. Add LBScheduleComponent>>#updateRoot as follows:

```
updateRoot: anHtmlRoot

   super updateRoot: anHtmlRoot.
   anHtmlRoot title: anHtmlRoot title , ' -- Event'.
```

    d. Try navigating to the various subcomponents and observe that the title does not change. This is because Seaside is rendering the page head element (which contains the page title element) before it discovers what components will be included in the page which are in the page body element. To provide a solution to this problem, Seaside inquires of each component for a list of 'children' before it starts rendering. This allows the subcomponents to participate more fully in preparing the page. Add LBMain>>#children as follows:

```
children

   ^super children , (Array with: mainArea).
```

    e. Now try navigating to the various subcomponents and observe that the title does change to match the current subcomponent.

    While the #children method is not necessary to process callbacks in Seaside 3.0 (as it was in earlier version of Seaside), it is considered good practice to do so. The absence of the #children method can cause subtle errors where components are displayed but seem to be ignored for some purposes.

10. Save your Pharo image.

So far we have identified the specific Seaside messages to create particular HTML constructs in an ad-hoc manner as needed for particular features. Now we will attempt a more systematic approach by looking at the various pieces of an HTML document and how Seaside generates the pieces.

An HTML document consists of a series of nested elements. An element can be a single, self-closing, tag (such as '
' for a line break) or a pair of tags (such as '<h1>Seaside</h1>' for a level-one heading) enclosing text and/or embedded elements. Depending on the tag type, the open tag may contain attributes that further describe the element (such as 'GLASS' where 'href' is the attribute name and the URL is the attribute value). The outermost element in an HTML document is the 'html' element, and it contains one 'head' and one 'body' element.

The primary role of any web framework is to generate HTML pages to be rendered on the user's browser. In Seaside, the HTML page is represented by one or more subclasses of WAComponent and methods may be implemented in the subclass to provide content to both the head and the body.

The component adds content to the page's head element by implementing a method 'updateRoot:' which is passed an instance of WAHtmlRoot. The head element must contain a 'title' element, so Seaside provides a default of 'Seaside' that can be overridden by sending 'title:' to the object passed to the 'updateRoot:' method. Other elements in the head are optional and include meta data and references to external CSS and JavaScript files (using the 'link' element). All components have an opportunity to add information to the page's head element before any of the body is rendered. The tree of components is identified by sending the message 'children' to the root component (so if you fail to implement the 'tree' method and include subcomponents, then the subcomponents will not have an opportunity to contribute to the head element).

The body element is the main place things are displayed on a HTML document. A component adds content to the page's body element by implementing a method 'renderContentOn:' which (by default) is passed an instance of WARenderCanvas. The component can add content to the page by sending messages to the html canvas received as an argument to this method.

To add text to a page, send a string as an argument with the 'text:' message to the html canvas. The string will be encoded properly so that special characters (such as '<' that would otherwise be interpreted as the beginning of a tag) are displayed on the browser. To add raw HTML code to a page, use the 'html:' message to avoid any encoding.

To add an element to a page, send a unary message to the html canvas identifying the element type desired, and then send messages to the resulting "brush" (representing the element) to (1) add attributes to the element and (2) embed content in the element. To embed content, send the 'with:' message providing as an argument either a block (that will be evaluated) or another object (that will be converted to a string, by default with 'displayString' via 'greaseString'). For example, to add an emphasis element with the text 'very' ('very'), you could use the following code.

```
canvas emphasis with: 'very'.
```


As a shortcut for cases (like the above) in which no attributes need to be set, there are a parallel set of keyword messages that take one argument that is passed with the 'with:' message.

```
canvas emphasis: 'very'.
```

Code blocks are generally used as the argument to a 'with:' message to embed other elements inside an element, though you could, of course, use a code block where text would be sufficient. The following is equivalent to the above two examples.

```
canvas emphasis: ['very'].
```

The HTML 4.01 Specification (http://www.w3.org/TR/REC-html40/) contains an index of 91 elements (http://www.w3.org/TR/REC-html40/index/elements.html), ten of which are deprecated. In Seaside, WARenderCanvas (and its superclass, WAHtmlCanvas) provides support for most of the non-deprecated elements through methods identified in the following table. As you can see, the Seaside approach is that a full message name is preferred over an abbreviation. This follows the general Smalltalk philosophy that code is more often read than written and spelling things out imposes less mental effort on the reader.

HTML Element Name	Seaside Message	Smalltalk Class (if not WAGenericTag)
A	anchor	WAAnchorTag
ABBR	abbreviated	
ACRONYM	acronym	
ADDRESS	address	
BIG	big	
BLOCKQUOTE	blockquote	
BR	break	WABreakTag
BUTTON	button	WAButtonTag
CAPTION	tableCaption	
CITE	citation	
CODE	code	
COL	tableColumn	WATableColumnTag
COLGROUP	tableColumnGroup	WATableColumnGroupTag
DD	definitionData	
DEL	deleted	WAEditTag
DFN	definition	
DIV	div	WADivTag
DL	definitionList	
DT	definitionTerm	
EM	emphasis	
FIELDSET	fieldSet	WAFieldSetTag
FORM	form	WAFormTag
H1	heading	WAHeadingTag
H2	heading	WAHeadingTag
H3	heading	WAHeadingTag
H4	heading	WAHeadingTag
H5	heading	WAHeadingTag

Name	Seaside	Description
H6	heading	WAHeadingTag
HR	horizontalRule	WAHorizontalRuleTag
IFRAME	iframe	WAIframeTag
IMG	image	WAImageTag
INPUT	cancelButton	WACancelButtonTag
	checkbox	WACheckboxTag
	fileUploader	WAFileUploadTag
	imageButton	WAImageButtonTag
	multiSelect	WAMultiSelectTag
	passwordInput	WAPasswordInputTag
	radioButton	WARadioButtonTag
	submitButton	WASubmitButtonTag
	textInput	WATextInputTag
INS	inserted	WAEditTag
KBD	keyboard	
LABEL	label	WALabelTag
LEGEND	legend	
LI	listItem	
MAP	map	WAImageMapTag
OBJECT	object	WAObjectTag
OL	orderedList	WAOrderedListTag
OPTION	option	WAOptionTag
OPTGROUP	optionGroup	WAOptionGroupTag
P	paragraph	
PARAM	parameter	WAParameterTag
PRE	preformatted	
Q	quote	
SAMP	sample	
SCRIPT	script	WAScriptTag
SELECT	select	WASelectTag
SMALL	small	
SPAN	span	
STRONG	strong	
SUB	subscript	
SUP	superscript	
TABLE	table	WATableTag
TBODY	tableBody	
TD	tableData	WATableDataTag
TEXTAREA	textArea	WATextAreaTag
TFOOT	tableFoot	
TH	tableHeading	WATableHeadingTag
THEAD	tableHead	
TR	tableRow	
TT	teletype	
UL	unorderedList	WAUnorderedListTag
VAR	variable	

There are several elements that Seaside does not directly support (but could be included using a generic tag by sending #'tag:').

- The following elements are deprecated in the HTML specification: applet, basefont, center, dir, font, isindex, menu, s strike, and u.
- The b and i elements are related to styling that should be done with CSS.
- The following elements are structural or appear only in the head section so are provided using other mechanisms: base, body, head, html, link, meta, style, and title.
- The remaining elements that have no direct Seaside support are the following: area, bdo, frame, frameset, noframes, and noscript.

Once you have identified a tag for an element, there might be attributes to set on that tag. The following core attributes are available on most elements by sending a message to the element object with a string argument:

- id (#'id:') – a document-wide unique identifier for the element. Because Seaside components can be embedded in other components, it is generally not a good idea to hard-code an identifier. When possible, send the message #'nextId' to the instance of WARenderCanvas passed to #'renderContentOn:' in order to get a unique identifier. Alternatively, you can send #'ensureId' to a brush.
- class (#'class:') – a space-separated list of class names. This attribute is primarily used to associate CSS style with an element.
- style (#'style:') – a style declaration for the element. It is considered a better practice to use an external style sheet.
- title (#'title:') – advisory information about the element. Visual browsers will often show this as a tool tip.

The following attributes are available on most elements:

- lang (#'language:') – the base language of an element's attribute values and text content.
- dir (#'direction:') – directionality of text. Acceptable values are 'ltr' for left-to-right text and 'rtl' for right-to-left text.

The following event-related attributes can be set on most elements and will trigger a script (typically JavaScript) in the browser:

- onclick (#'onClick:') – a pointer button was clicked.
- ondblclick (#'onDoubleClick:') – a pointer button was double clicked.
- onmousedown (#'onMouseDown:') – a pointer button was pressed down
- onmouseup (#'onMouseUp:') – a pointer button was released
- onmouseover (#'onMouseOver:') – a pointer was moved onto
- onmousemove (#'onMouseMove:') – a pointer was moved within
- onmouseout (#'onMouseOut:') – a pointer was moved away

- onkeypress (#'onKeyPress:') – a key was pressed and released
- onkeydown (#'onKeyDown:') – a key was pressed down
- onkeyup (#'onKeyUp:') – a key was released

Other attributes tend to be element-specific and can be found in the element's class (shown in the third column in the above table). Now we will look at some of the various classes and methods involved in the many steps from starting a web server to accepting a HTML request on a socket to sending an HTML response.

- The Seaside Control Panel (one if the three initial windows in the Seaside One-Click Experience) provides a graphical user interface over the default instance (singleton) of WAServerManager that manages a collection of instances of subclasses of WAServerAdapter.
- Initially, there is one adapter, an instance of WAComancheAdaptor, which is configured to listen on port 8080 and (by default) pass requests to the default instance (singleton) WADispatcher.
- WAComancheAdaptor>>#'basicStart' creates an HttpService on the provided port.
- HttpService>>#'runWhile:' creates a TcpListener that calls back to HttpService>>#'value:' with each newly accepted socket.
- HttpService>>#'serve:' creates an instance of HttpAdaptor and passes it the socket and itself.
- HttpAdaptor>>#'beginConversation' reads from the TCP socket and instantiates an instance of HttpRequest. This request is passed to HttpAdaptor>>#'dispatchRequest:' to get an instance of HttpResponse that is returned on the socket.
- HttpAdaptor>>#'dispatchRequest:' calls HttpService>>#'processHttpRequest:' on the instance first created above and it calls WAComancheAdaptor>>#'processHttpRequest:' on the instance first created above.
- WAComancheAdaptor>>#'processHttpRequest:' calls #'process:' on itself with the HttpRequest.
- WAServerAdaptor>>#'process:' creates an instance of WARequestContext (containing an instance of WARequest and WABufferedResponse).
- WAServerAdaptor>>#'handleRequest:' passes the instance of WARequest and to WADispatcher>>#'handleRequest:'. The result is an instance of WAResponse that is converted to an instance of HttpResponse that is eventually passed back to the client browser as a web page.

As described above, WAComancheAdaptor creates an instance of WARequest and passes it to the default WADispatcher that is responsible for finding someone to create a WAResponse to return to WAComancheAdaptor.

- There is a top-level dispatcher returned by the class-side method, #'default', which selects an instance of a subclass of WARequestHandler to receive the request. Of course, in order for an application to be found, an instance of WAApplication must have been registered. This is what is done when you send #'register:asApplicationAt:' to WAAdmin with your subclass of WAComponent and a string. The string you pass (such as 'hello') is used to define a path on your web server (such as 'http://localhost:8080/hello').

- The dispatcher sends the instance of WARequest to WAApplication>>#'handleRequest:' that looks for an instance of WASession to handle the request. The lookup is done using the '_s' field included in the URL or in a cookie (depending on a configuration setting). If the session key is not found (perhaps it expired and has been removed from the cache), is not provided, or has expired then a new session is created.
- WARegistry>>#'dispatch:to:' calls WASession>>#'handle:', which is implemented in WARequestHandler and eventually calls #'handleFiltered:' on itself (an instance of WASession or a subclass).
- The #'handleFiltered:' method looks at the passed-in URL fields for an 'action key' (the one with the key '_k'). If such a field is found in the request then the value is used as a lookup into the session's continuation dictionary. If there is a continuation available with the key, then it is evaluated with the request. We will look more at that below. First we look at the handling of an initial request.
- If the request does not include an 'action key', then the #'start:' message is sent to the session. If the request includes an 'action key' but there is no continuation for that key, then #'unknownRequest' is called which, by default, calls #'start:'.
- WASession>>#'start:' looks for the application's configuration preference for a 'mainClass' (by default, this is the class WARenderLoopMain) and calls #'start' on a new instance.
- WARenderLoopMain>>#'start' creates a new instance of the application's renderPhaseContinuationClass (by default, WARenderPhaseContinuation), and sends #'initialRequest:' to each visible presenter (the root component and its children). This is an opportunity to capture any parameters (e.g., to simulate a RESTful application) and redirect or configure the initial page based on these parameters.
- Once the initialRequest: messages have been sent, WARenderLoopMain>>#'start' calls #'captureAndInvoke' on the continuation. After some setup, WARenderPhaseContinuation>>#'processRendering:' creates the document (typically an instance of WAHtmlDocument).
- At this #'updateUrl:' is sent to each presenter. This gives them an opportunity to modify the base URL used when rendering.
- Next, #'updateRoot:' is sent to each presenter. This allows you to customize the <head> element of the HTML document. Typically, you would update the title and add CSS and JavaScript links here.
- Finally, #'renderWithContext:' is called on the root-level component (the one that was registered as the application). This uses WARenderVisitor to call your component's #'renderContentOn:' method.

If the request does not result in #'start:' being sent to the session, then we are dealing with a continuation created by an earlier page. Let's walk through how that continuation is created and then used.

- When rendering an HTML document, one might add an anchor element to the page and give the anchor a callback by sending #'callback:' to an instance of WAAnchorTag. For example:

```
renderLogoutAnchorOn: canvas

   canvas anchor
      callback: [self logout];
      with: 'Logout'.
```

- Sending the #'callback:' message will invoke WAAnchorTag>>#'callback:' which can be refactored as follows to allow easier discussion.

```
WAAnchorTag>>#callback: aNiladicValuable

      | callback id |
"4"   aNiladicValuable argumentCount > 0 ifTrue: [
"5"      GRInvalidArgumentCount signal: 'Anchors expect a niladic callback.'
"6"   ].
"7"   callback := WAActionCallback on: aNiladicValuable.
"8"   id := self storeCallback: callback.
"9"   self url addField: id.
```

- Line 4 checks to ensure that the block does not expect any arguments and line 5 provides an error in this case.
- Line 7 wraps the block in an instance of WAActionCallback.
- Line 8 saves the callback in an instance of WACallbackRegistry associated with the current session and returns a unique key.
- Line 9 adds the unique key to the URL that will be associated with the anchor tag.
- When the page is rendered, the instance of WAAnchorTag will add itself to the page as an <a> element with an 'href' attribute of the new URL. In the following URL, there is a parameter '4' that can be used to find the WAActionCallback holding the block.

```
http://localhost:8080/boquitas?_s=cDBBHwMjSSQFwGAD&_k=8k2j2bSW&4
```

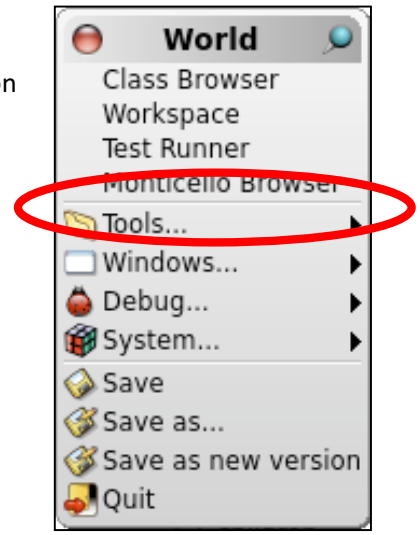
- When the user clicks on the anchor an HTTP GET request is submitted to the server. The server dispatches it to the application registered with the name 'boquitas'. The application then looks for a session with the key 'cDBBHwMjSSQFwGAD' and passes the request to the session. The session then looks for a 'continuation' with the key '8k2j2bSW' and passes the request to the instance of WAActionPhaseContinuation built during the earlier page rendering.
- WAActionPhaseContinuation>>#'runCallbacks' then invokes WACallbackRegistry>>#'handle:' to find and evaluate the appropriate callbacks based on the unique key added to the URL.

To learn more about this process, select one of the methods listed above and add 'self halt.' to the code.

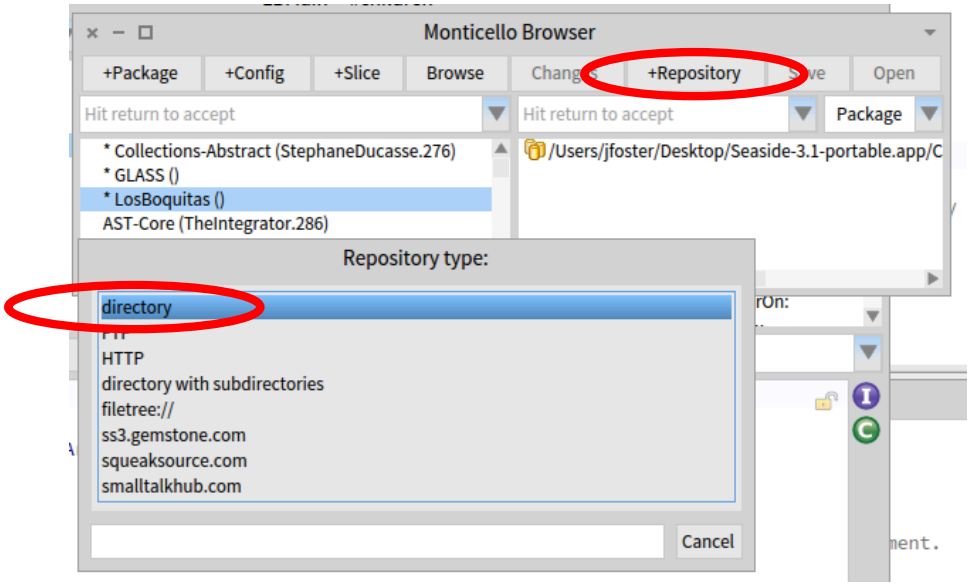
Monticello (pronounced either män-tə-`sel-ō or män-tə-`chel-ō) "is a distributed optimistic concurrent versioning system for Pharo code source. It was written by Avi Bryant and Colin Putney, with contributions from many members of the Pharo community" (see http://www.wiresong.ca/Monticello/). So far we have been saving our code by saving the Pharo image. Monticello gives us much more powerful tools for managing source code.

1. Create a Monticello package.

   Open your Pharo image that has the Los Boquitas application we have developed in earlier chapters. Open a Monticello Browser by left-clicking on the desktop and selecting 'Monticello Browser.'

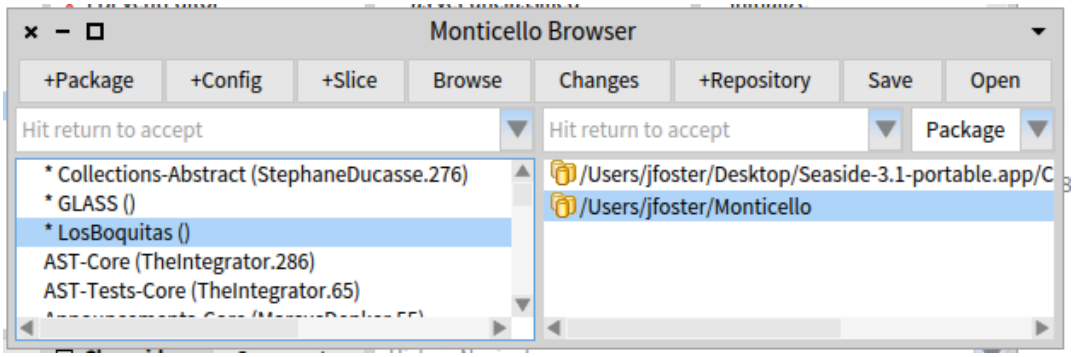


2. Create a Monticello Repository to hold the code.
   a. In your operating system create a directory to be used for Monticello packages. For now, consider something like 'C:\Monticello' or '/Monticello' since it will be easy to find. (If you don't want to clutter your root directory then you probably know how to pick a better spot!)
   b. With the LosBoquitas package selected, click on the '+Repository' button (center). This will open a menu with a list of repository types. For now we will use a directory repository. Click on 'directory.'

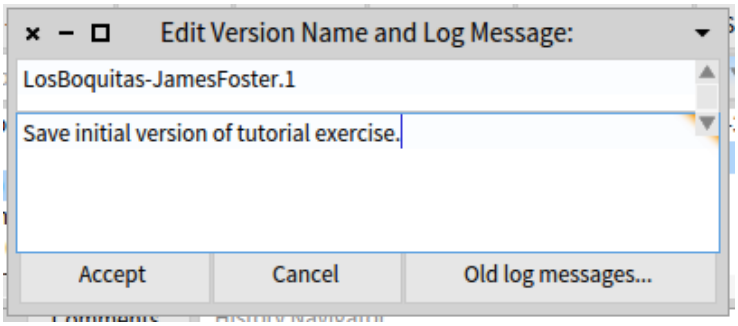


c. This will open a folder selection dialog in which you can select the directory you created above. In this example, I'm on a Macintosh and have selected '/Users/jfoster/Monticello' as my folder.

d. At this point the Monticello Browser should show the LosBoquitas package on the left and the directory repository on the right.

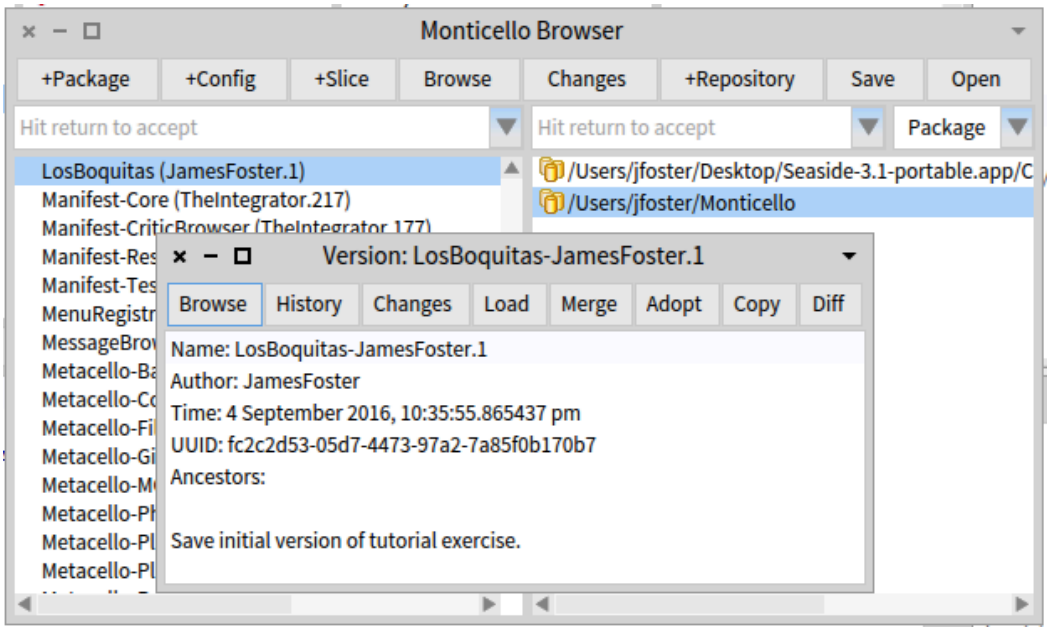


3. Save your code to the new repository.
   a. In the Monticello Browser, click the 'Save' button (the one in the center). This will bring up a dialog where you can edit the version name and a log message.

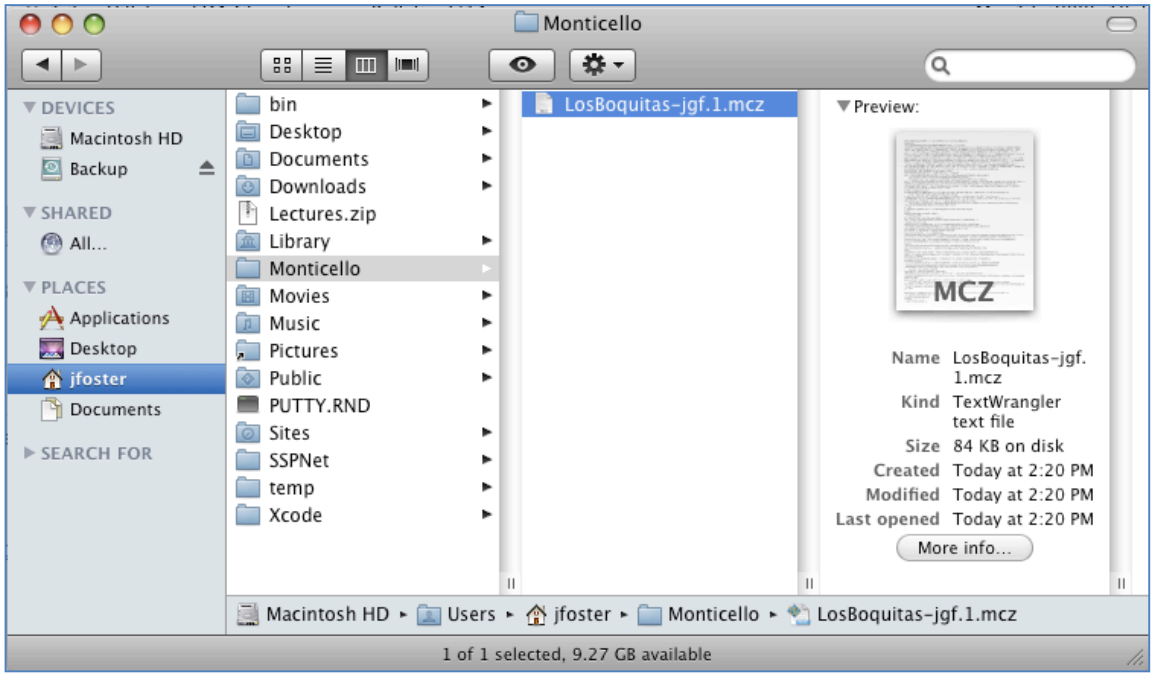


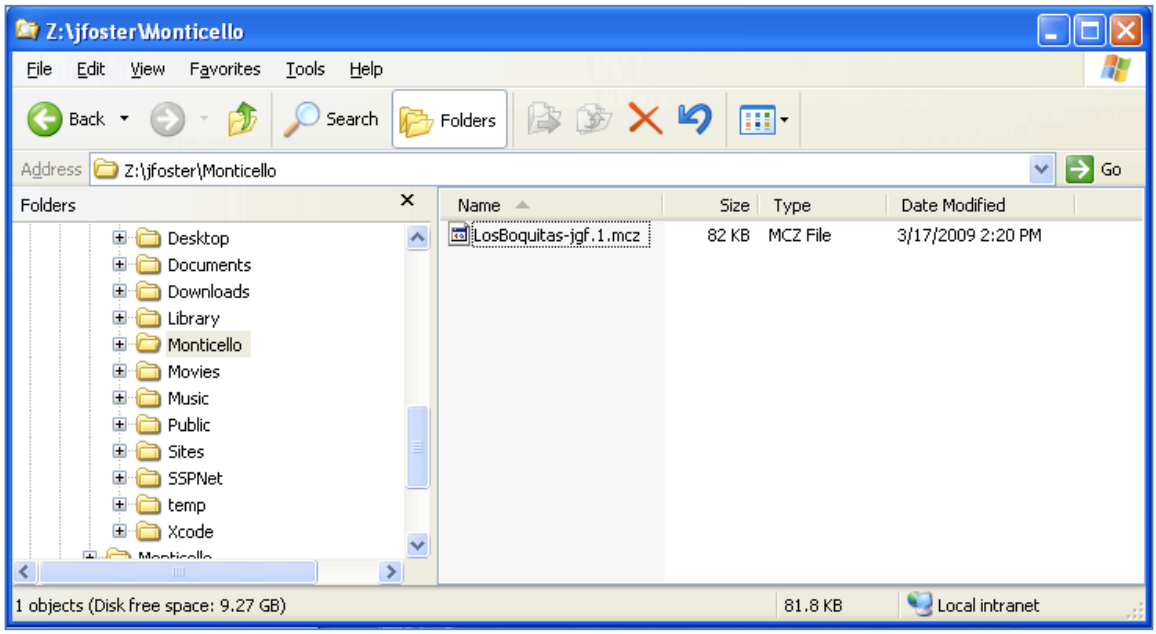
b. Verify that the version name begins with 'LosBoquitas-', has your name (not mine!), and ends with '.1' (the version number). Enter a log message in the text area and then click the 'Accept' button.

c. At this point you should see a Version window (shown in front here) and the Monticello Browser should be updated with the current version name in the package list (shown behind the version window in this screen shot).



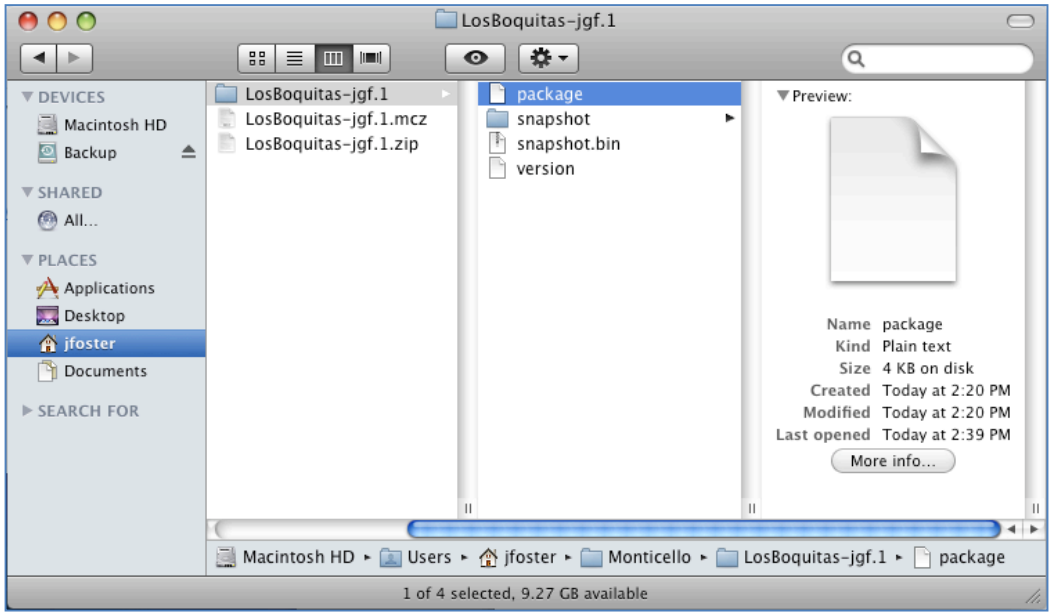
4. Using your native OS tools, navigate to the directory being used as the Monticello repository. You should see something like the following (depending on your operating system).





   a. Duplicate the MCZ file and rename it with a ZIP extension (e.g., 'LosBoquitas.zip').

b. Unzip the new file and view the contents. You should see three files and a directory in the new directory.



c. The 'package' file is a text file that contains the following:

```
(name 'LosBoquitas')
```

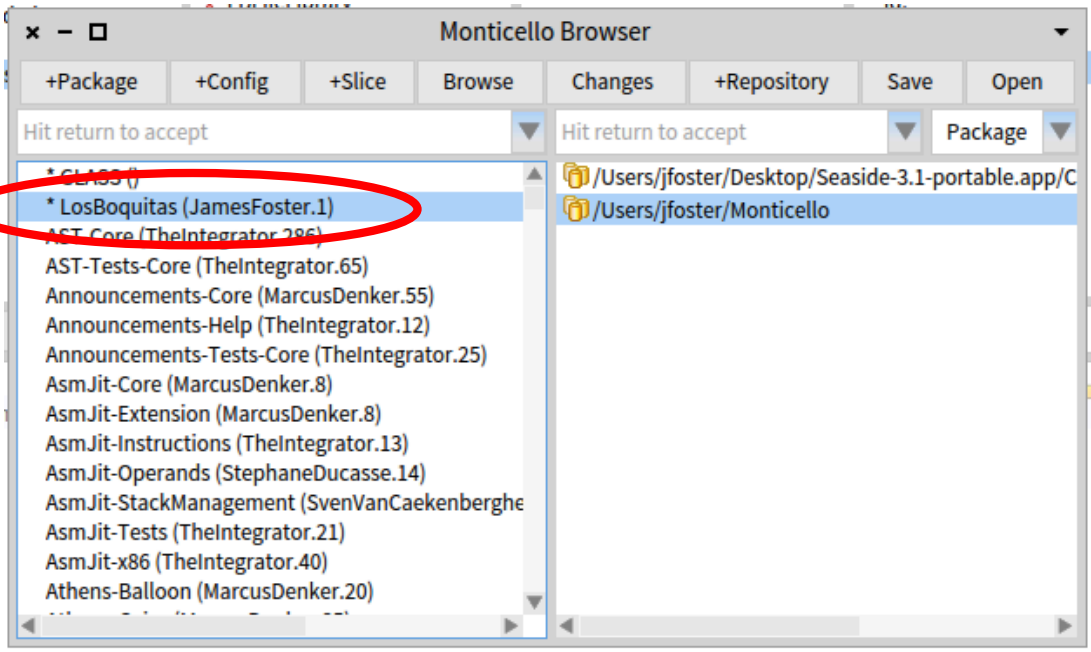
d. The 'snapshot.bin' file is a binary file used by the Monticello tools to load and compare packages.

e. The 'version' file is a text file containing a full version history of the package. In our case the history is quite simple. In the following, the formatting has been changed to make the text more readable.

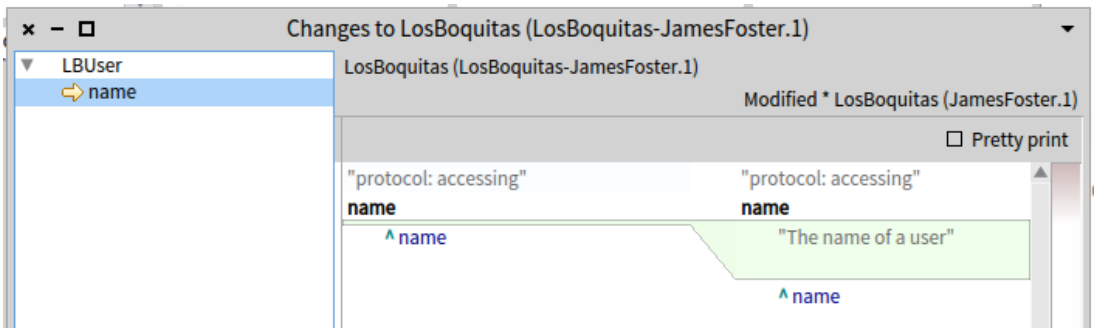
```
(
   name 'LosBoquitas-JamesFoster.1'
   message 'Save initial version of tutorial exercise.'
   id 'acc13957-a73b-4112-a1d5-9152062a7e08'
   date '12 February 2011'
   time '8:05:10 pm'
   author 'JamesFoster'
   ancestors ()
   stepChildren ()
)
```

f. The 'snapshot' directory contains a text file, 'sources.st' that is suitable for filing in to a Pharo image to generate the classes and methods in the package.

5. View changes using Monticello.
    a. Select a method in the LosBoquitas package and modify it. For this example, I've modified LBUser>>#'name' to add a comment. Note that the Monticello Browser now shows an asterisk next to the package name.



    b. With a package and repository selected, click on the 'Changes' button to open a Patch Browser. The saved package contents are shown in red type and the new package contents are shown in green type.



In subsequent chapters we will look at how Monticello can be used to transfer code from Pharo to GemStone.

So far we have been using Pharo's 'Seaside One-Click Experience' to learn Seaside and Smalltalk. Pharo is just one of several dialects of Smalltalk supporting Seaside, each with a unique focus. One dialect of Smalltalk, GemStone/S, has been available from GemStone Systems, Inc. since 1985 and provides a distinctive emphasis on server-side scaling and persistence. That is, while other Smalltalk dialects provide a system in which a single virtual machine interacts with an object space that fits in RAM (generally up to a few hundred megabytes on modern hardware), GemStone provides a system in which thousands of virtual machines on multiple machines can interact—using database semantics—with an object space that can be terabytes in size.

For its first couple of decades, GemStone's traditional customers have been large enterprises that do internal development of mission-critical applications using Smalltalk. Typically, they use Cincom's VisualWorks (formerly from ParcPlace) or Instantiation's VA Smalltalk (formerly VisualAge Smalltalk from IBM) to build applications that run on employee workstations (the client) and connect to a GemStone/S database (the server). By using Smalltalk on the client they are able to rapidly develop complex applications using pure objects and the powerful libraries provided by their dialect's vendors. By using Smalltalk on the server, they are able to eliminate the complexity and impedance mismatch of object-relational mapping. In addition, selected back-end processing can be transferred from the client to the server reducing network overhead and allowing more powerful servers to be used.

As the Internet became increasingly popular, libraries and frameworks supporting web application were added to various Smalltalk dialects (including VisualWorks and VisualAge). None of these, however, achieved the traction that Seaside realized and maintainers of other dialects have generally abandoned their proprietary approaches and are focused on supporting Seaside. Seaside originated in Squeak (http://www.squeak.org/) and is now being maintained primarily in Pharo (http://www.pharo-project.org/). At this point there are ports of Seaside to Cincom Smalltalk from Cincom (http://www.cincomsmalltalk.com/), VA Smalltalk from Instantiations (http://instantiations.com/VAST/index.html), Dolphin Smalltalk from Object Arts (http://www.object-arts.com/), GNU Smalltalk (http://smalltalk.gnu.org/), and, of course, GemStone Smalltalk from VMware (http://seaside.gemstone.com/).

While each dialect brings unique strengths (and weaknesses) to Seaside, we will be focusing on GemStone/S in this (and subsequent) chapters. Because of GemStone's background as a server Smalltalk, it is uniquely positioned to provide scalability (in both objects and number of users) to a Seaside application. The trade-off is that a GemStone installation tends to be larger (thus more complex to use and manage) does not have a native suite of GUI tools (so relies on other Smalltalk dialects for development tools such as browsers), and the Seaside-enabled version only runs on 64-bit *nix operating systems (including AIX, HPUX, Linux, Macintosh, and Solaris, but not Microsoft's Windows).

GemStone has a reputation in the Smalltalk community of being very desirable (who would want to use SQL when they could use Smalltalk?) but also very expensive. Fortunately, the cost issue has been addressed in part with a no-cost ('free as in beer') license of the product for Linux with the primary limitation of 2 CPUs and a 2 gigabyte shared page cache (see http://seaside.gemstone.com/docs/GLASS-Announcement.htm for details). A version is also available for Macintosh.

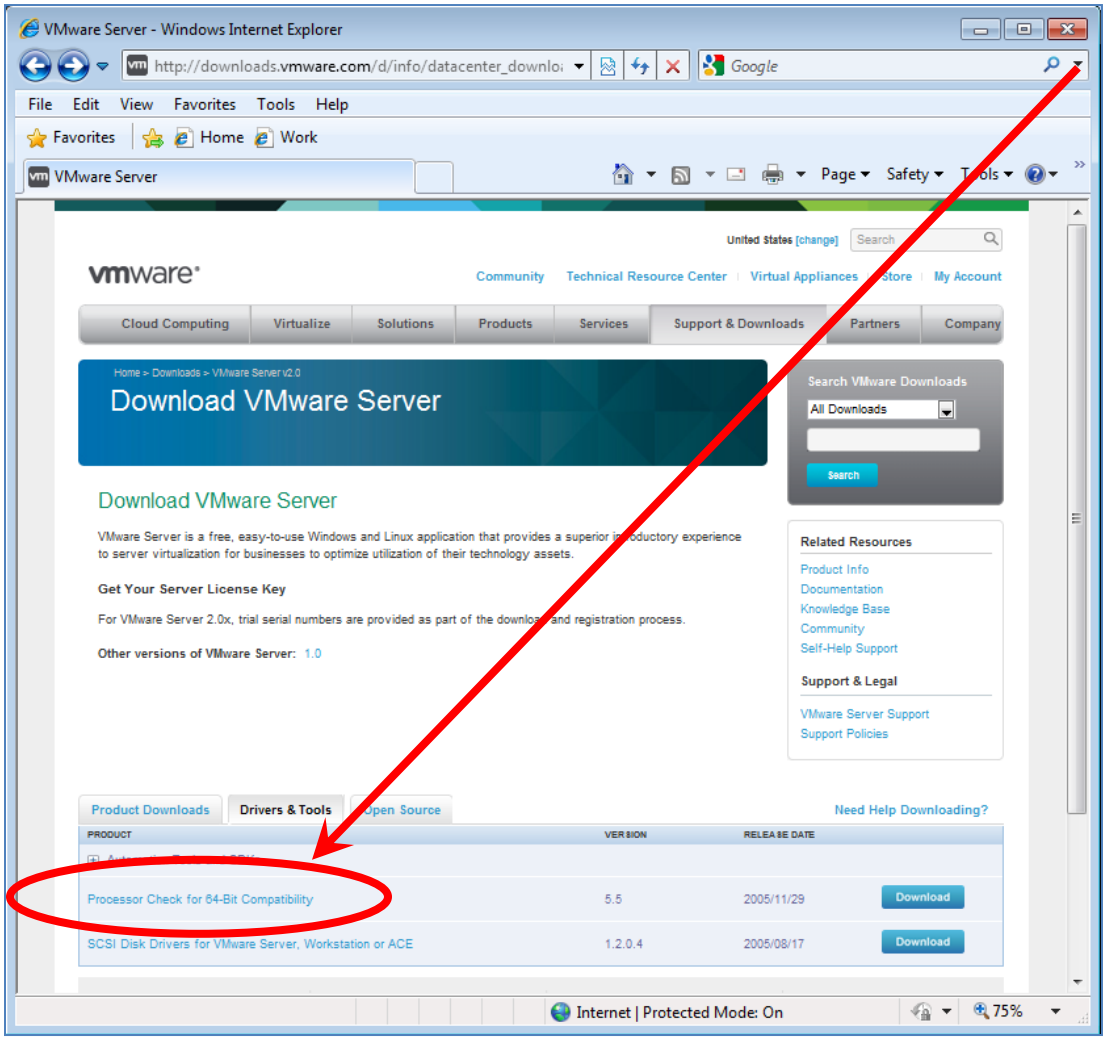
While getting started with GemStone is not quite as easy as using Pharo's 'Seaside One-Click Experience,' it is not nearly as difficult as, say, an Oracle install. There are two options for installing GemStone/S 64 Bit: (1) native, and (2) VMware. In this chapter we focus on the VMware approach.

VMware Virtual Appliance. This is the approach most likely to get everything working quickly, but requires 64-bit hardware with virtualization support and an installed copy of VMware Server (free for Linux and Windows) or VMware Fusion (for Macintosh) along with a license key. The Virtual Appliance has a full install of 64-bit Linux, Apache, GemStone/S 64 Bit, and other components (such as FastCGI).

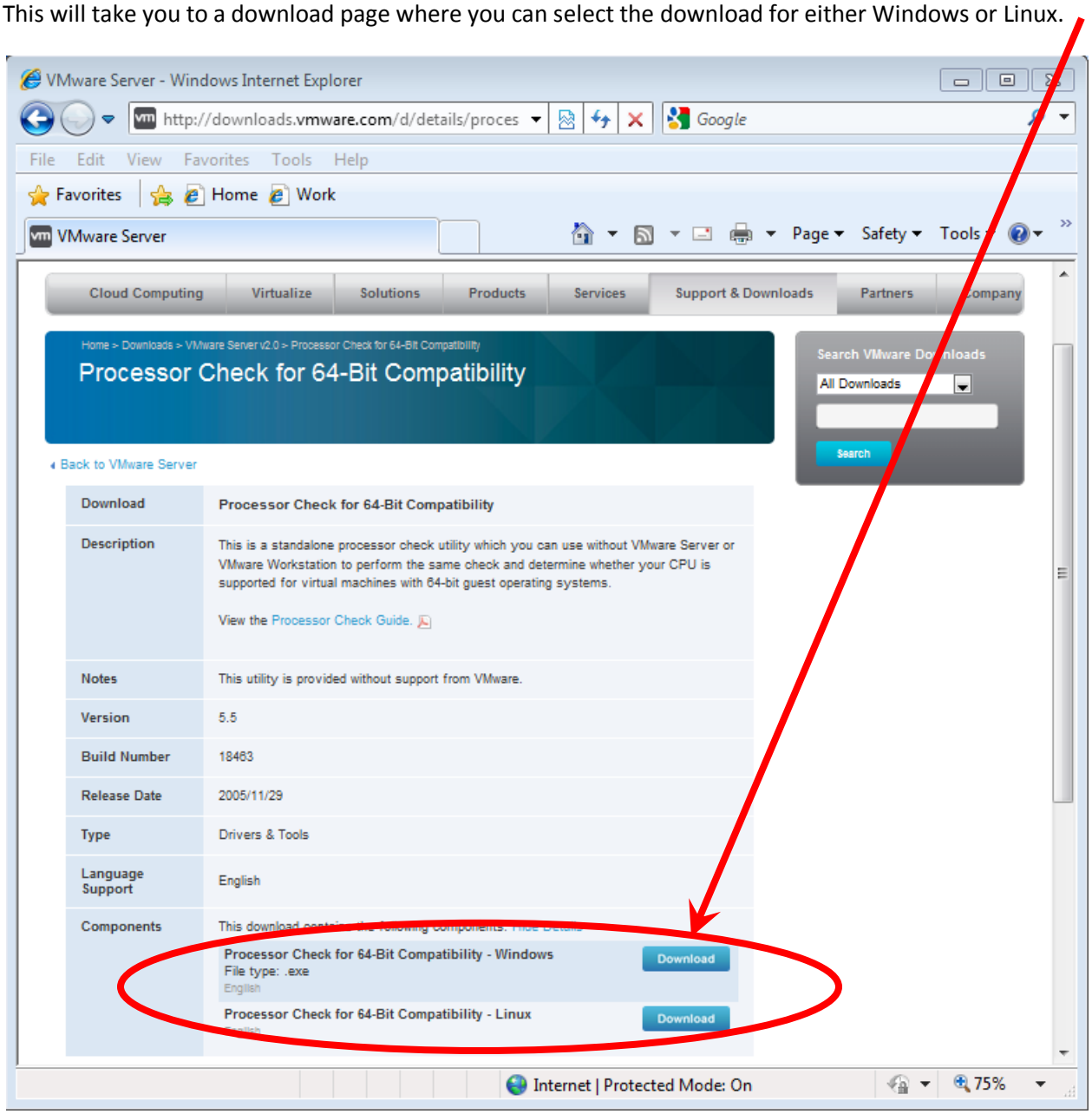
The first step is to see if your hardware supports 64-bit virtualization. If you are running a recent MacBook, then you are fine. If you are running Windows or Linux, then you need to check your machine. Open a web browser on the following link:

```
http://www.vmware.com/download/server/drivers_tools.html
```

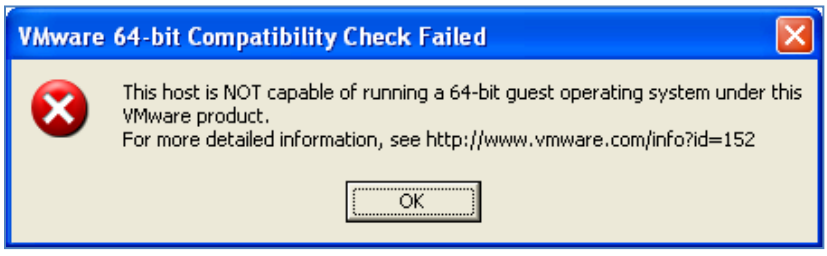
Click the link 'Processor Check for 64-Bit Compatibility' in the Drivers & Tools tab.



This will take you to a download page where you can select the download for either Windows or Linux.

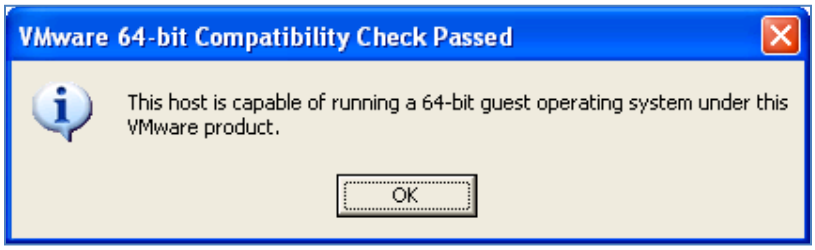


After downloading the guest check application run it to see if your hardware supports virtualization. If the answer is 'no,' then you need to pursue another approach (run GemStone/S native if you are on a 64-bit Linux OS or get access to another computer).

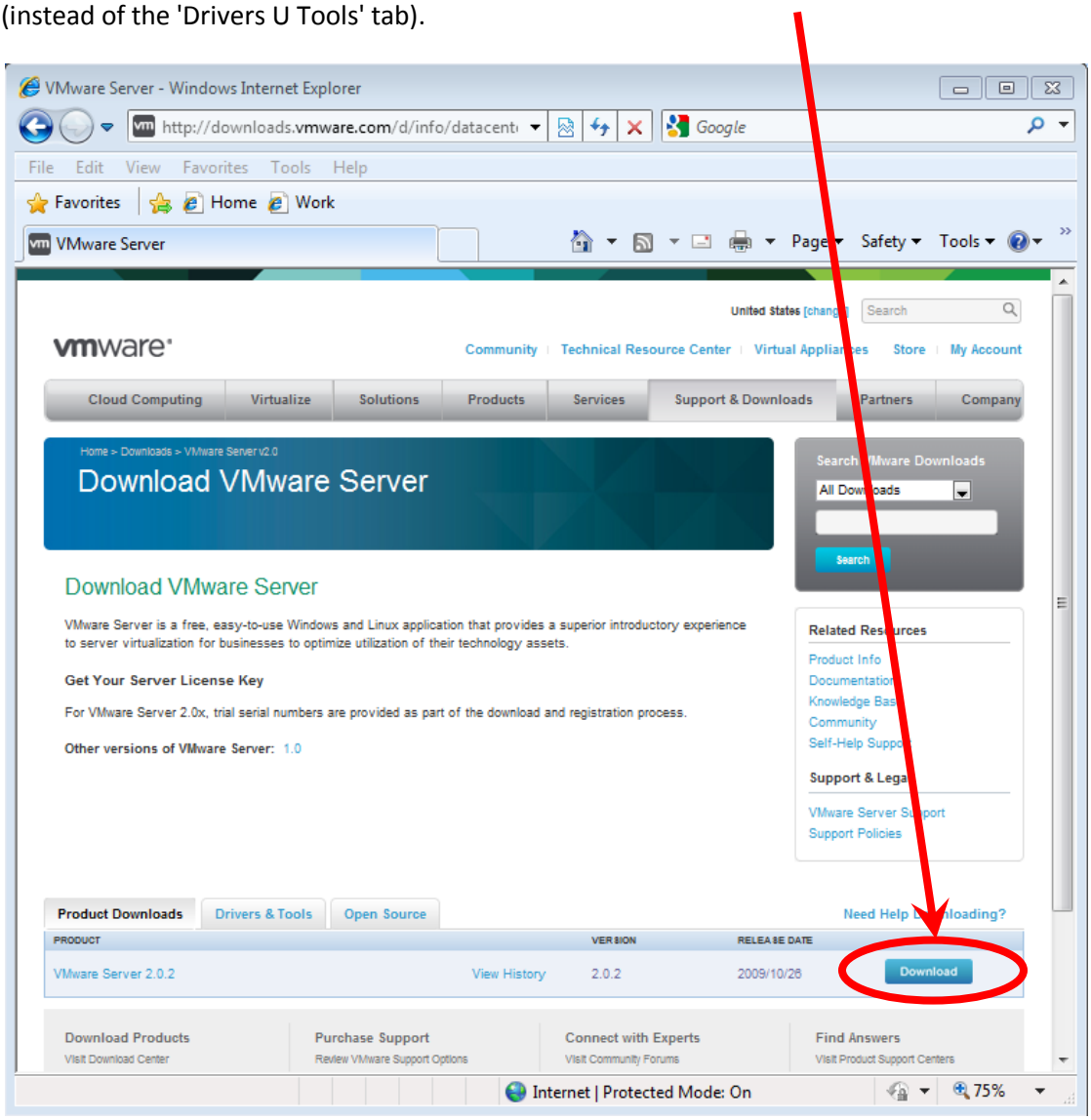


If your processor supports 64-bit virtualization, then you need to see if it is enabled in your computer's BIOS. Reboot your PC and press <F2> during the boot process to get into the BIOS setup. Look for a setting named something like 'Enable VT,' see that it is true/yes, save the settings, and then reboot the machine. Unfortunately, we are unable to give more detailed instructions because the BIOS settings differ depending on your machine. See http://kb.vmware.com/selfservice/viewContent.do?externalId=1003944&sliceId=1 for some instructions from VMware.

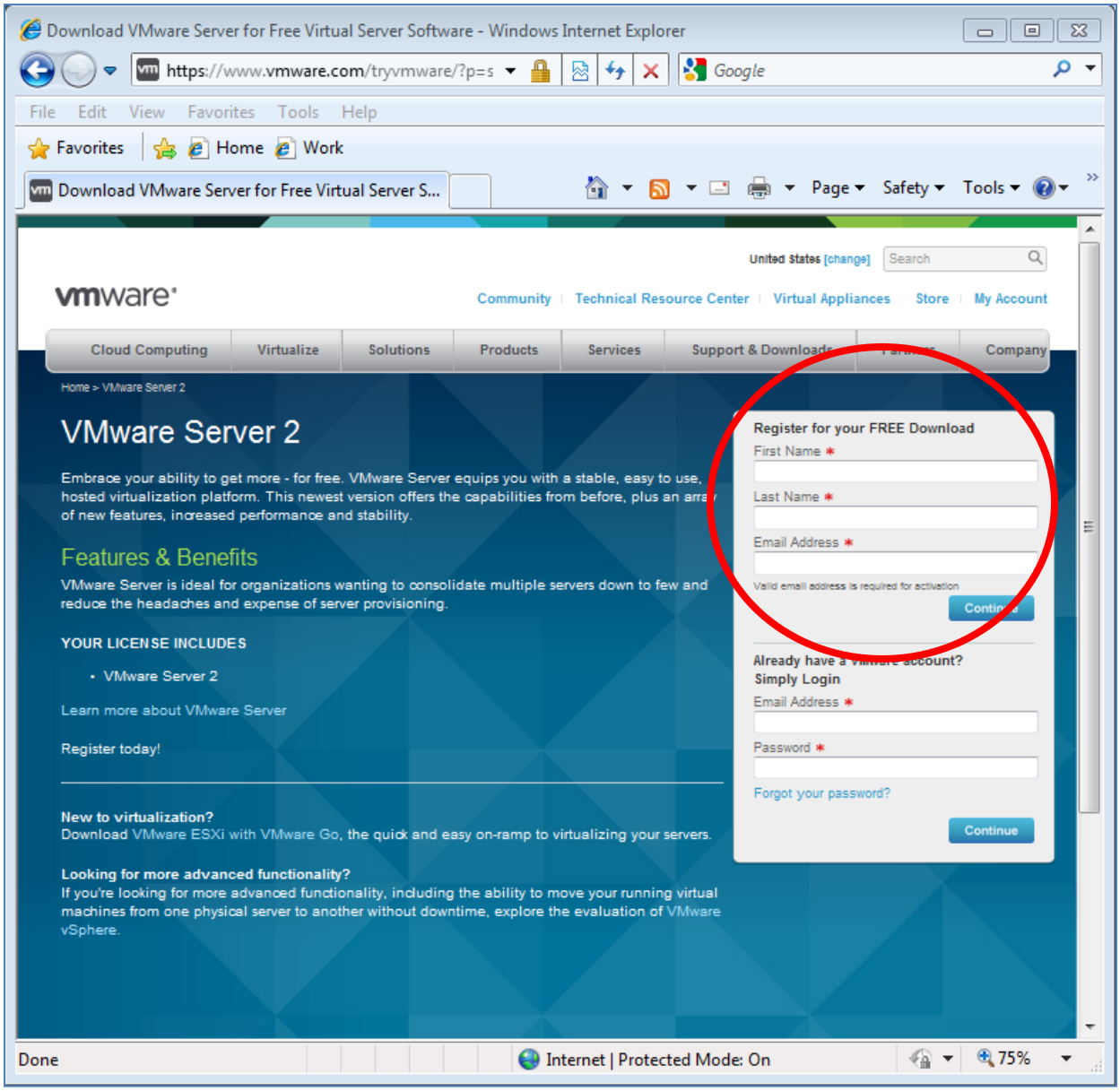
If things work properly, then you should see the following dialog.



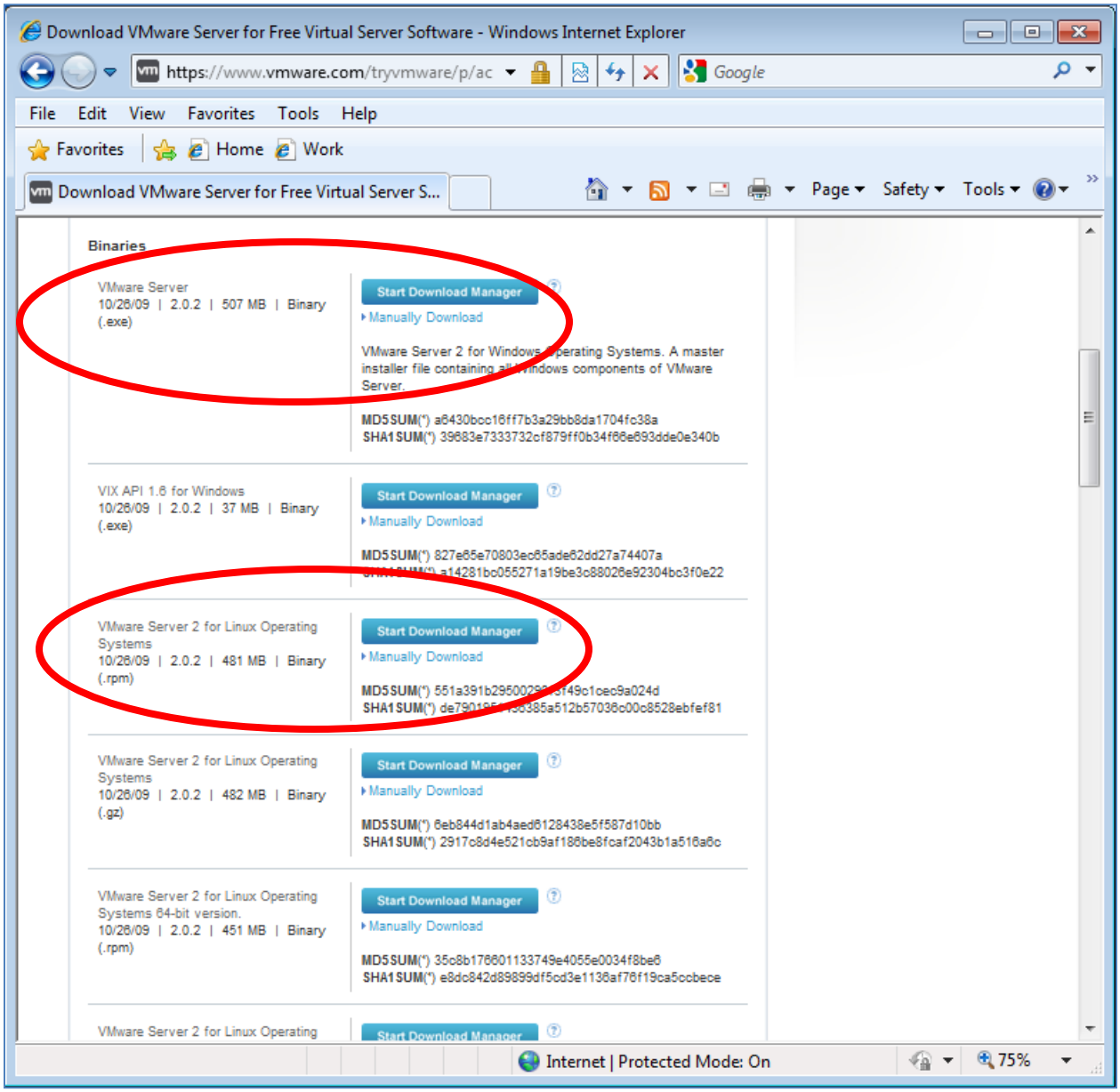
Next, download VMware Server 2.0 from the previous web site by selecting the 'Product Downloads' tab (instead of the 'Drivers U Tools' tab).



During this download process you will be required to register and a registration key will be sent to an email address you provide.

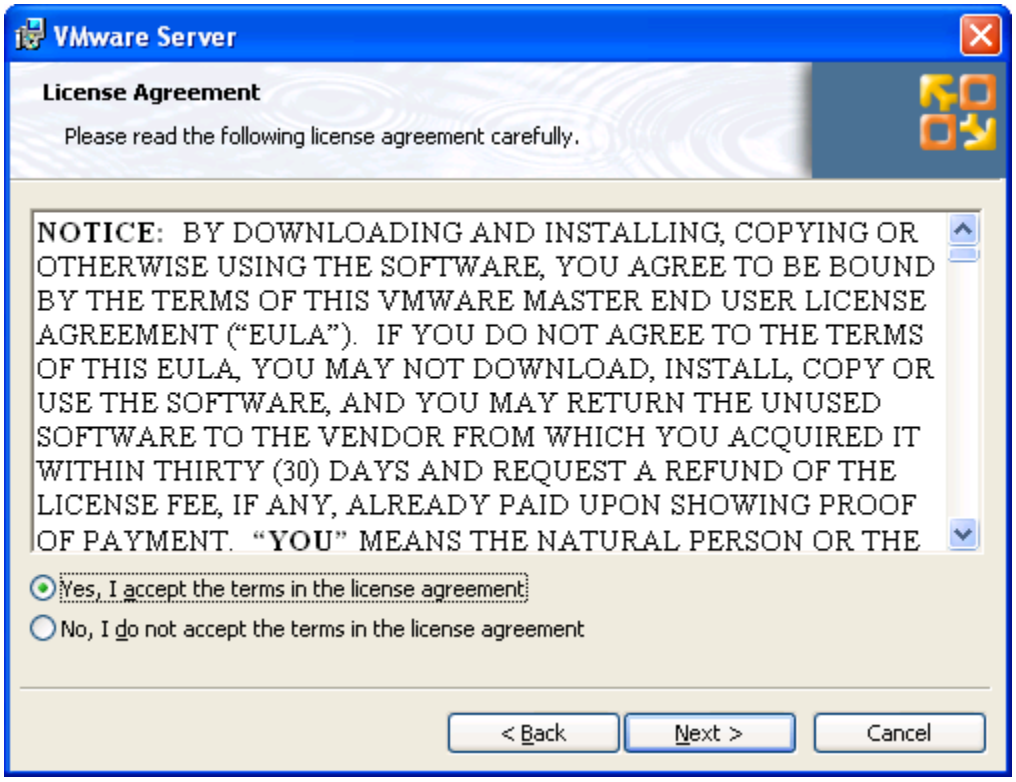


Once you complete the registration process (including a license page) you will be presented with a download page. Download the version of VMware Server 2 appropriate for your operating system. We will demonstrate the process of installing on 32-bit Windows XP.

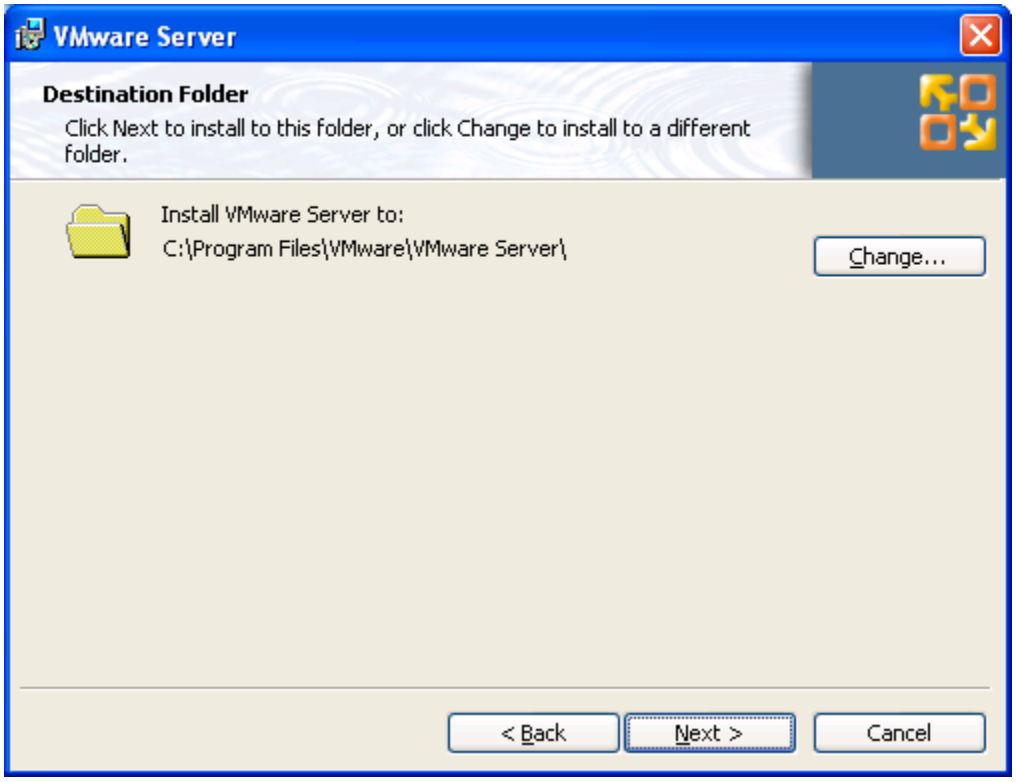


Once you download the installer, launch it.

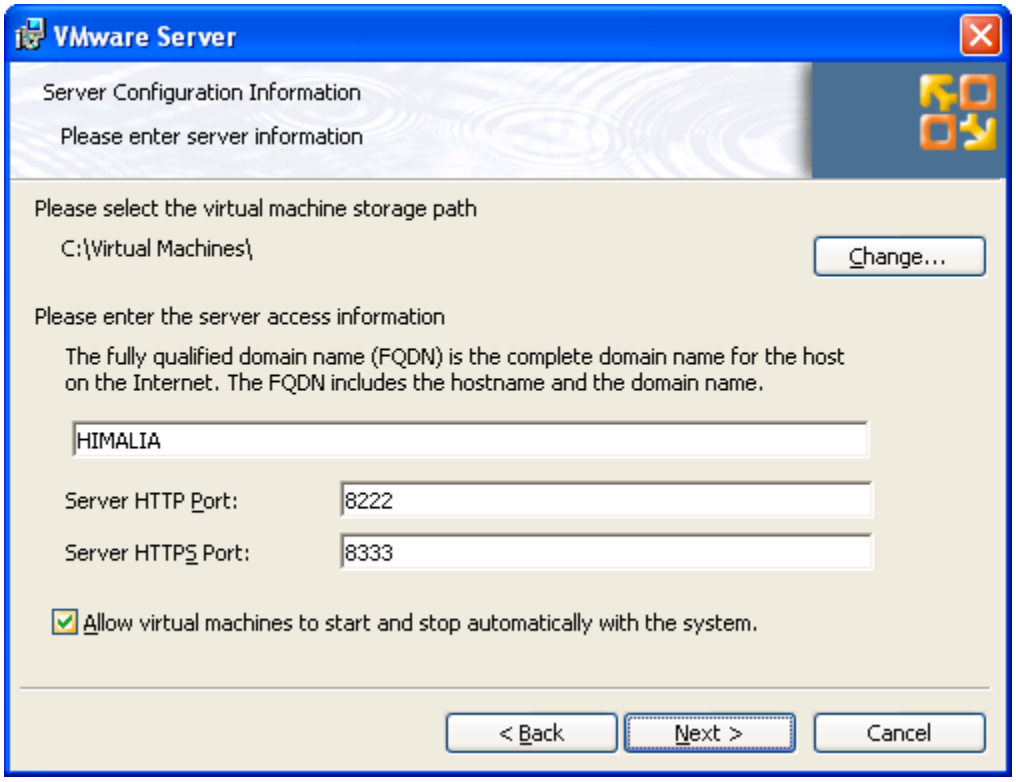
After reading the license (you do that don't you?), click 'yes' (if you agree) and then 'Next.'



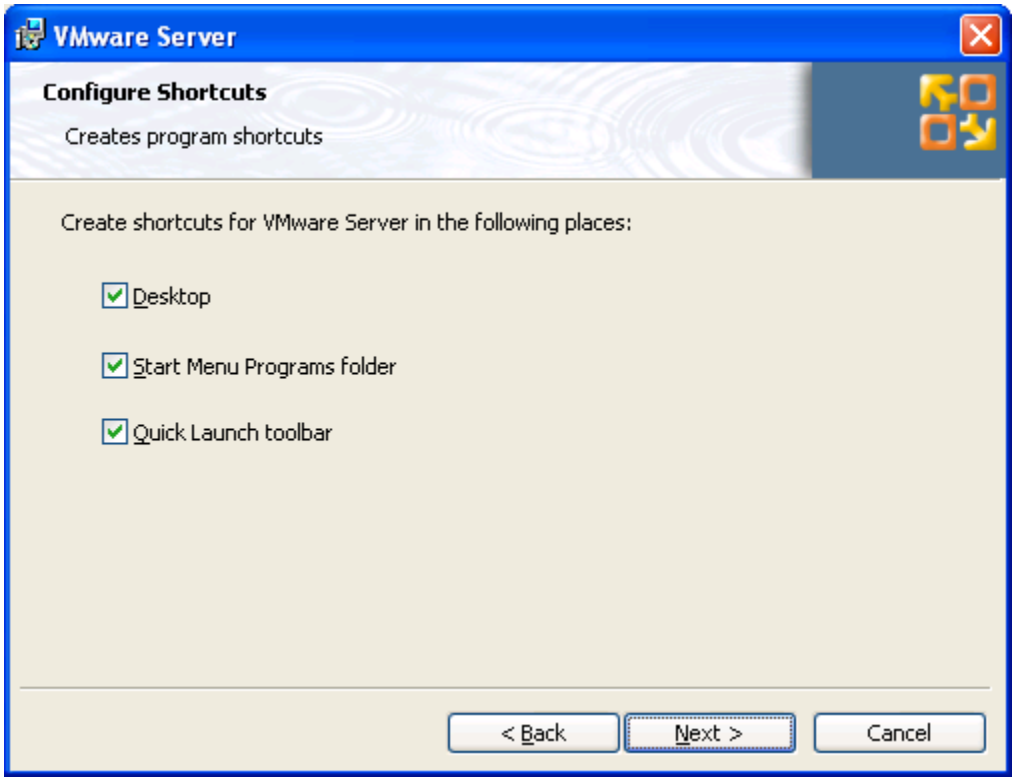
Select a location in which to install the software. By default it will go in Program Files.



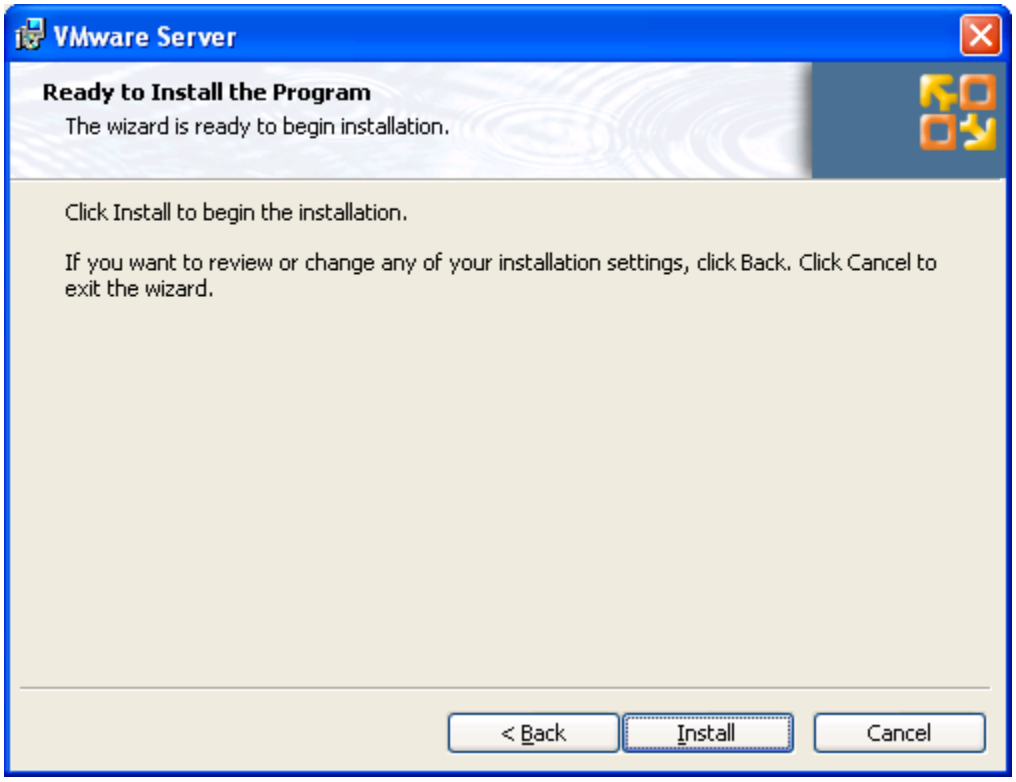
Unless you have a good reason, the server configuration information should be left as is.



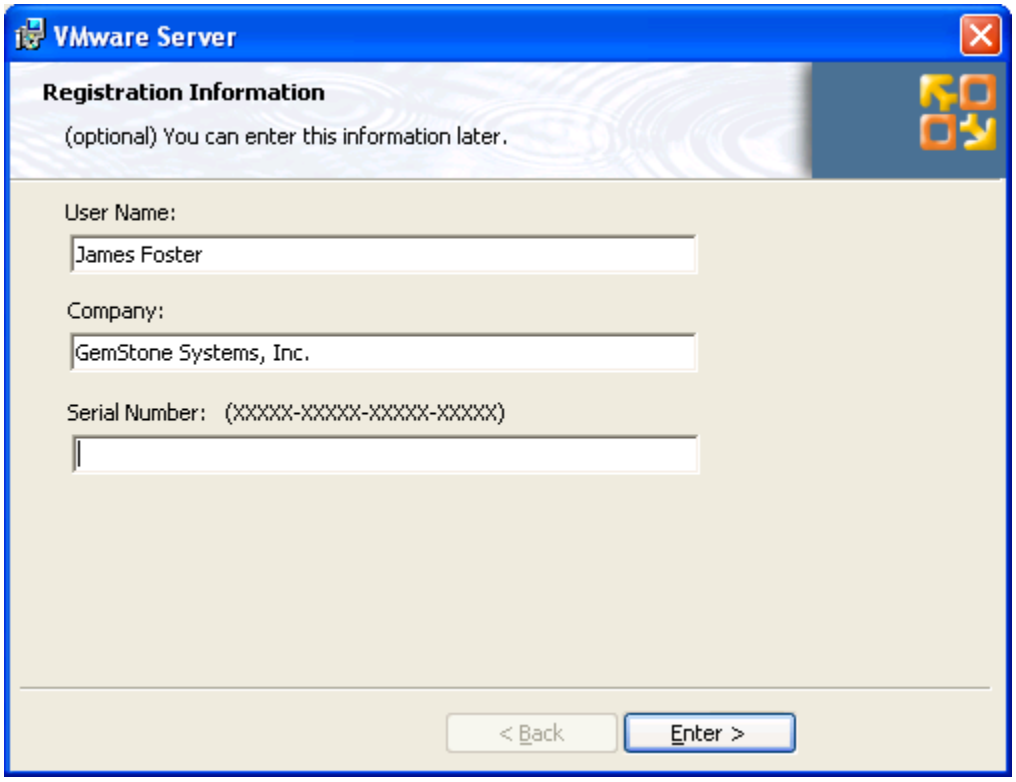
Decide whether to allow shortcuts to be included in various places.



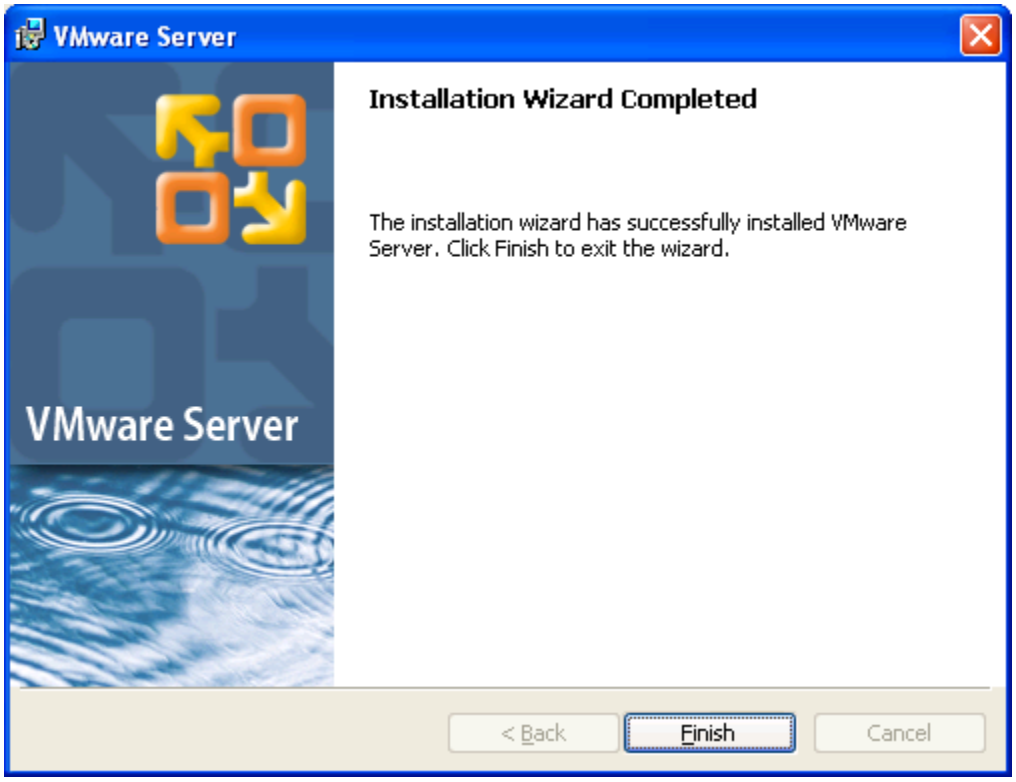
When the installer has all the setup information it needs, it can start.



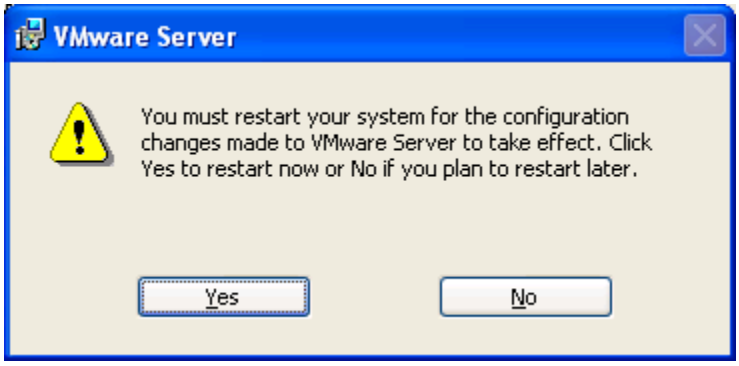
When you registered for the download you gave an email address. Check that mailbox for a key.



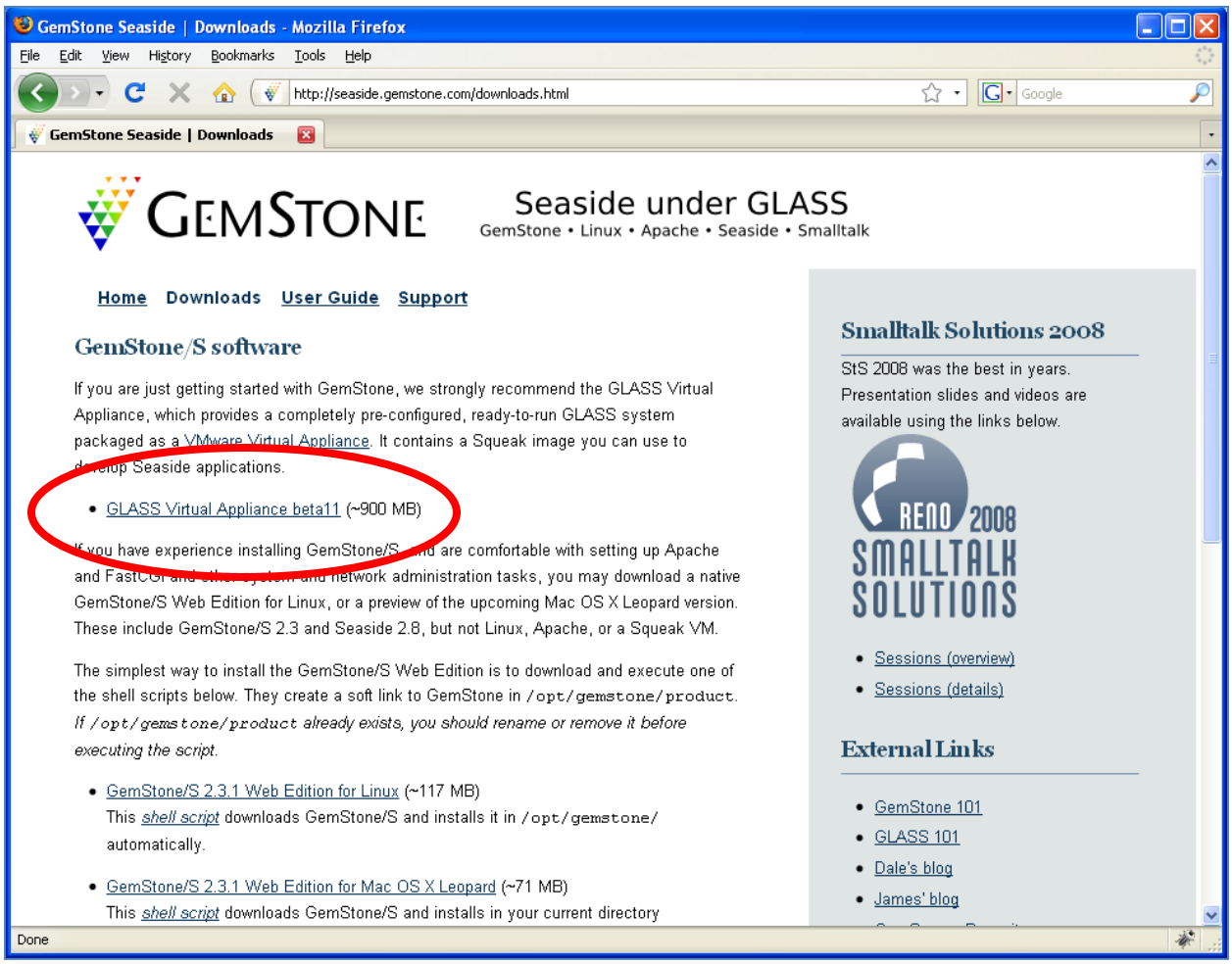
When the installation is done click 'Finish.'



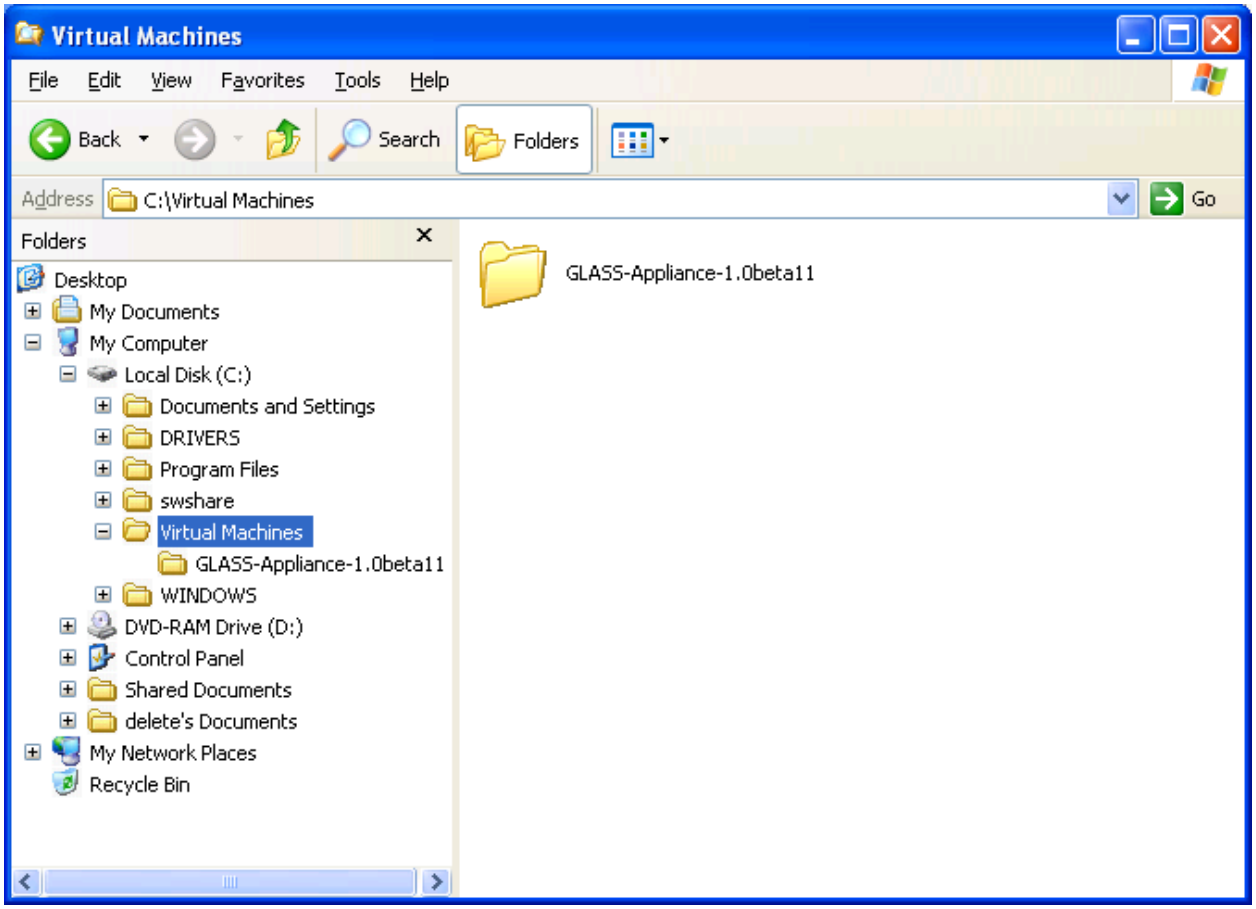
Like many other applications, an installation requires a restart.



Using a web browser, navigate to GemStone's download page (http://seaside.gemstone.com/downloads.html) and download the GLASS Virtual Appliance.

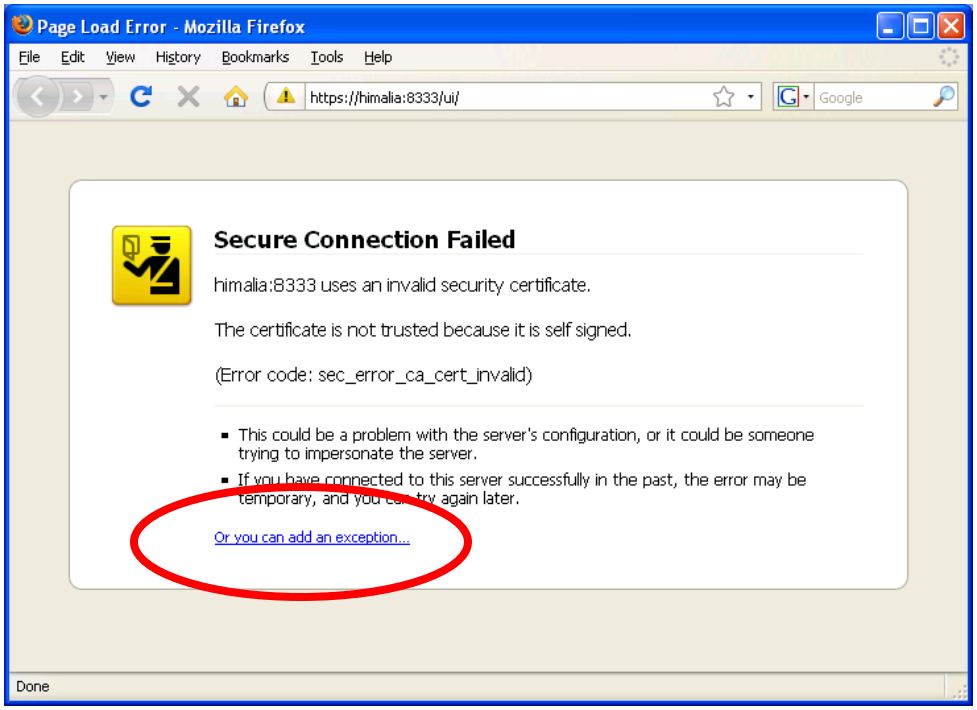


After the file downloads, unzip it and place it in 'C:\Virtual Machines' for VMware to find.

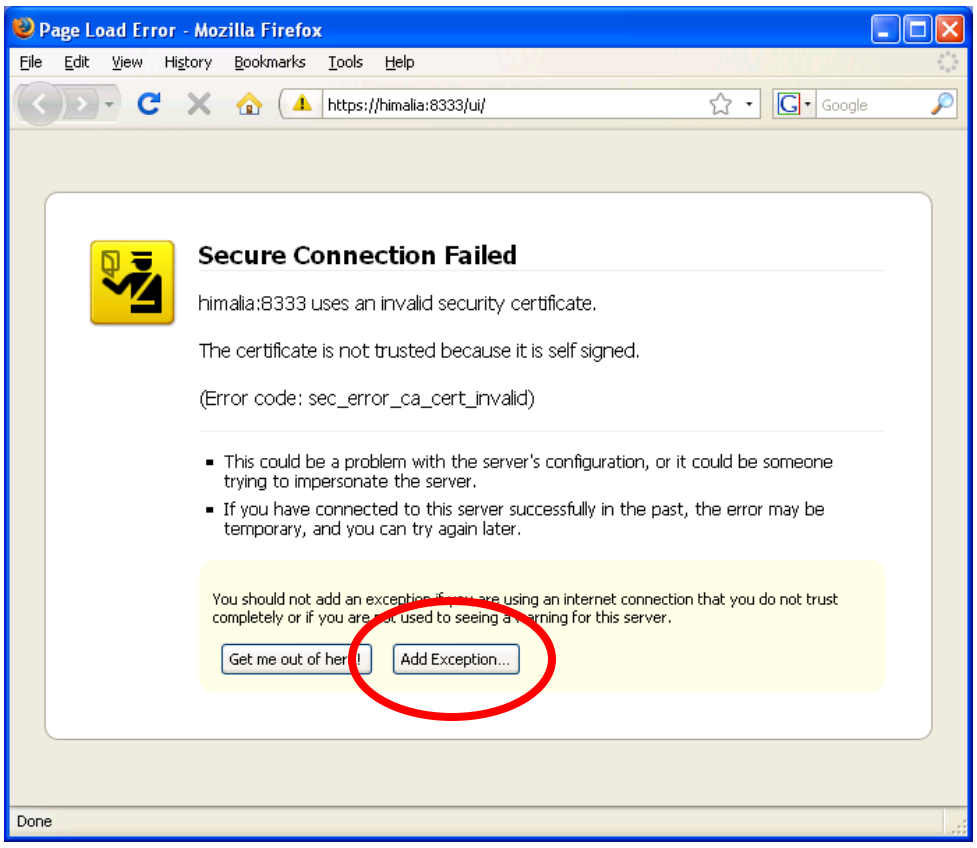


Now launch VMware Server using one of the shortcuts provided by the installer. This will open a web browser and, depending on your security settings, will report a problem.

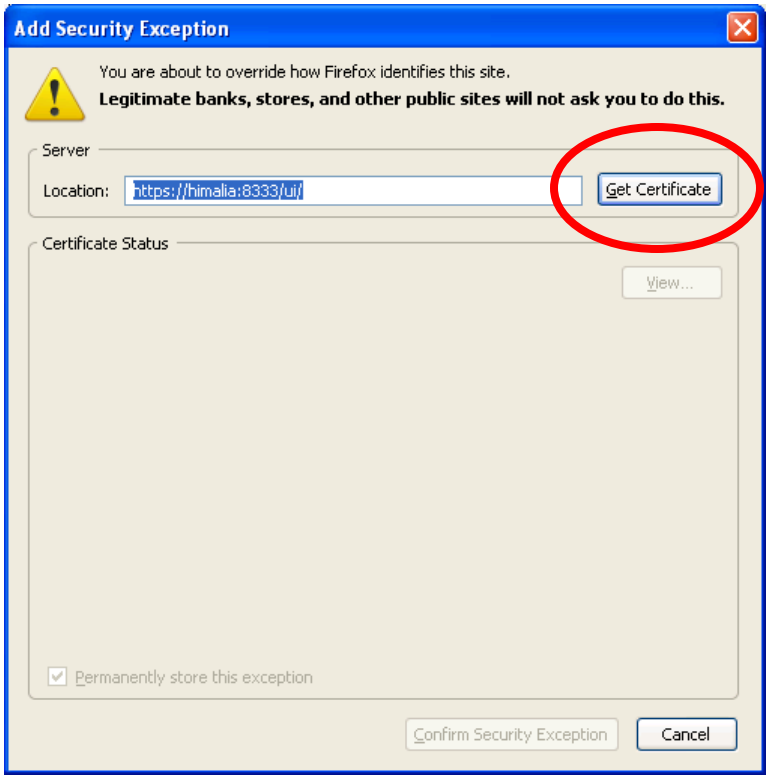
Click OK and the browser will show an error. Click on the 'Or you can add an exception…' link.



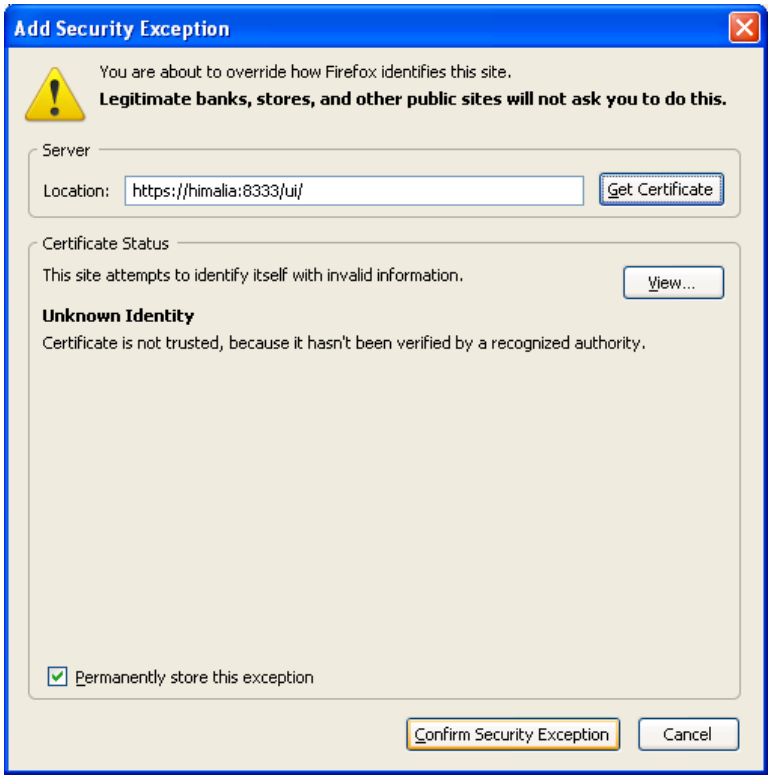
After you click the link you will be shown two buttons. Click the 'Add Exception…' button.



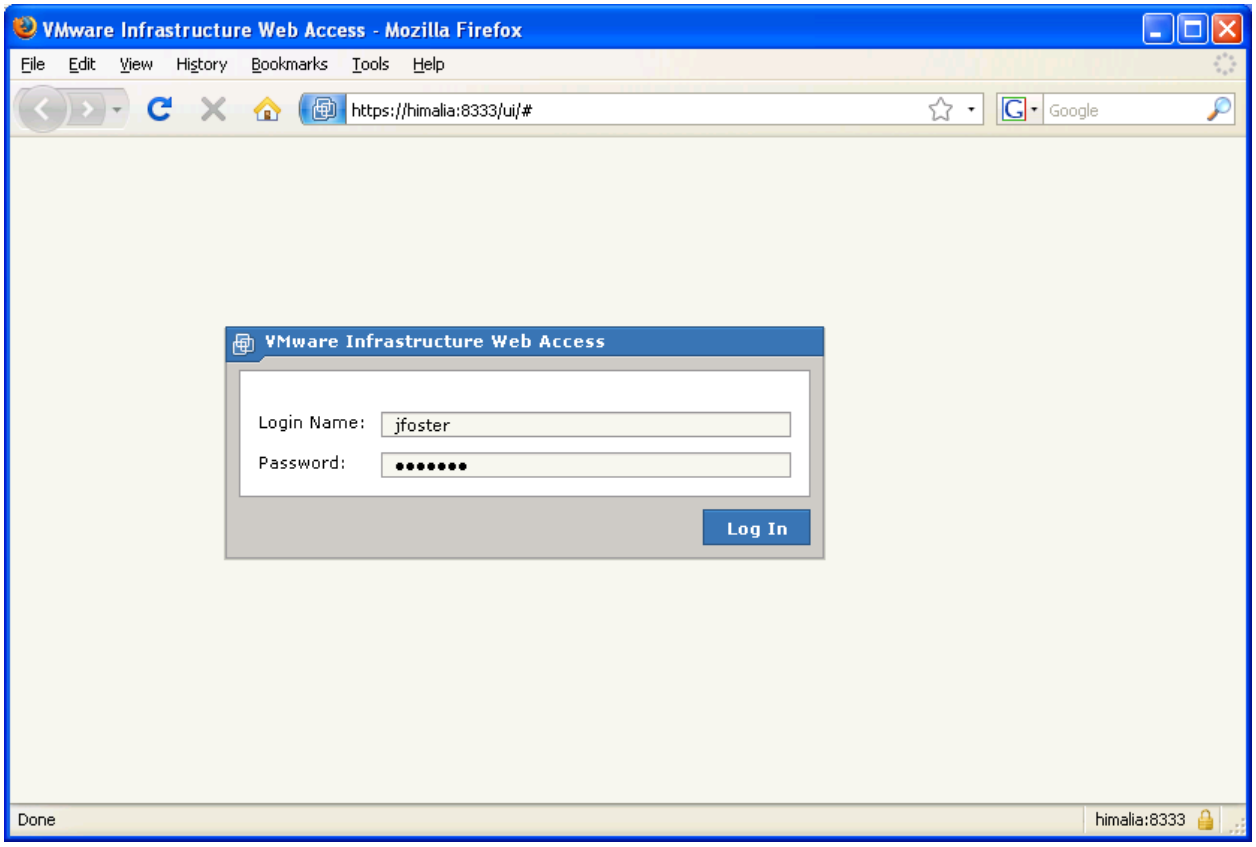
Firefox requires you to get the certificate before you can confirm the exception.



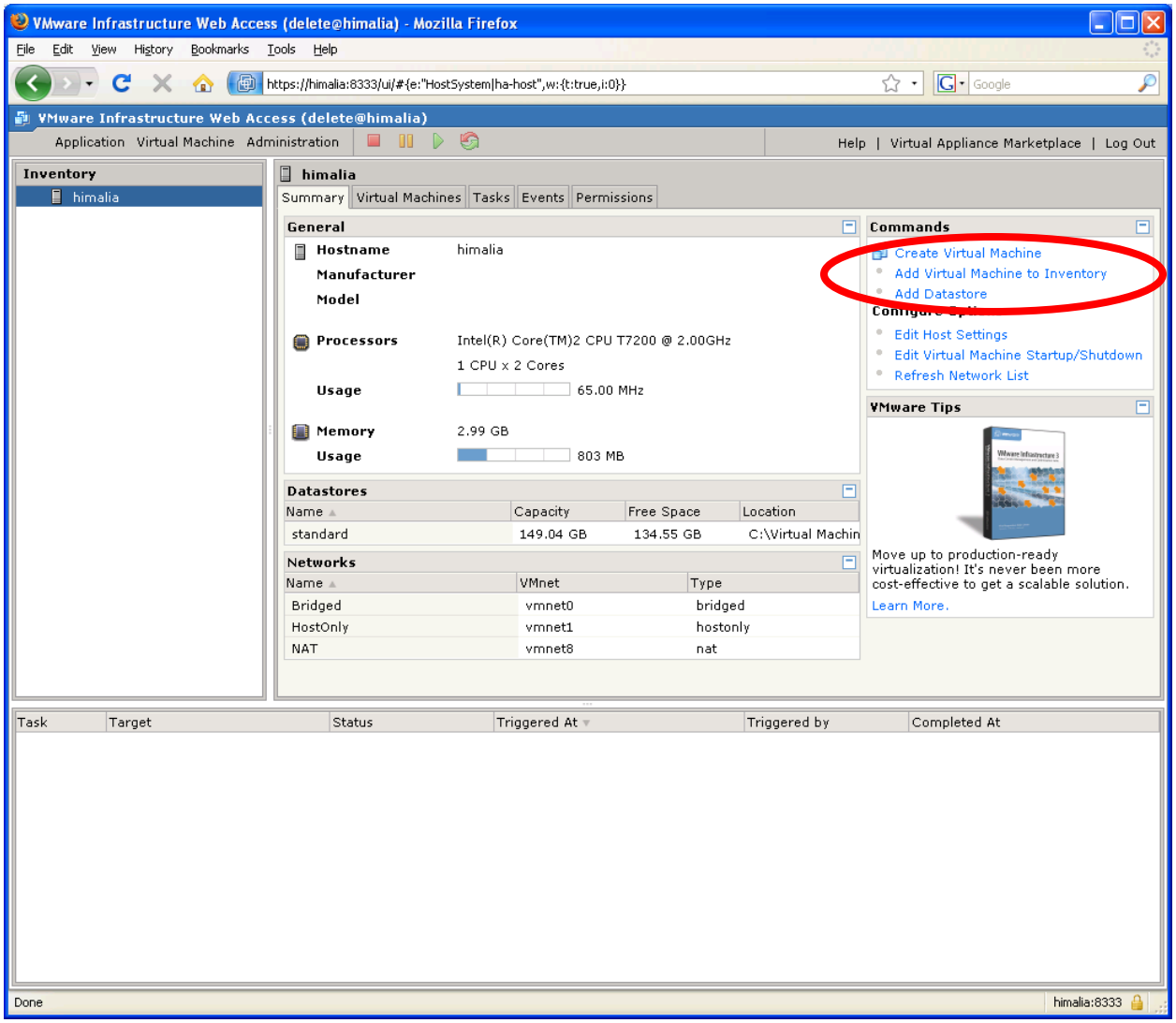
Once you get the certificate, click the 'Confirm Security Exception' button.



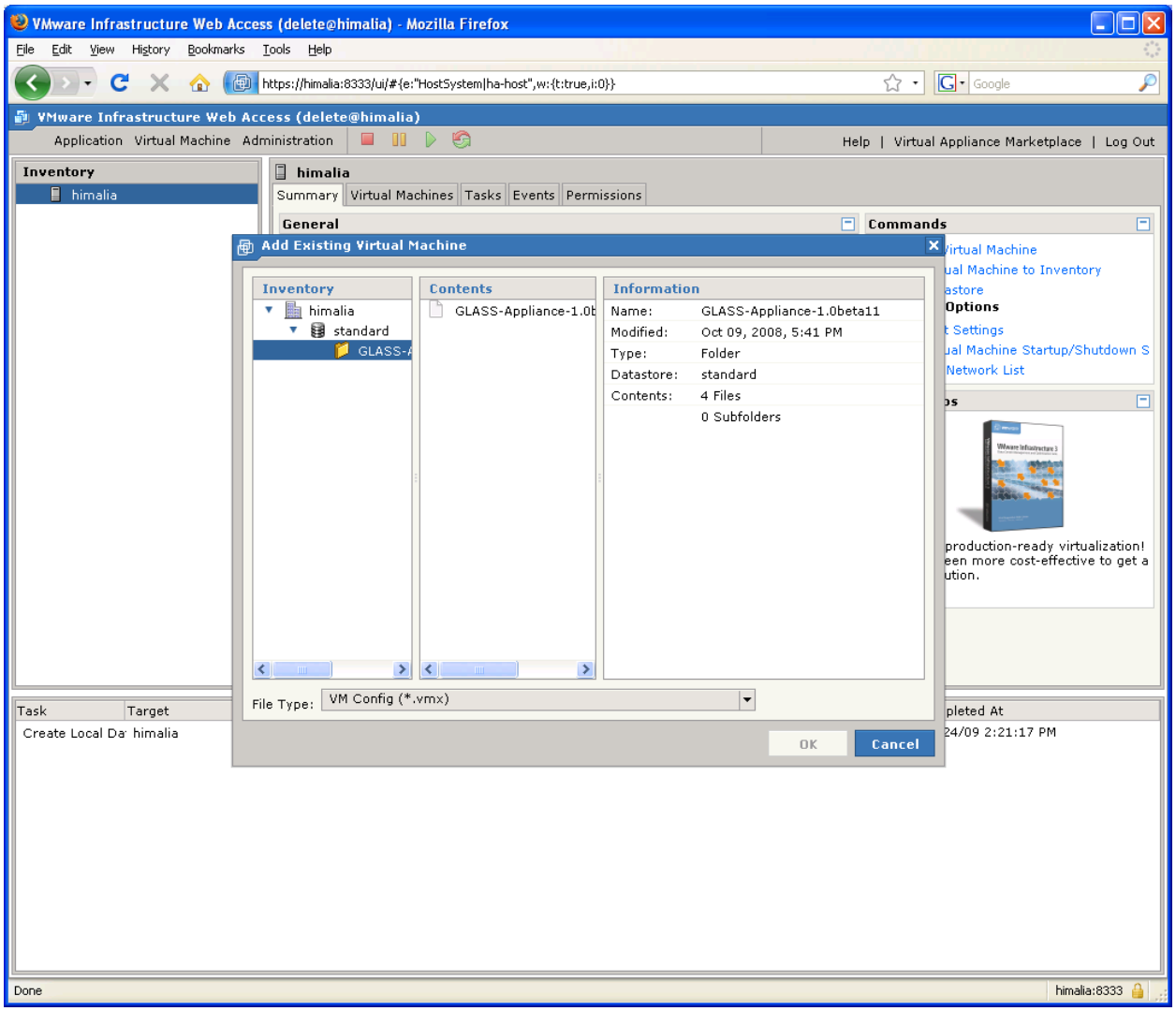
Once you confirm the security exception, you should get a web browser with a login screen asking for you host operating system user and password. Note that VMware requires a password. If your Windows account does not have a password you will need to add one and then give VMware your Windows password. Note that you could access this screen from another computer on the network using the same URL.



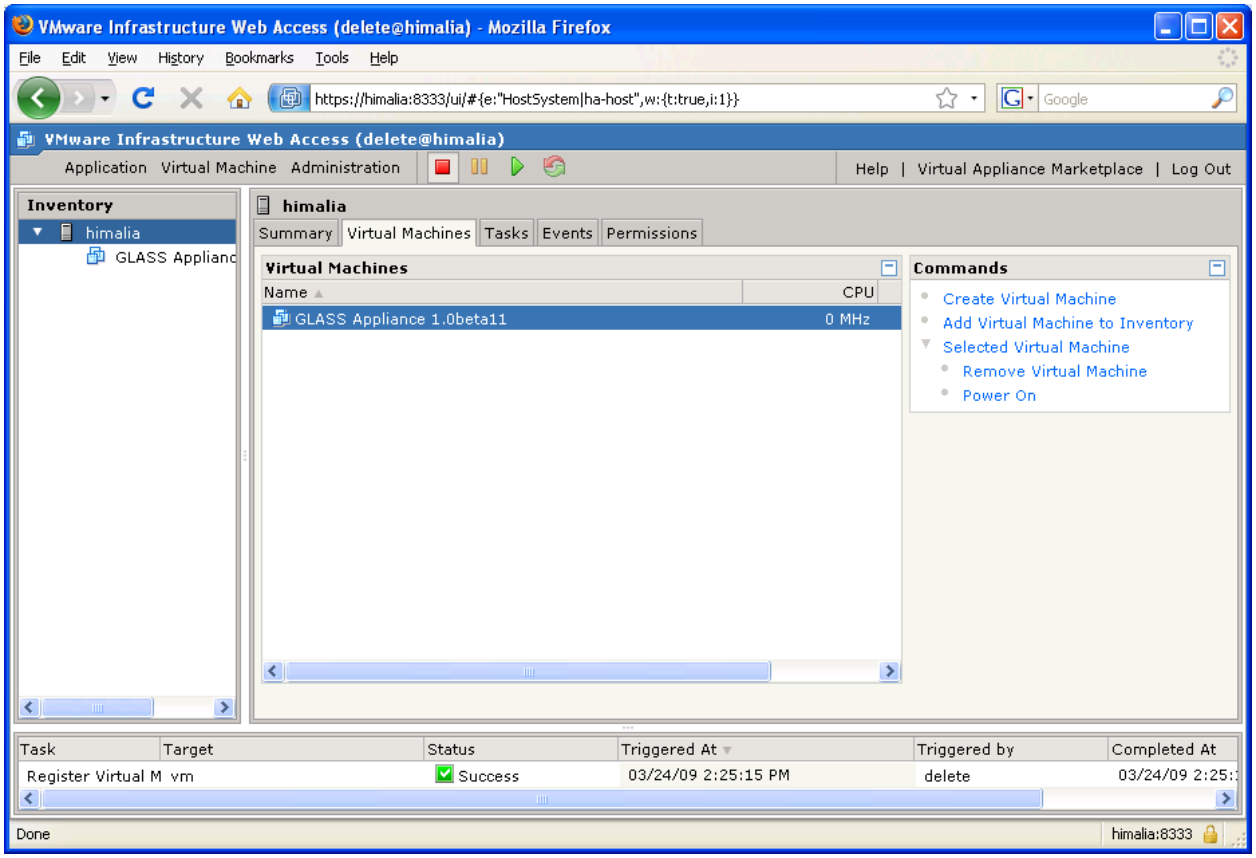
Once you have a successful login, you should see the VMware Infrastructure Web Access. Click on the command link ‘Add Virtual Machine to Inventory.’



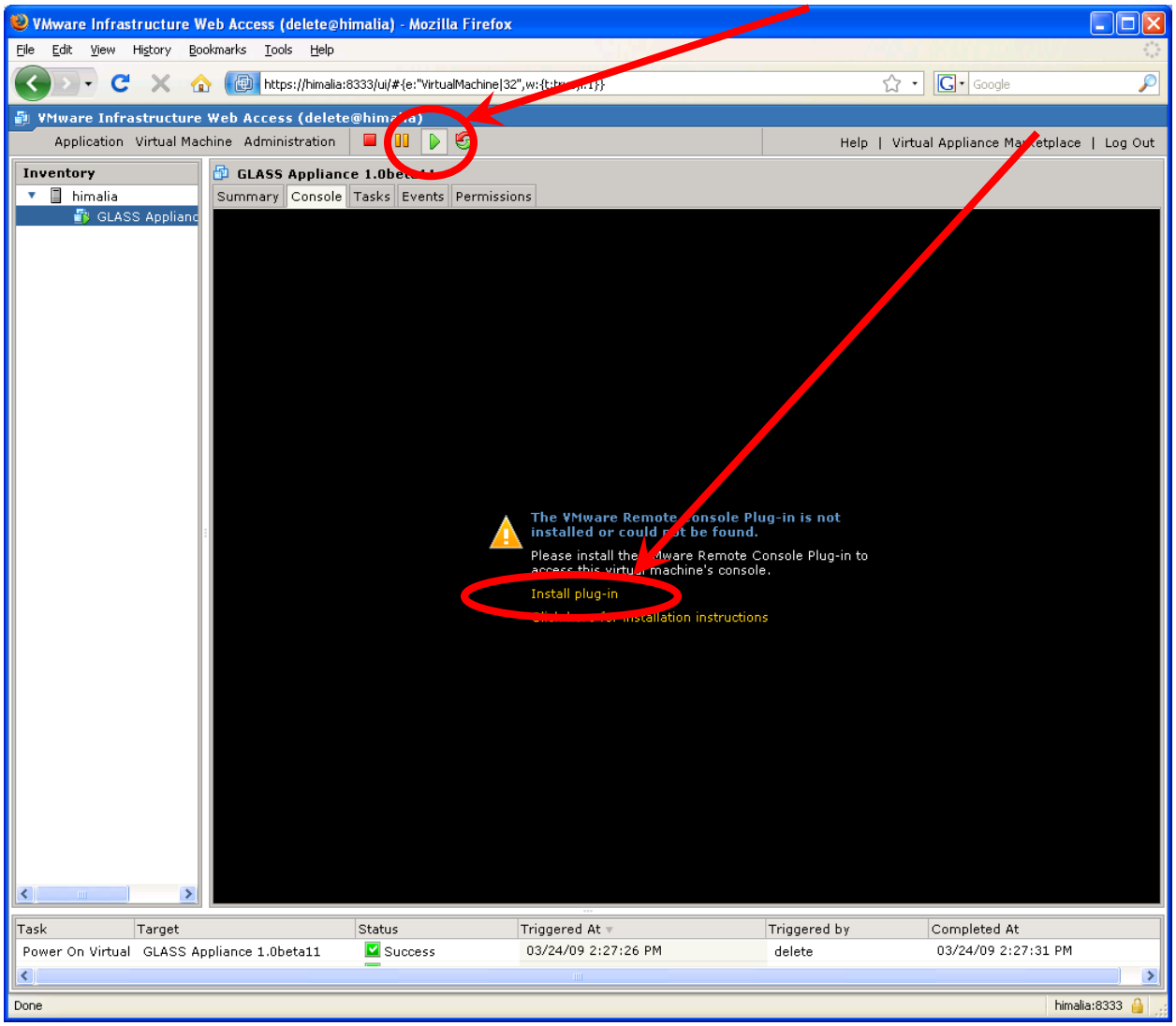
This will present a dialog. In the first column (Inventory), expand the machine, the inventory, and select the GLASS directory. In the second column (Contents) select the GLASS-Appliance. Finally, click the 'OK' button.



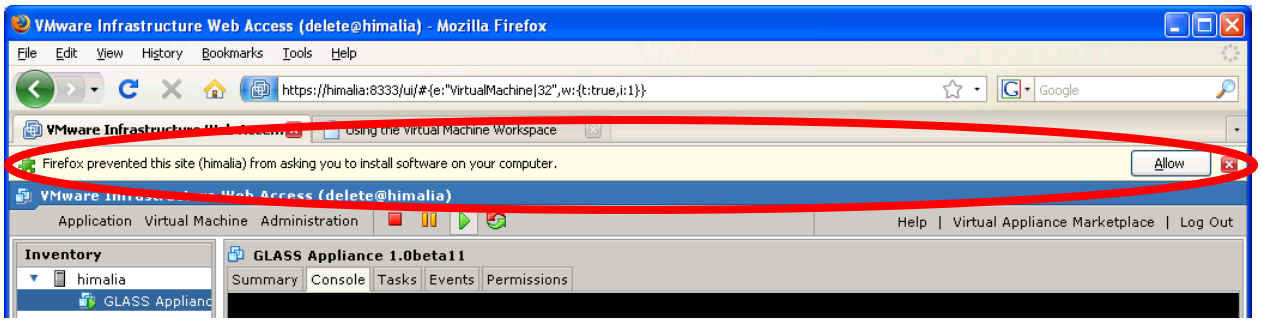
Once you have added the GLASS Appliance it should show up in the list of virtual machines.



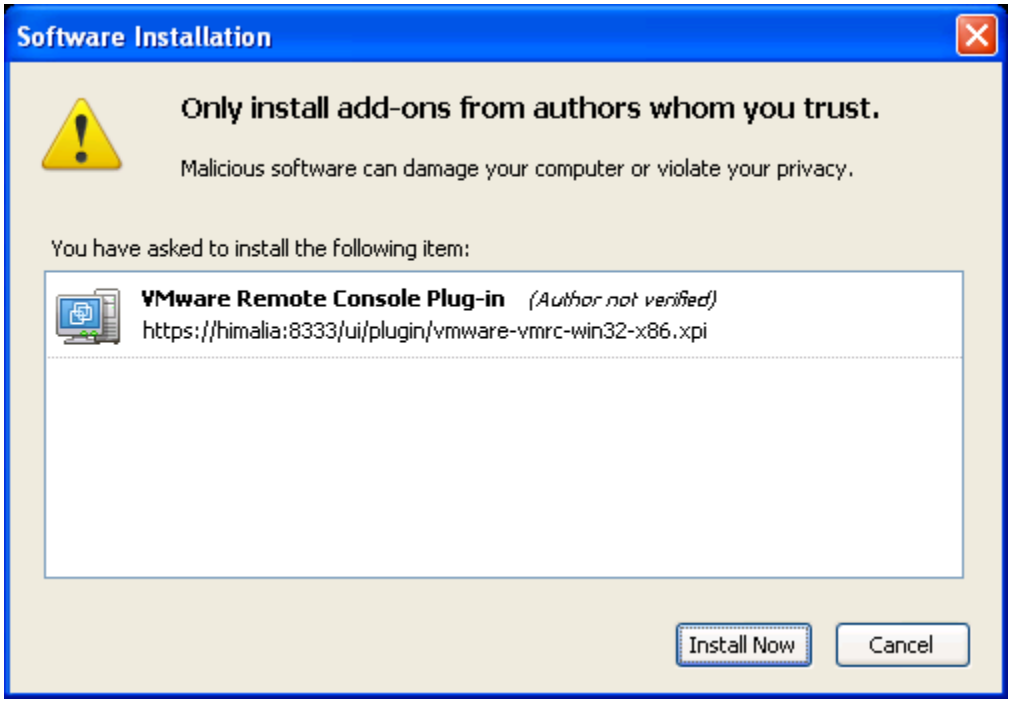
Click on ‘GLASS Appliance’ in the Inventory list (the first column). This will change the area to the right showing a summary and other tabs about the appliance. Click the green start button and then switch to the ‘Console’ tab. This will inform you that the VMware Remote Console Plug-in is not installed. Click on the ‘Install plug-in’ link.



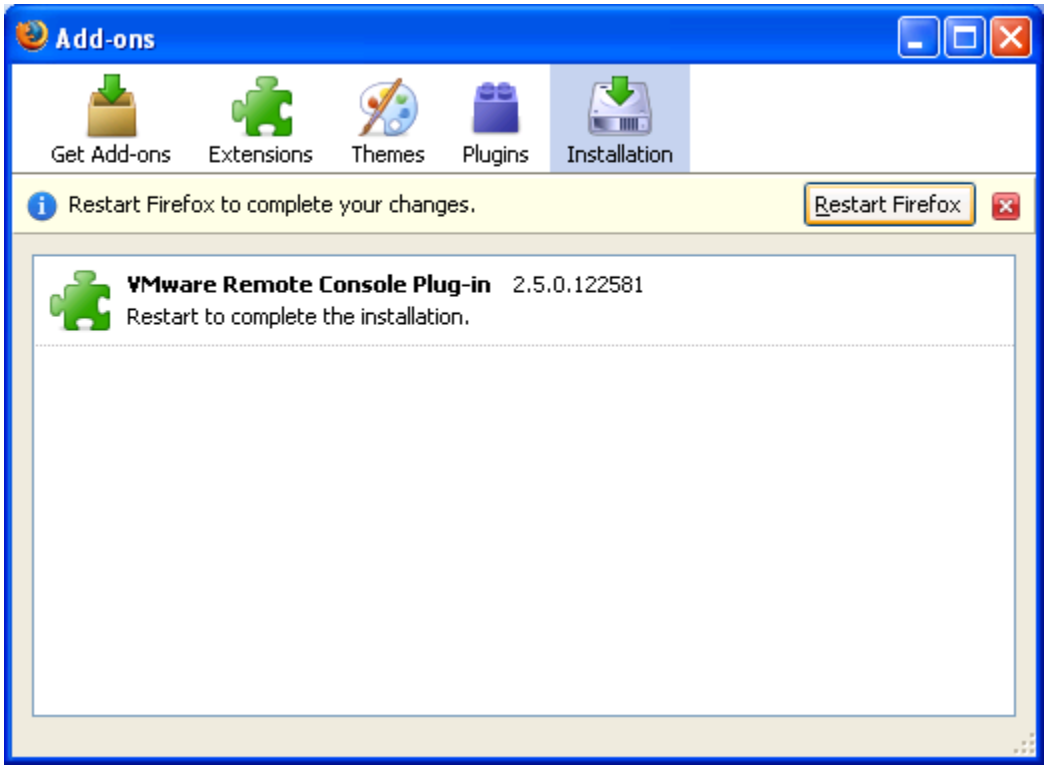
The plug-in installation might prompt you to allow the installation. Click the ‘Allow’ button.



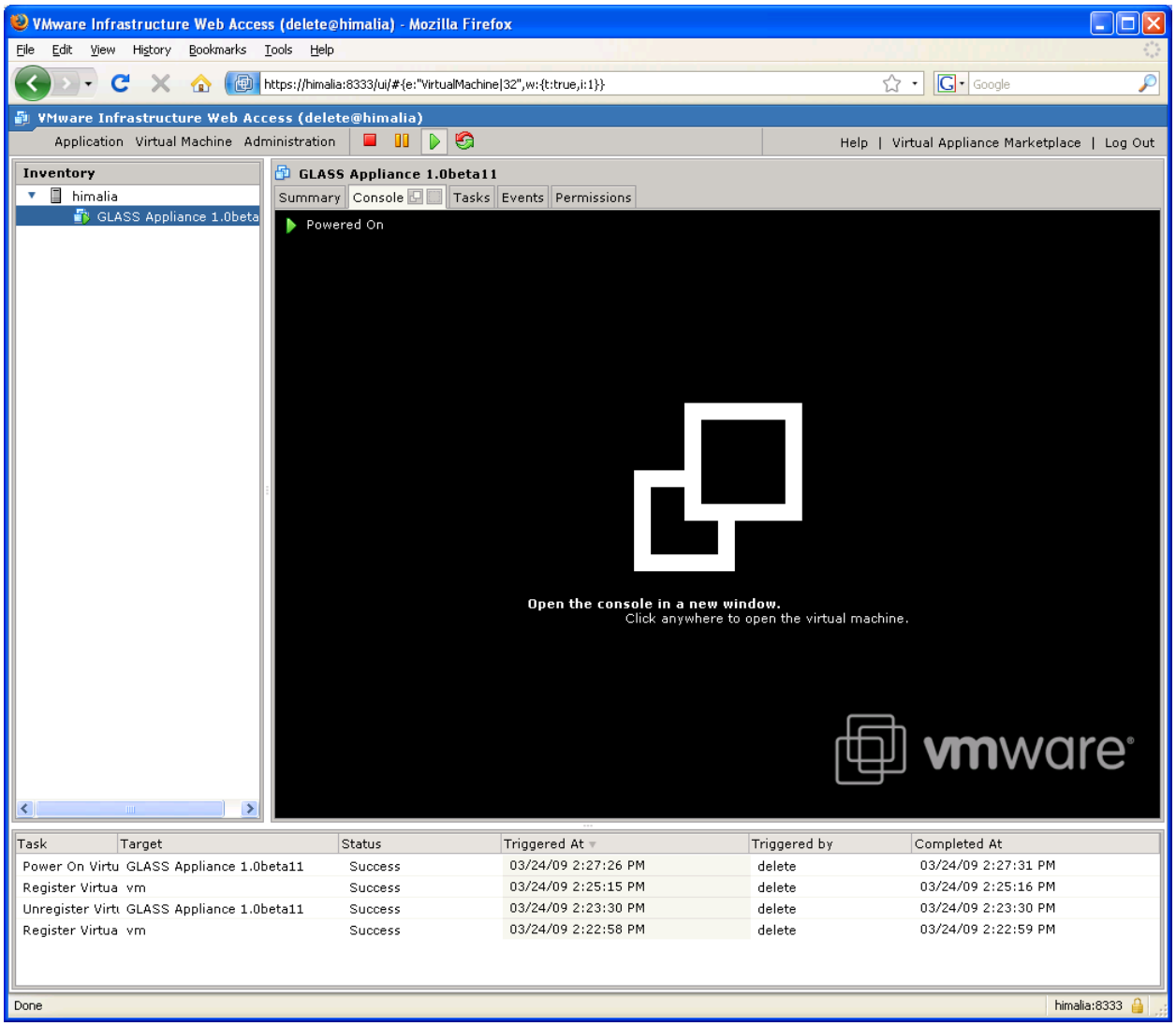
You might be prompted to confirm the installation.



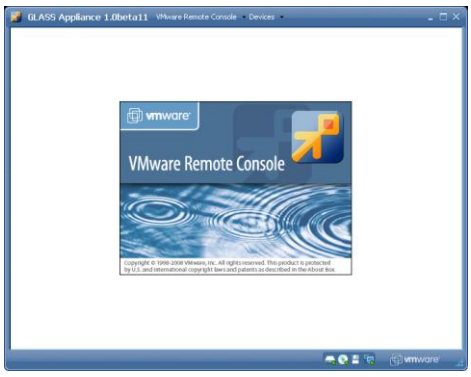
You might need to restart your browser.



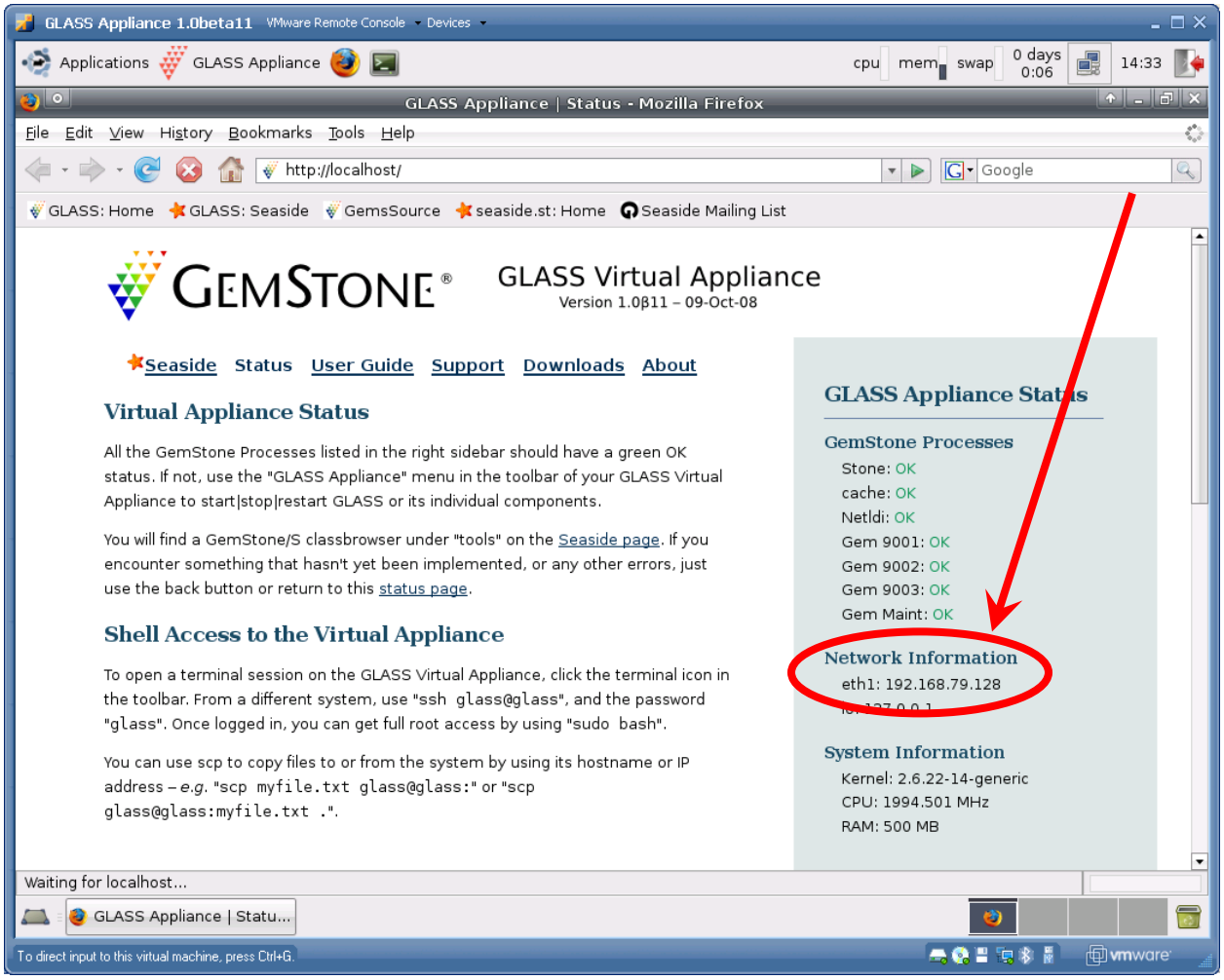
If the browser does not reopen to the same place, restart the VMware Infrastructure Web Access, select the GLASS Appliance in the Inventory list, select the Console tab, and click anywhere in the black area to open the virtual machine.



This will open a new window with a splash screen while the console is connecting to the virtual machine.

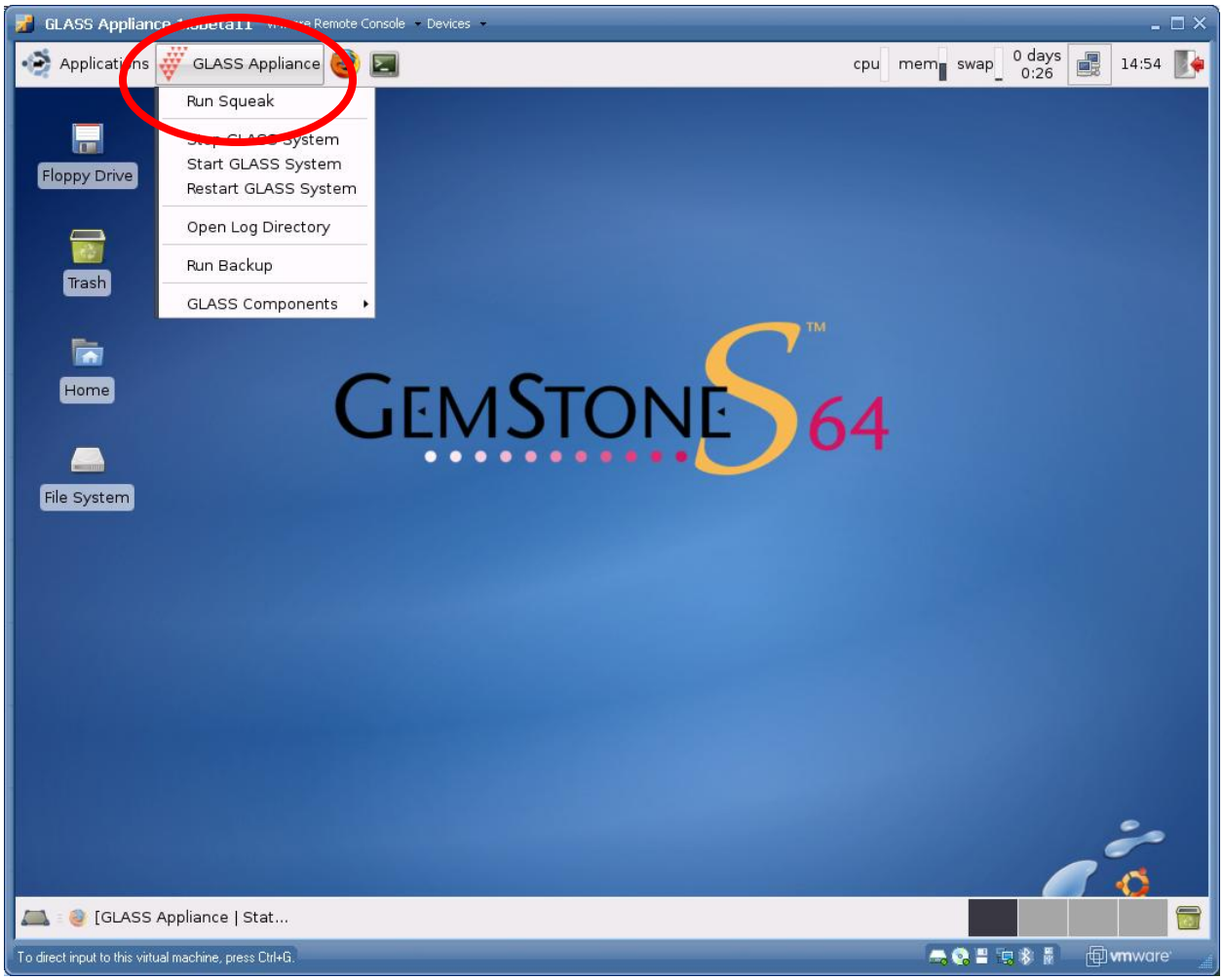


Once the console connects to the virtual machine, you should have a window showing what you would see if the GLASS Virtual Appliance were running on a real machine. In this case we have Linux with a GUI and a Firefox browser launched looking at http://localhost/. This is host is not your Windows machine, but a 'new' ('virtual') machine running in VMware that has its own operating system (Ubuntu Linux), its own web server (Apache), and a GemStone Smalltalk system already configured to start when the operating system starts. This machine has its own IP address (in my case it is 192.168.79.128, but it might be different for you). Note the series of green 'OK's for the seven GemStone processes.

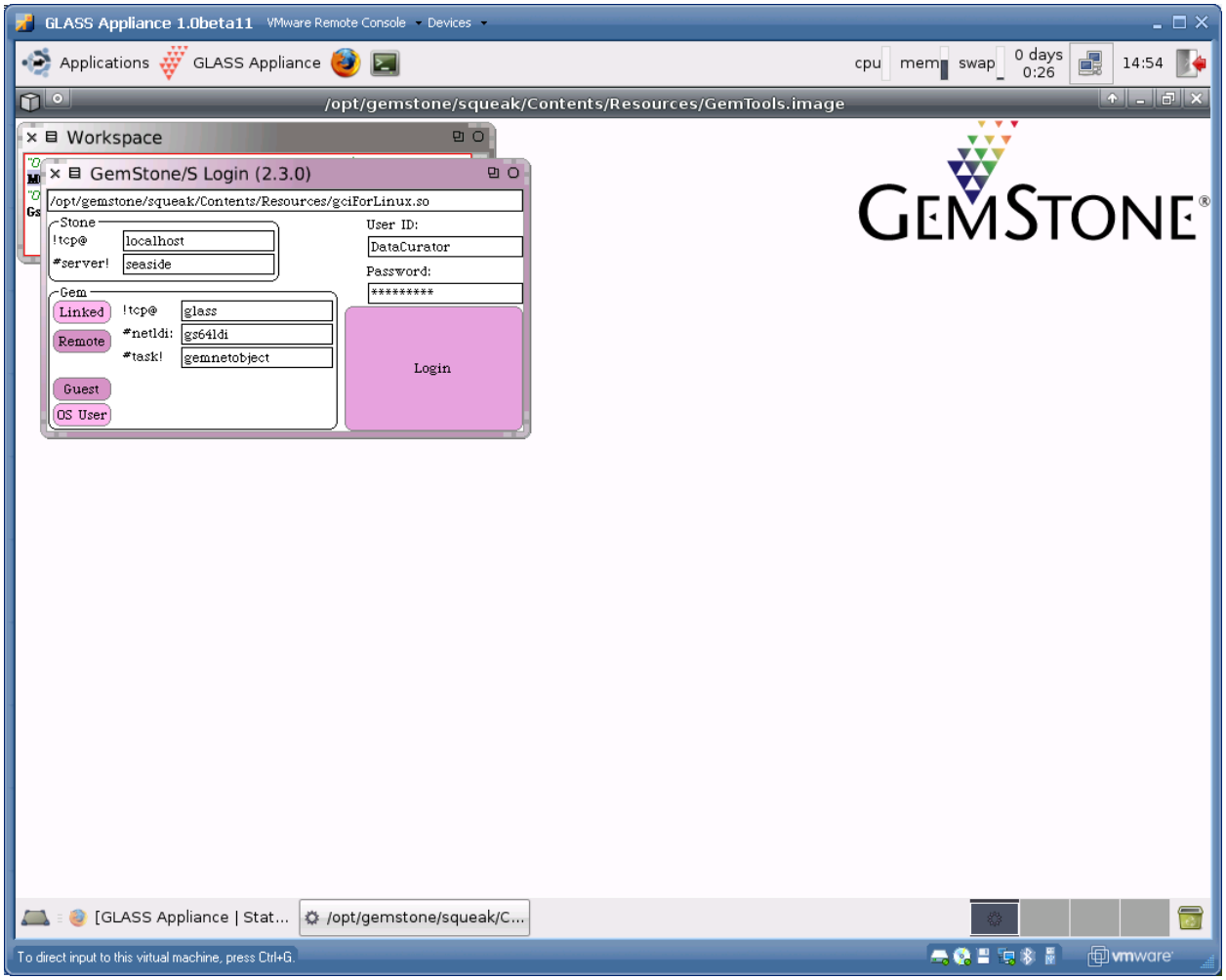


Copy the IP address out of the GLASS Appliance and past it into a web browser outside the virtual machine. Note that the web server in the virtual machine now is serving a page to you outside the virtual machine. You can start to think of the virtual machine as a host running in your company's server room that you access over the web using TCP/IP.

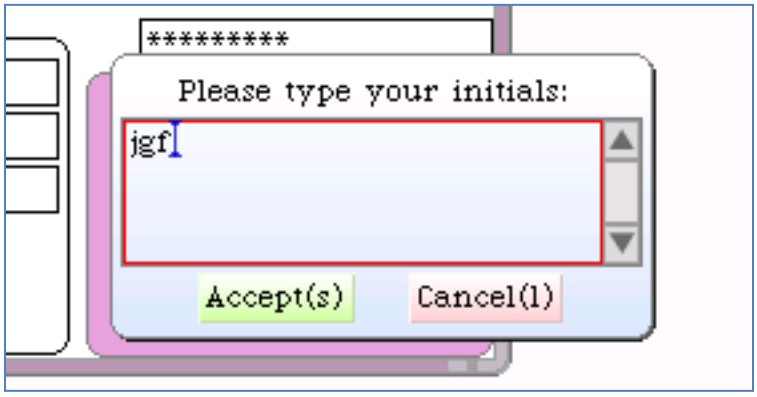
Returning to the virtual machine console, you can minimize Firefox and see the Linux desktop with menus. Select 'Run Pharo' from the 'GLASS Appliance' menu to start the GemTools application.



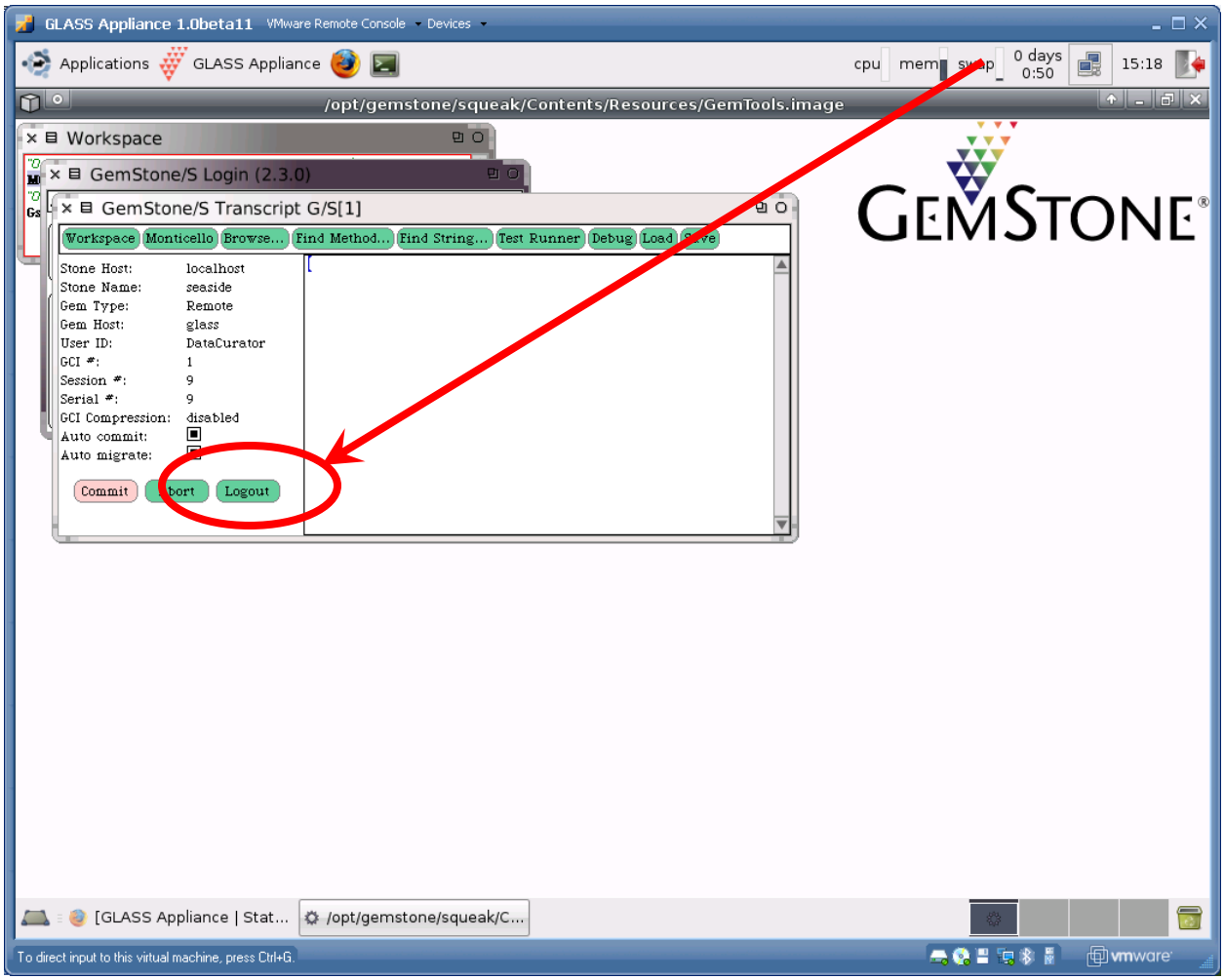
The GemTools application that ships with GLASS-Appliance-1.0beta11 is a Pharo application that fills the screen. Click the 'Login' button



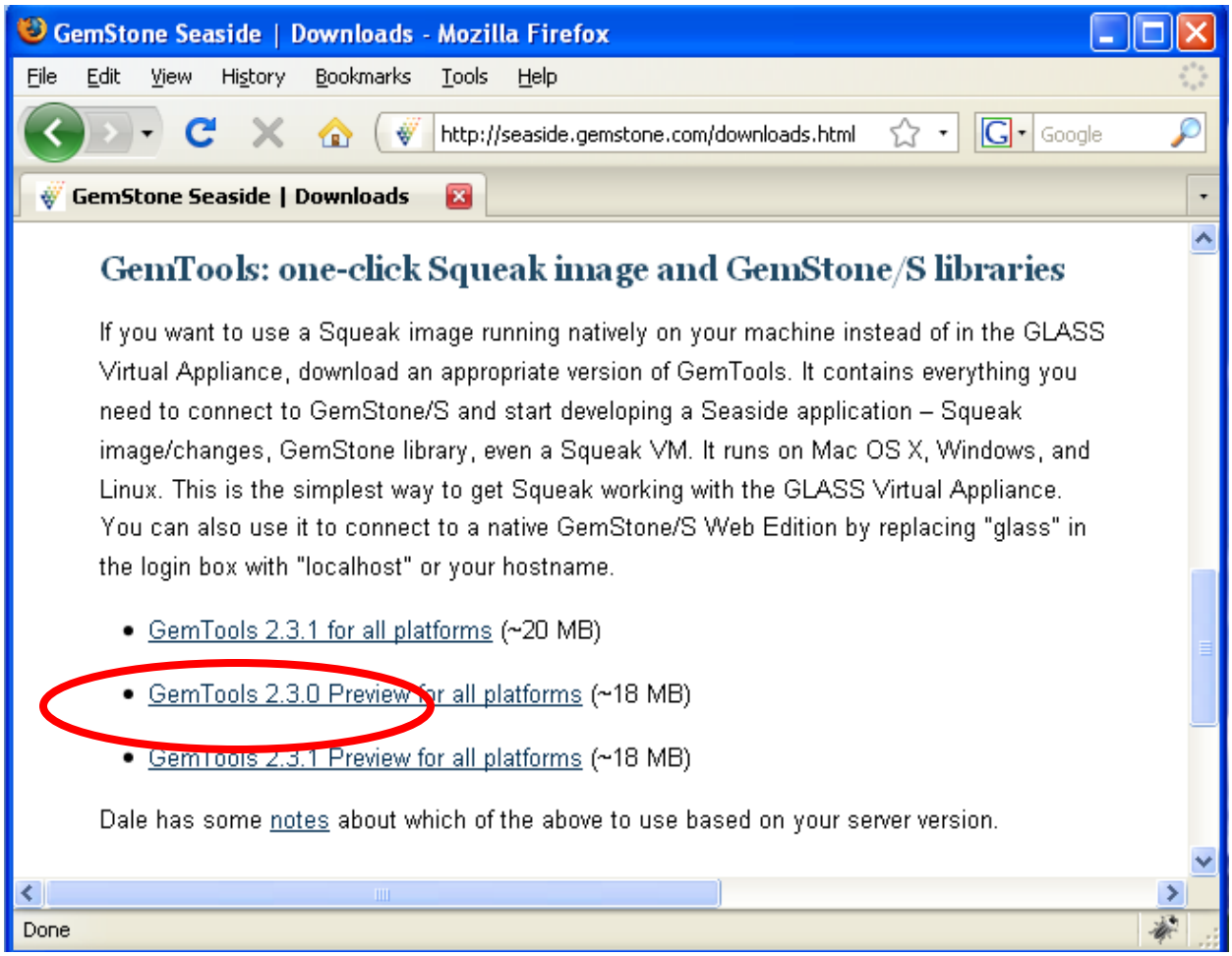
When prompted, type your initials (this is used by the Monticello code management tools to identify authors of methods). After typing your initials (by convention, in lowercase), click the 'Accept(s)' button.



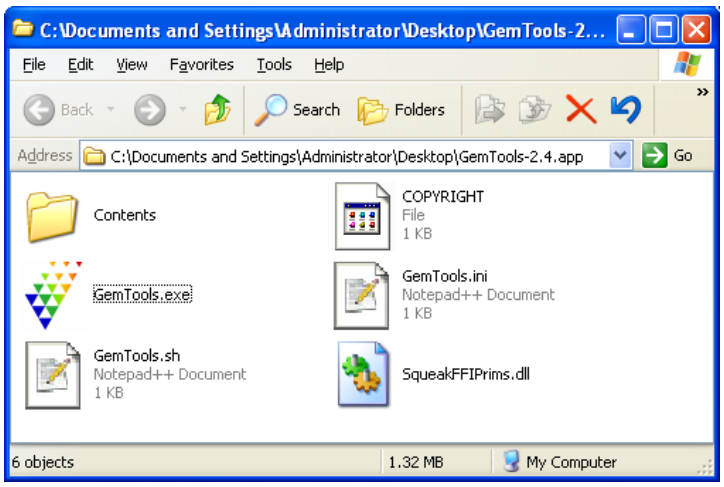
After you login, you should see a 'Transcript' window. We will do most of our interaction with the system using GemTools running outside the appliance, so go ahead and click the 'Logout' button on the Transcript.



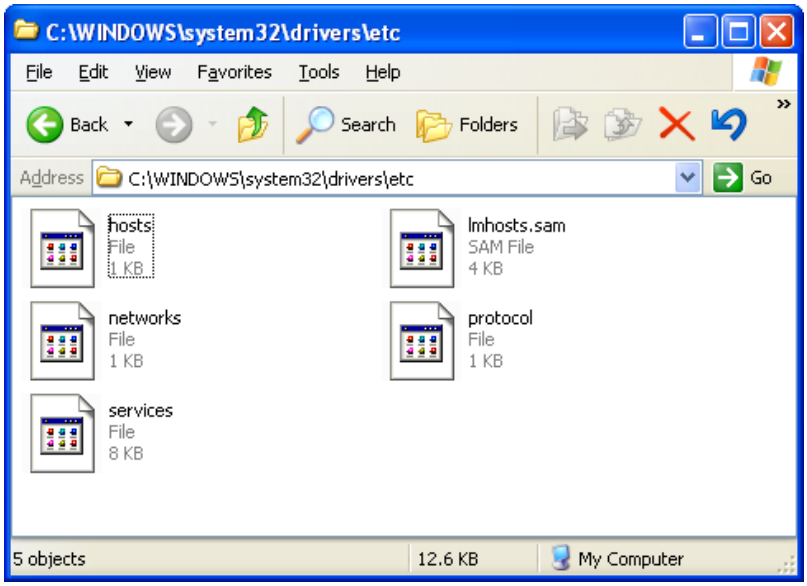
The tools have been substantially updated since the version included with the appliance. To get the latest tools, navigate to the GemStone downloads page where you got the appliance and scroll down to the bottom. Click the 'GemTools 2.3.0 Preview for all platforms' link to download a zip file containing the tools package. (This version matches the server executables in the virtual appliance.)



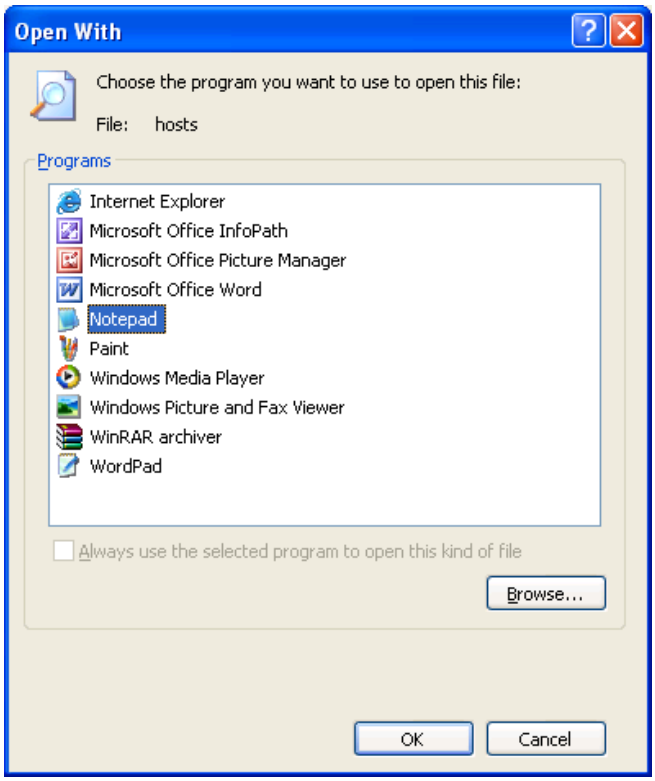
When you unzip this package it will consist of a folder (or directory) containing various pieces. On the Macintosh, the entire folder will appear as an executable package. Double-click the icon to launch the tools. Otherwise, open the folder and launch 'GemTools.exe' on Window or 'GemTools.sh' on Linux.



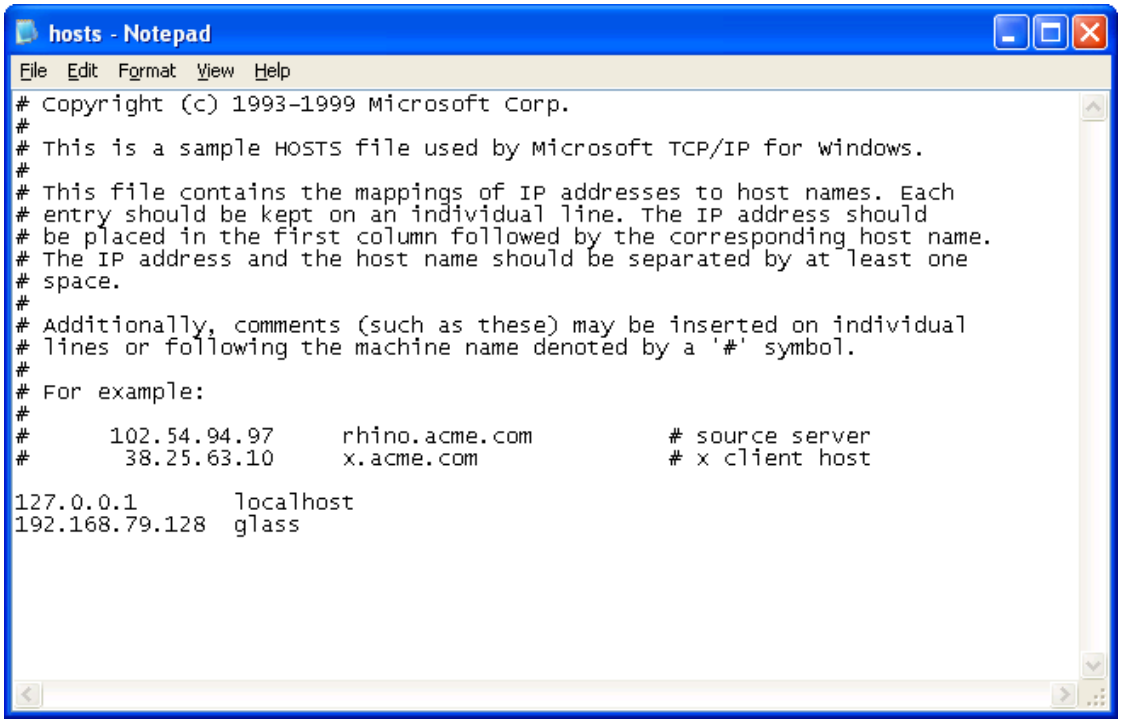
Before we can connect to GemStone on the virtual machine, we need to edit a few files that are used by the local machine to handle networking. First, we need to add the virtual machine to the hosts file so we can refer to it by name. On Linux or Macintosh you should find 'hosts' in /etc/ and you can edit it from a terminal using the command 'sudo vi /etc/hosts'. We will give more detailed instructions to our friends on Windows. On Windows, open a Windows Explorer to C:\WINDOWS\system32\drivers\etc and double-click on the 'hosts' icon.



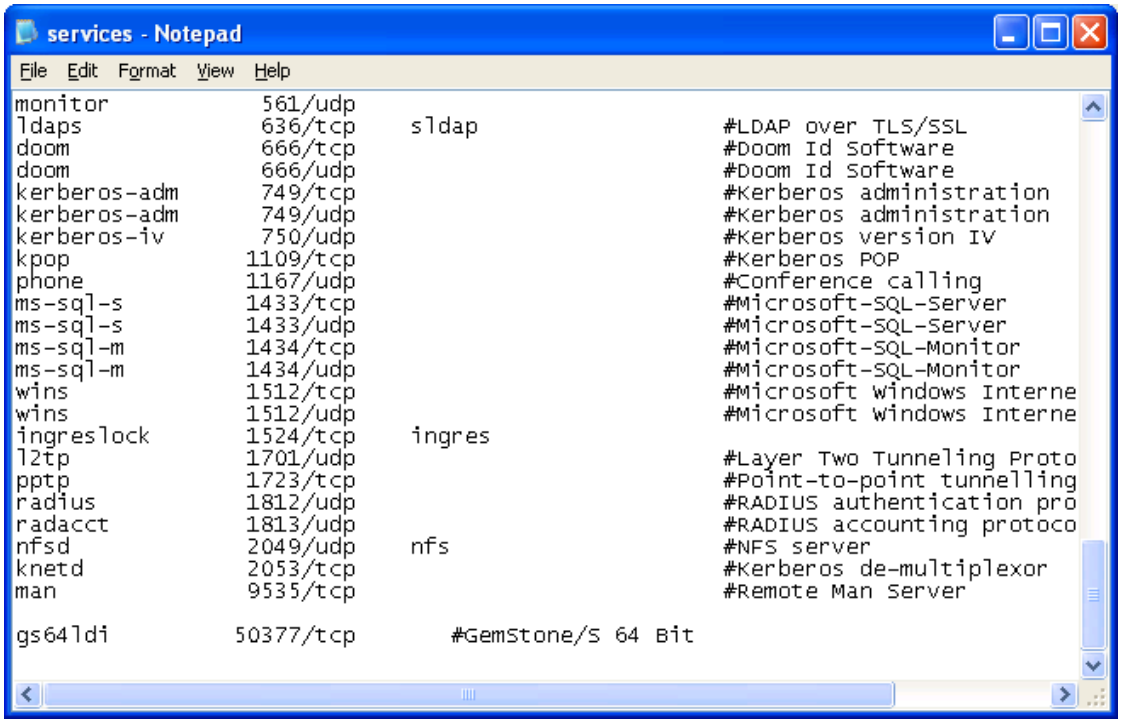
When asked what application to use to open the file, select 'Notepad' and click OK.



Add a line at the end with the IP address of the virtual machine (shown in the web browser in the virtual machine's console) and the word 'glass.' In my case the IP address is 192.168.79.128 but in your case it might be different. Save the file and close.



Open the ‘services’ file in the same manner and add ‘gs64ldi 50377/tcp #GemStone/S’ to the end. Save the file and close Notepad.

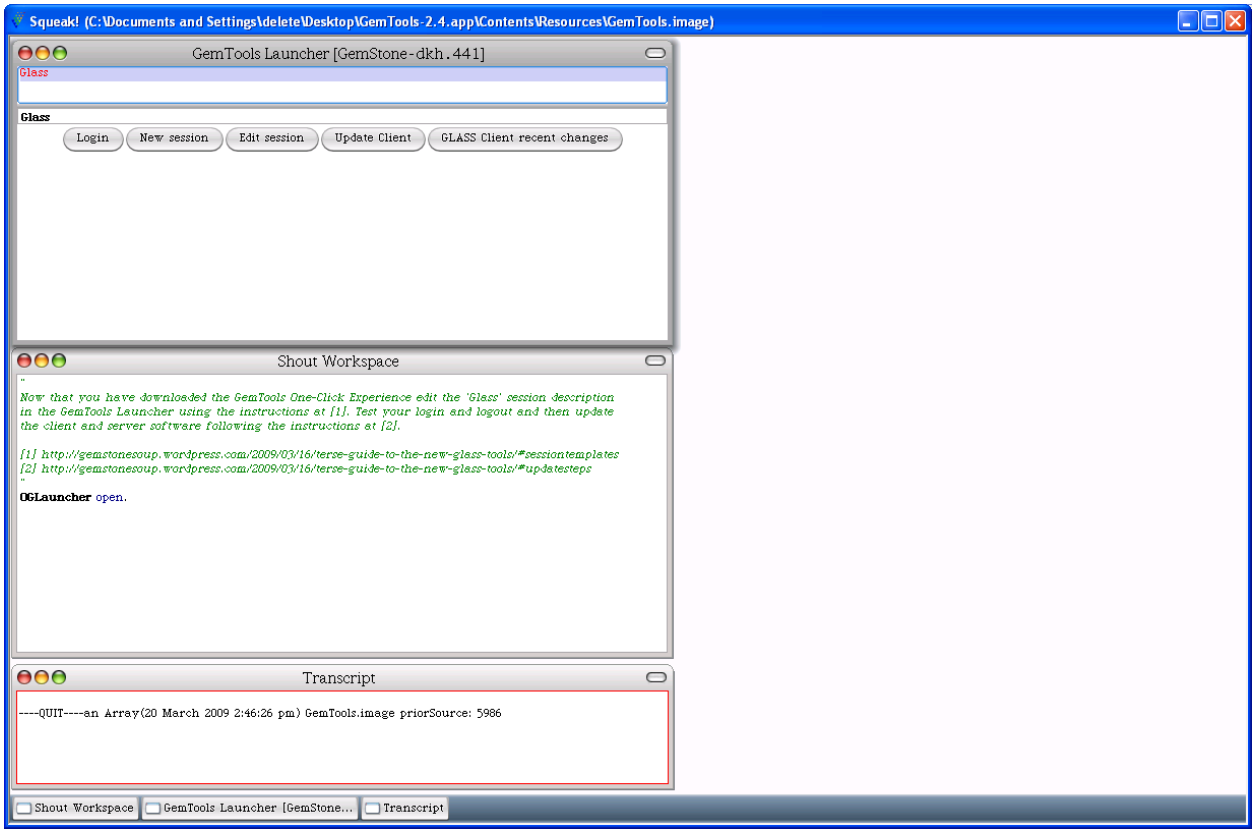


Now return to the GemTools application and launch it. This will launch Pharo with an image that contains three windows: (1) a GemTools Launcher; (2) a Shout Workspace; and (3) a Transcript. Instead of the earlier approach of using the Pharo ‘Seaside One-Click Experience’ to create and serve web pages, we are now using Pharo to run a set of tools that give us access to a GemStone system.

In the Shout Workspace we have the Smalltalk expression that was evaluated in this Pharo system to open the GemTools Launcher shown above the Workspace. Otherwise the Workspace has some instructions and a reference to Dale’s ‘Terse Guide to the New Glass Tools’ (which will give more detail than we will give here). Once you have copied the link, you probably should close the Workspace to avoid confusion (since you should not need to evaluate expressions in Pharo.

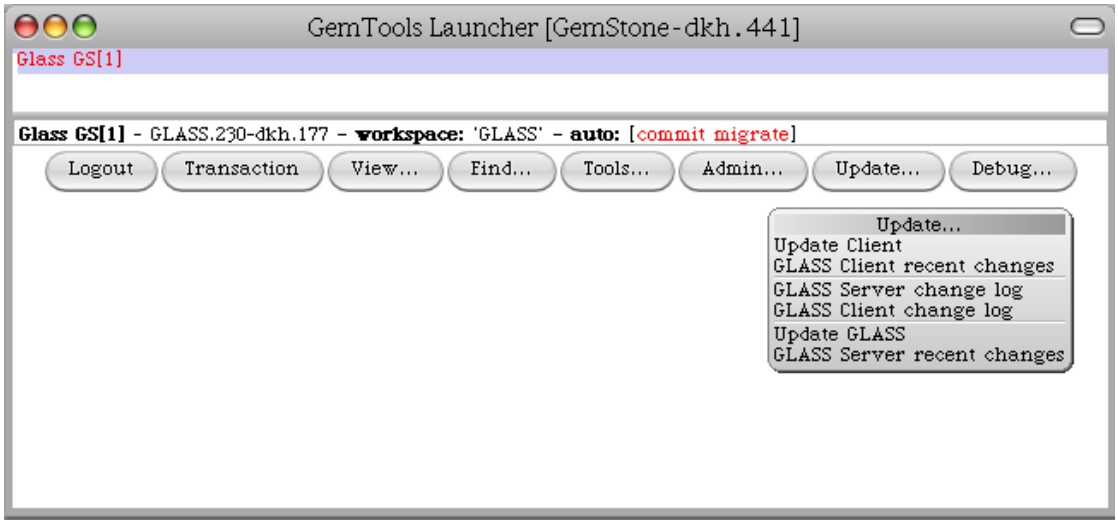
The Transcript will show notes about various operations that are happening.

The GemTools Launcher has a pre-defined session that connects to a host named ‘glass’ using a port named ‘gs64ldi’ and asks for a connection to a stone named ‘seaside.’ The reason we added ‘glass’ to the hosts file and ‘gs64ldi’ to the services file is so that these identifiers are properly defined. With ‘Glass’ selected in the list at the top of the GemTools Launcher, click the ‘Login’ button.

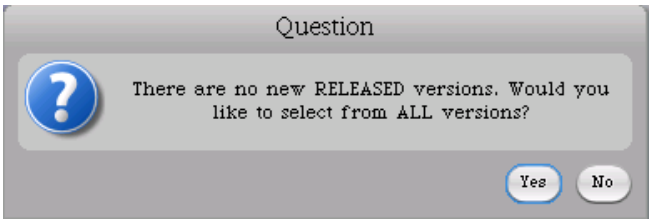


When prompted, enter your initials so that Monticello can track any edits to code.

Once you are logged in to the GLASS system, the GemTools Launcher will change to show options available when logged in to the system. If you click the 'Update' button you will get a menu of options.

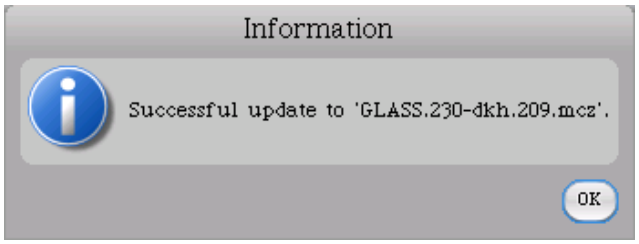


Select first command, 'Update Client' to update the GemTools code in Pharo. If the client is up-to-date then you will get a dialog offering you the option of selection from non-released version. Click 'No' to stay with the current code.



Repeat the update process and select the option to 'Update GLASS.' This should provide a list of released versions available. Select the most recent.

Once the update process completes, you should be informed of that fact.



At this point you can logout and quit the GemTools Pharo application.

If you have are running Macintosh OS X 10.6 or a 64-bit version of Linux you have the option of installing GemStone as a native application on your machine. This approach is a bit more complex to get right, but allows you to run GemStone without the virtual machine overhead. The following script shows how to do the install (see http://seaside.gemstone.com/downloads.html for details):

```
mkdir /opt/gemstone /opt/gemstone/locks /opt/gemstone/log
cd /opt/gemstone
curl http://seaside.gemstone.com/scripts/installGemstone.sh > \
   installGemstone.sh
./installGemstone.sh
```

This script will prompt you for your password so it can update kernel memory settings (feel free to study the script if you have any concerns). After the script completes, continue with the following to install Seaside 3.0:

```
source /opt/gemstone/product/seaside/defSeaside
startnet
startGemstone
```

Now to login using a GUI.

```
curl http://seaside.gemstone.com/squeak/GemTools-1.0-beta.8-244x.app.zip > \
   GemTools.zip
unzip GemTools.zip
open GemTools-1.0-beta.8-244x.app/
```

and the server is started, use 'startSeaside_Hyper 8000' to start a web server. In a web browser go to http://localhost:8000/seaside.

If you have are running Macintosh OS X 10.6 or a 64-bit version of Linux you have the option of installing GemStone as a native application on your machine. This approach is a bit more complex to get right, but allows you to run GemStone without the virtual machine overhead. The following script shows how to do the install (see http://seaside.gemstone.com/downloads.html for details):

```
mkdir /opt/gemstone \
   /opt/gemstone/backups /opt/gemstone/locks /opt/gemstone/log
cd /opt/gemstone
curl http://seaside.gemstone.com/jade/GemStone64Bit2.4.4.4-i386.Darwin.zip \
   >GemStone64Bit2.4.4.4-i386.Darwin.zip
curl http://seaside.gemstone.com/scripts/installGemstone.sh \
   >installGemstone.bak
sed s/2.4.4.1/2.4.4.4/ <installGemstone.bak | \
   sed s/x86_64.MacOSX/i386.Darwin/ >installGemstone.sh
chmod 700 installGemstone.sh
./installGemstone.sh
```

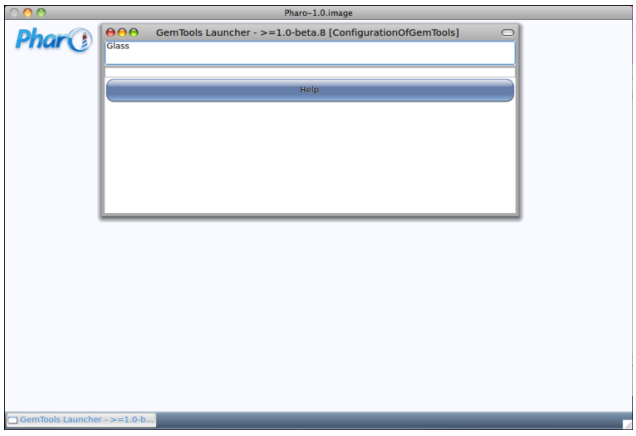
This script will prompt you for your password so it can update kernel memory settings (feel free to study the script if you have any concerns). After the script completes, continue with the following to start GemStone (you can use this script later to start GemStone again):

```
source /opt/gemstone/product/seaside/defSeaside
startnet
startGemstone
gslist -cvl
```

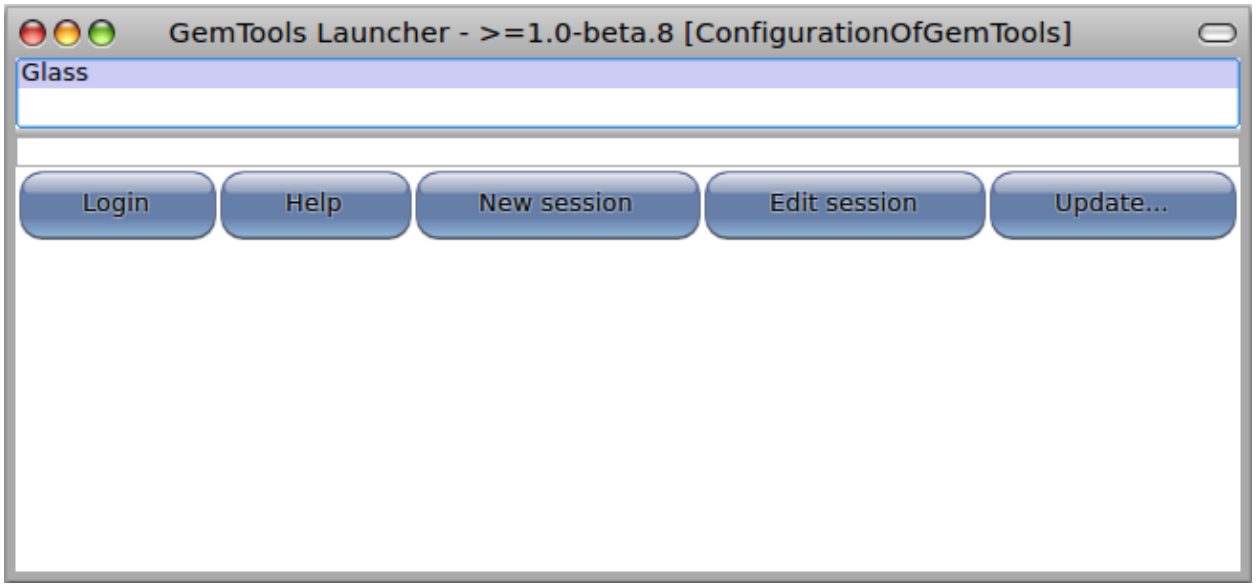
If things are working properly, you should see three processes running. Now to login using a GUI we need to download GemTools, a Pharo-based GUI for GemStone/S.

```
curl http://seaside.gemstone.com/squeak/GemTools-1.0-beta.8-244x.app.zip > \
   GemTools.zip
unzip GemTools.zip
open GemTools-1.0-beta.8-244x.app/
```

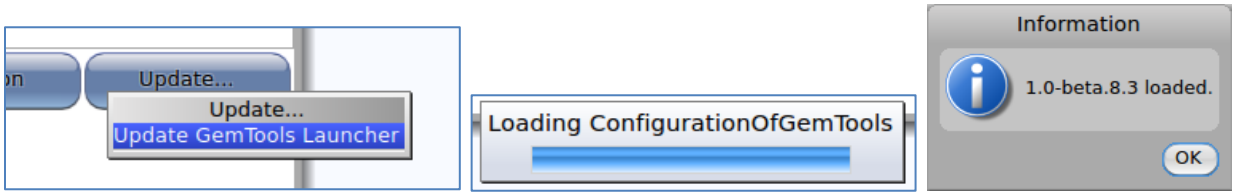
This should open GemTools with a single window, the GemTools Launcher:



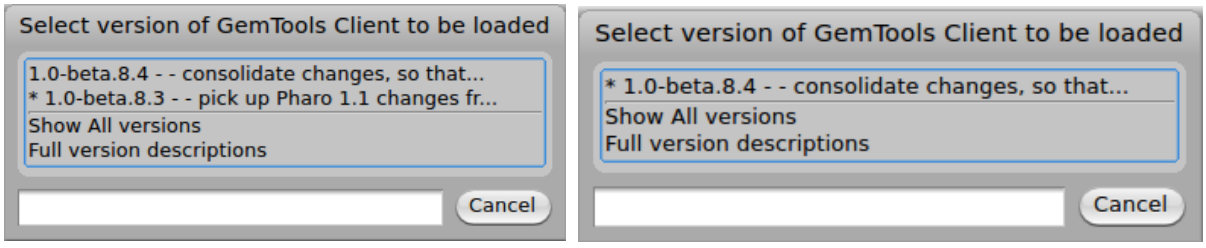
Click on the 'Glass' entry in the list area in the top section of the Launcher. This will change the single 'Help' button into a row of five buttons.



Click on the 'Update…' button and then select 'Update GemTools Launcher' from the update menu. As this is working it will show some progress dialogs. When it is done it will inform you what version is loaded.

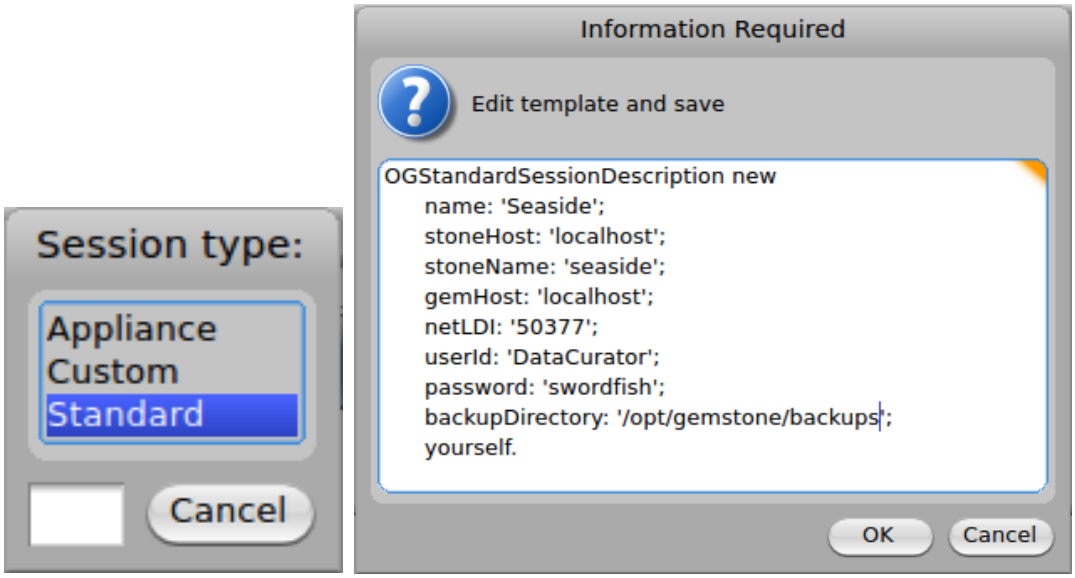


Repeat the above process and select the top line (here, 1.0-beta.8.4). This will open a Pharo Help window (you may close it), and inform you that version 1.0-beta.8.4 is loaded. Continue to repeat the process until the top line has a * next to it (indicating that the most recent 'released' version is loaded). At this point you may click 'Cancel' since we do not need to update the client (we will update the server soon).

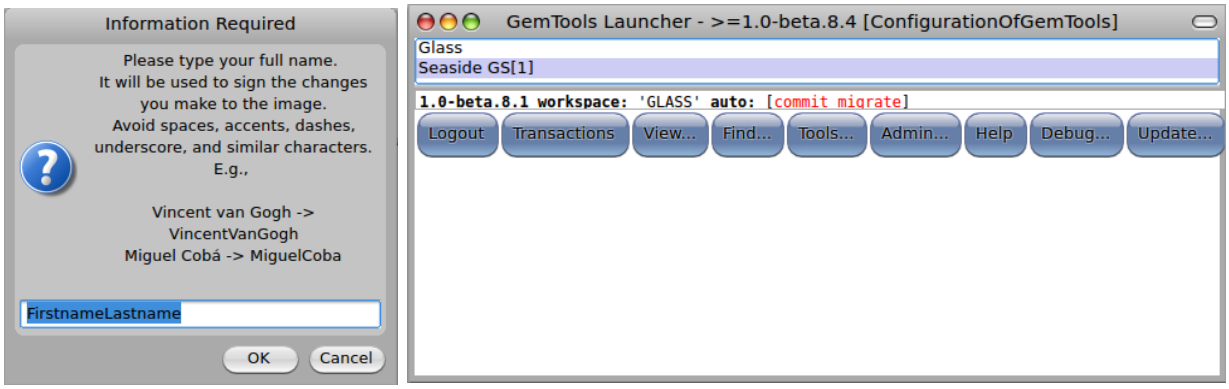


After the update process we will create the login parameters for a new session. Click the 'New Session' button in the center of the five buttons and select 'Standard' from the list of types. Edit the template to

change the name to 'Seaside', replace the stoneHost and the gemHost with 'localhost', and set the backup directory to '/opt/gemstone/backups'. When you have made the edits click 'OK'.

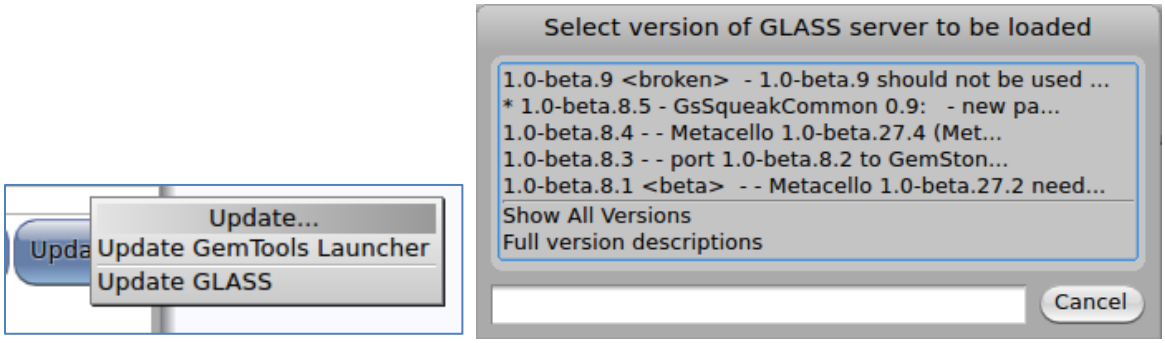


This will add a new 'Seaside' line to the list of defined session types in the Launcher. Select the new session type by clicking on the 'Seaside' line and then click on the 'Login' button. This should present you with a dialog asking for your name. Enter your name without spaces and without punctuation characters, and click 'OK'. A successful login should result in a change in the buttons to include a set starting with 'Logout' and ending with 'Update'.

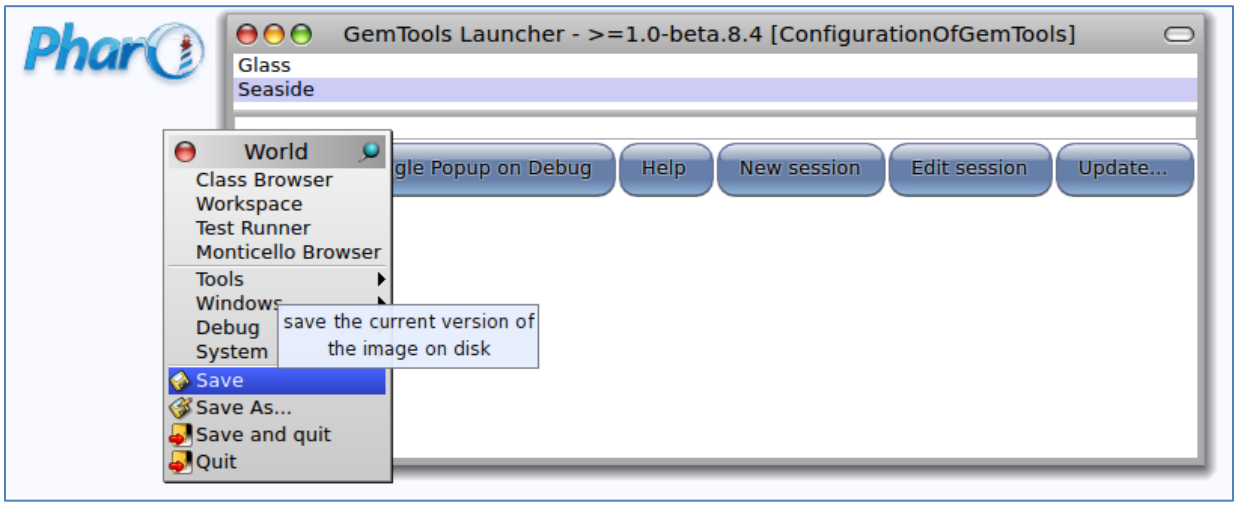


Note that after the login, there is an info line that tells us that we have 1.0-beta.8.1 loaded into the server (GemStone/S), we have a workspace named 'GLASS', and GemTools is configured for auto-commit and auto-migrate. Auto-commit means that each server-related action in GemTools will be followed by a commit. This is the default mode and ensures that you do not lose any work. The auto-migrate mode means that when you change the schema for a class (typically by adding an instance variable), GemTools will scan the database for all instances of that class and migrate the instances to the new schema. This setting provides behavior comparable to what you experience with other Smalltalks but is associated with non-trivial overhead in GemStone since the object space is disk-based rather than RAM based and can be quite large.

Click on the 'Update' button and select 'Update GLASS'. Select 1.0-beta.8.5 (even though it has an asterisk by it implying that it is already loaded—it is not loaded, this is a bug in beta.8.1). When the update finishes, note that the info line shows 1.0-beta.8.5 as the current version.



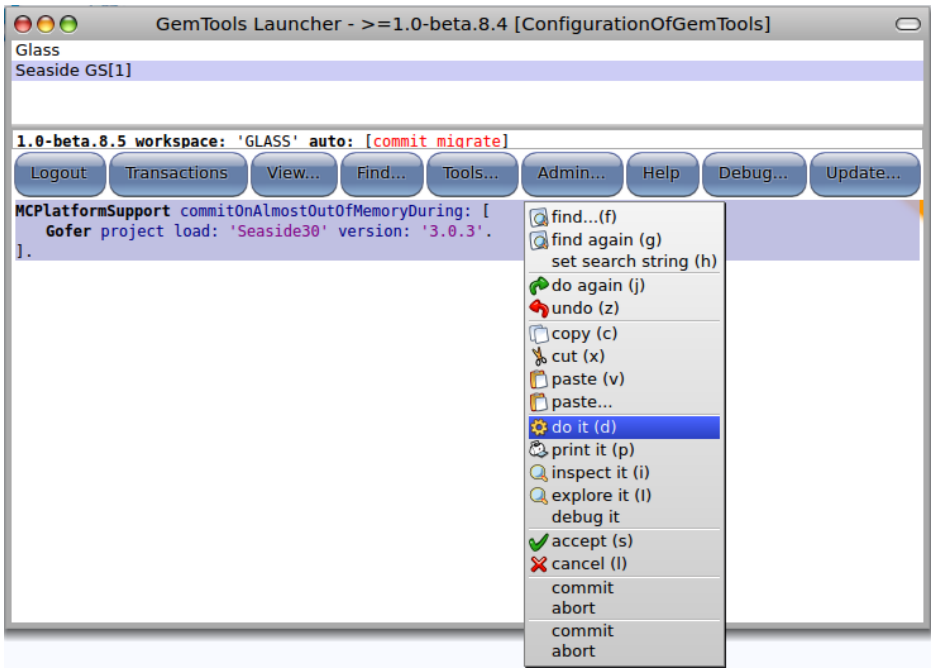
At this point, click the 'Logout' button to disconnect from the database server. Since we have added a session type and updated the client, we want to save the Pharo image. Click on the desktop and from the World menu select 'Save'.



Unlike the Seaside One-Click Experience, GemStone does not have Seaside pre-loaded. This is because you might want to load only part of Seaside. In our case, we will go ahead and load everything. Following the instructions at http://code.google.com/p/glassdb/wiki/Seaside30Configuration, start by logging in (click the Login button) and enter the following text into the workspace (the text area below the buttons):

```
MCPlatformSupport commitOnAlmostOutOfMemoryDuring: [
   Gofer project load: 'Seaside30' version: '3.0.3'.
].
```

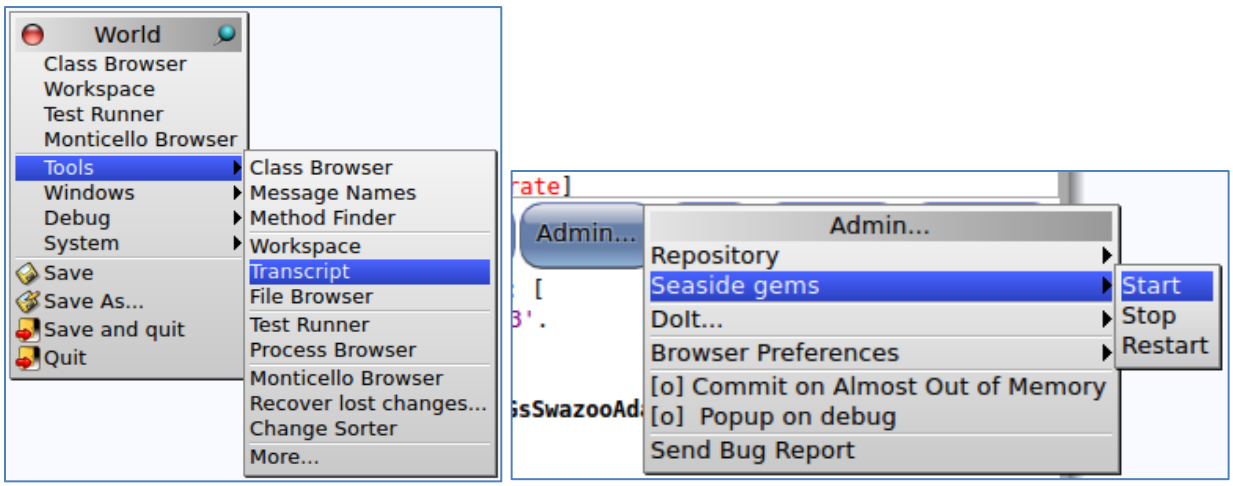

Select all the text (<Ctrl>+<S>), and then right-click to get a context menu and select 'do it (d)' from the menu (or use <Ctrl>+<D>). The cursor will change to something like a GemStone/S logo (six triangles in a triangle) and will run for some time (several minutes).



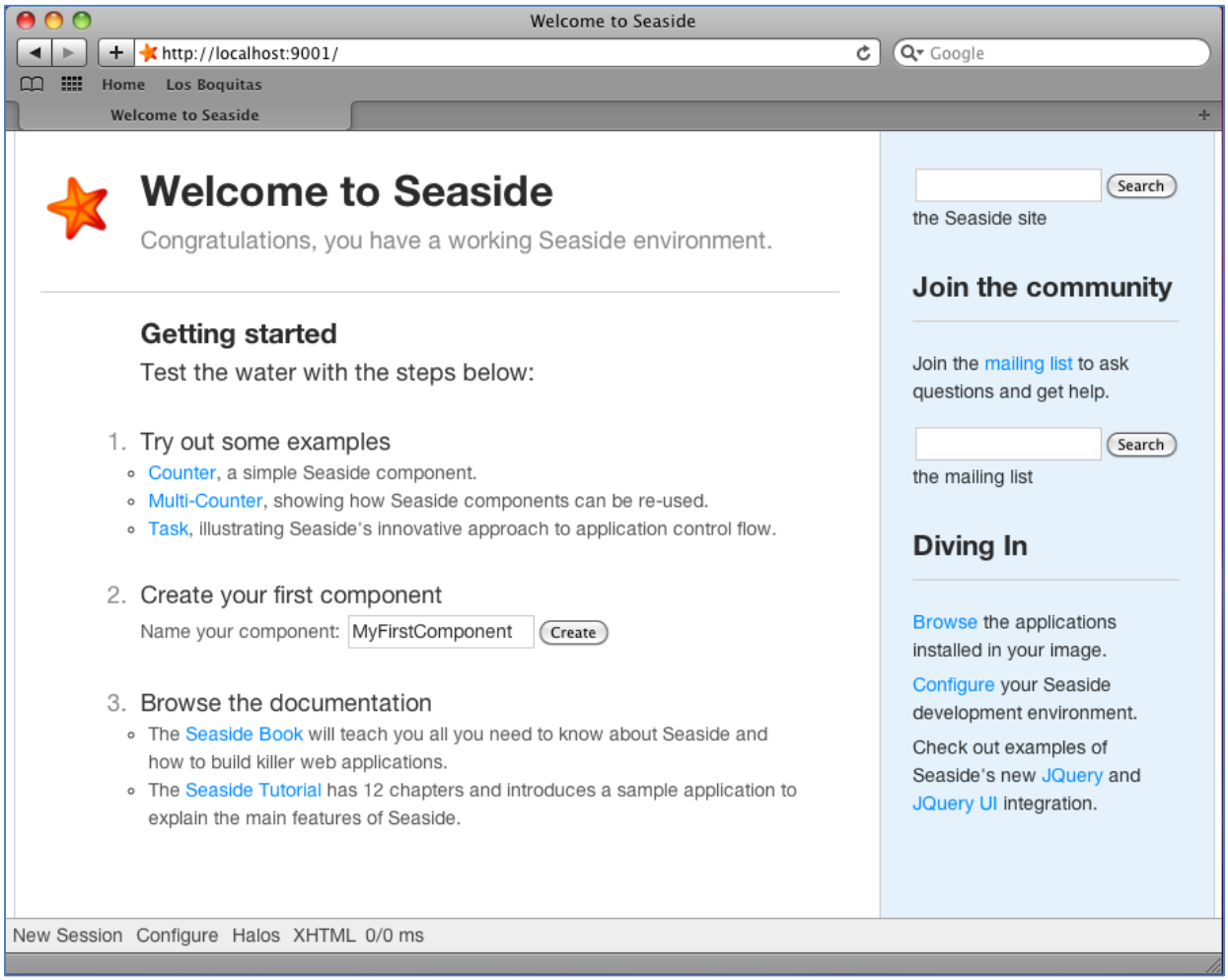
Now we will start a web server. Evaluate the following expression in the workspace:

```
WAGemStoneRunSeasideGems default adaptorClass: WAGsSwazooAdaptor.
```

This informs GemStone that you want to run a web server in GemStone. Before starting the web server, open a Pharo Transcript by clicking on the Pharo desktop to get the World menu, select 'Tools' then 'Transcript'. Then, from the Launcher window, click on the 'Admin...' button to get the Admin menu. Select 'Seaside gems' and 'Start'. The Transcript should show that four gems have been started, three WAGsSwazooAdaptor gems (on ports 9001 to 9003) and one maintenance gem.



Open a web browser on http://localhost:9001/ and you should see the Seaside welcome page.



This web page is being served by Swazoo, a Smalltalk web server in GemStone, but it is not the same virtual machine that is handling your GemTools window. It is another OS process. To get an idea of what is running on your system, enter the following command in a Terminal:

```
ps -ef | grep gemston[e] | cut -c-120
```

```
vienna:gemstone jfoster$ ps -ef | grep gemston[e] | cut -c-120
  587   792     1   0   0:00.03 ??         0:00.04 /opt/gemstone/GemStone64Bit2.4.4.4-i386.Darwin/sys/netldid gs64ldi -g
  587   827   334   0   0:51.24 ??         5:58.42 /opt/gemstone/GemTools-1.0-beta.8-244x.app/Contents/MacOS/Squeak VM O
  587  1663     1   0   0:01.44 ??         0:02.01 /opt/gemstone/GemStone64Bit2.4.4.4-i386.Darwin/sys/stoned seaside -z/
  587  1664  1663   0   0:01.51 ??         0:08.96 /opt/gemstone/product/sys/shrpcmonitor 'seaside@127.0.0.1' setlocatio
  587  1668  1663   0   0:00.21 ??         0:00.29 /opt/gemstone/product/sys/pgsvrmain seaside@127.0.0.1 0 1 -1
  587  1670  1663   0   0:00.46 ??         0:00.72 /opt/gemstone/product/sys/pgsvrmain TCP 50701 90
  587  1672  1663   0   0:12.18 ??         0:16.51 /opt/gemstone/product/sys/gem pagemanager 'seaside' -T 5000
  587  1676  1663   0   0:00.10 ??         0:00.50 /opt/gemstone/product/sys/gem reclaimgcgem 'seaside' 0 0 -T 5000
  587  1677  1663   0   0:00.05 ??         0:00.09 /opt/gemstone/product/sys/gem admingcgem 'seaside' -T 5000
  587  1678  1663   0   0:00.62 ??         0:08.57 /opt/gemstone/product/sys/gem symbolgem 'seaside' -T 10000
  587  1688   792   0   0:03.00 ??         3:41.70 /opt/gemstone/product/sys/gem TCP 50729 60
  587  2536     1   0   0:00.06 ??         0:00.29 /opt/gemstone/product/bin/topaz -l -T50000
  587  2537     1   0   0:00.02 ??         0:00.06 /opt/gemstone/product/bin/topaz -l -T50000
  587  2539     1   0   0:00.03 ??         0:00.10 /opt/gemstone/product/bin/topaz -l -T200000
  587  2540     1   0   0:00.02 ??         0:00.07 /opt/gemstone/product/bin/topaz -l -T50000
```


The first process, netldid (PID 792), is available to start gems for GemTools. The second process, Squeak (PID 827), is the VM that is running GemTools. The next process, stoned (PID 1663), is the primary database process and the following seven processes are started by the stone to handle various tasks (you can see that the third column, parent PID, is the stone). Next is a gem (PID 1688) started by the NetLDI (so the parent PID is 792). The remaining four processes are topaz and were started when you selected 'Start' from the 'Seaside gems' menu from the Admin menu.

Now we will explore using FastCGI with Apache as the web server. In a Terminal, create some directories and a new Apache config file. The Apache config file will be placed in a protected directory, so you will be prompted for an administrative password.

```
mkdir /opt/gemstone/apache /opt/gemstone/apache/logs \
   /opt/gemstone/apache/htdocs \
   /opt/gemstone/apache/htdocs/seaside1 \
   /opt/gemstone/apache/htdocs/seaside2 \
   /opt/gemstone/apache/htdocs/seaside3
echo '# Apache config file for Seaside using FastCGI in GemStone
<VirtualHost *:80>
   ProxyPreserveHost On     #make proxy rewrite urls in the output
   ProxyPass /favicon.ico !
   ProxyPass /css !
   ProxyPass /images !
   ProxyPass /scripts !
   ProxyPass / balancer://gemtrio/
   ProxyPassReverse / balancer://gemtrio/
   <Location /balancer-manager>
      SetHandler balancer-manager
      Order deny,allow
      Deny from all
      Allow from localhost
   </Location>
</VirtualHost>

<Proxy balancer://gemtrio>
   BalancerMember http://localhost:8081
   BalancerMember http://localhost:8082
   BalancerMember http://localhost:8083
</Proxy>

Listen 8081
<VirtualHost *:8081>
   CustomLog /opt/gemstone/apache/logs/seaside-vhost-8081-access.log combined
   ErrorLog  /opt/gemstone/apache/logs/seaside-vhost-8081-error.log
   LogLevel  warn
   DocumentRoot /opt/gemstone/apache/htdocs/seaside1/
   <Location "/">
      Order deny,allow
      Deny from all
```



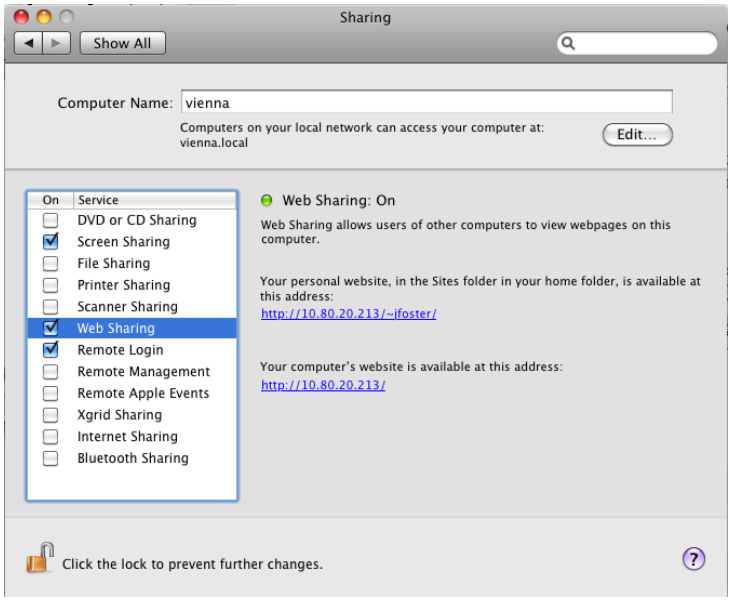
```
      Allow from localhost
   </Location>
</VirtualHost>

Listen 8082
<VirtualHost *:8082>
   CustomLog /opt/gemstone/apache/logs/seaside-vhost-8082-access.log combined
   ErrorLog  /opt/gemstone/apache/logs/seaside-vhost-8082-error.log
   LogLevel  warn
   DocumentRoot /opt/gemstone/apache/htdocs/seaside2/
   <Location "/">
      Order deny,allow
      Deny from all
      Allow from localhost
   </Location>
</VirtualHost>

Listen 8083
<VirtualHost *:8083>
   CustomLog /opt/gemstone/apache/logs/seaside-vhost-8083-access.log combined
   ErrorLog  /opt/gemstone/apache/logs/seaside-vhost-8083-error.log
   LogLevel  warn
   DocumentRoot /opt/gemstone/apache/htdocs/seaside3/
   <Location "/">
      Order deny,allow
      Deny from all
      Allow from localhost
   </Location>
</VirtualHost>

LoadModule fastcgi_module libexec/apache2/mod_fastcgi.so

FastCgiExternalServer /opt/gemstone/apache/htdocs/seaside1 \
   -host localhost:9001 -pass-header Authorization
FastCgiExternalServer /opt/gemstone/apache/htdocs/seaside2 \
   -host localhost:9002 -pass-header Authorization
FastCgiExternalServer /opt/gemstone/apache/htdocs/seaside3 \
   -host localhost:9003 -pass-header Authorization
' | sudo tee /etc/apache2/other/seaside.conf
```

At this point we need to restart so it can pick up the new configurations. Ideally, we should be able to do this from Terminal using the following:

```
sudo apachectl restart
```

If this does not complain, then you should be fine. If you get an invalid argument error ('/usr/sbin/apachectl: line 82: ulimit: open files: cannot modify limit: Invalid argument') then see http://www.456bereastreet.com/archive/201011/mac_os_x_1065_broke_my_apachectl/. As an

alternative, open System Preferences, select Sharing, and then uncheck and then check the 'Web Sharing' checkbox. This will stop and start Apache.

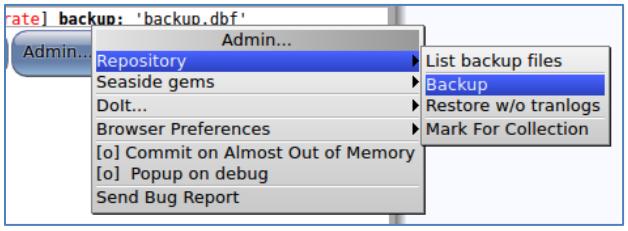


In your GemTools workspace, set Seaside to use FastCGI (<Ctrl>+<D>):

```
WAGemStoneRunSeasideGems default adaptorClass: WAFastCGIAdaptor.
```

Now, from the Admin menu, select 'Seaside gems' and 'Start'. You should be able to go to a web browser and navigate to http://localhost/. Note that we did not need to provide a port (such as 8080 or 9001), because we are now using the web server provided as part of the Mac OS. This allows you to use Apache to serve static files (images, CSS, JavaScript, etc.), while using Seaside to serve the dynamic content. You can edit the config file to exclude other local directories.

Now would be a good time to do a backup. From the Launcher, click the 'Admin...' button, select 'Repository' and then 'Backup'. You can name your backup (or accept the default), and then click OK.



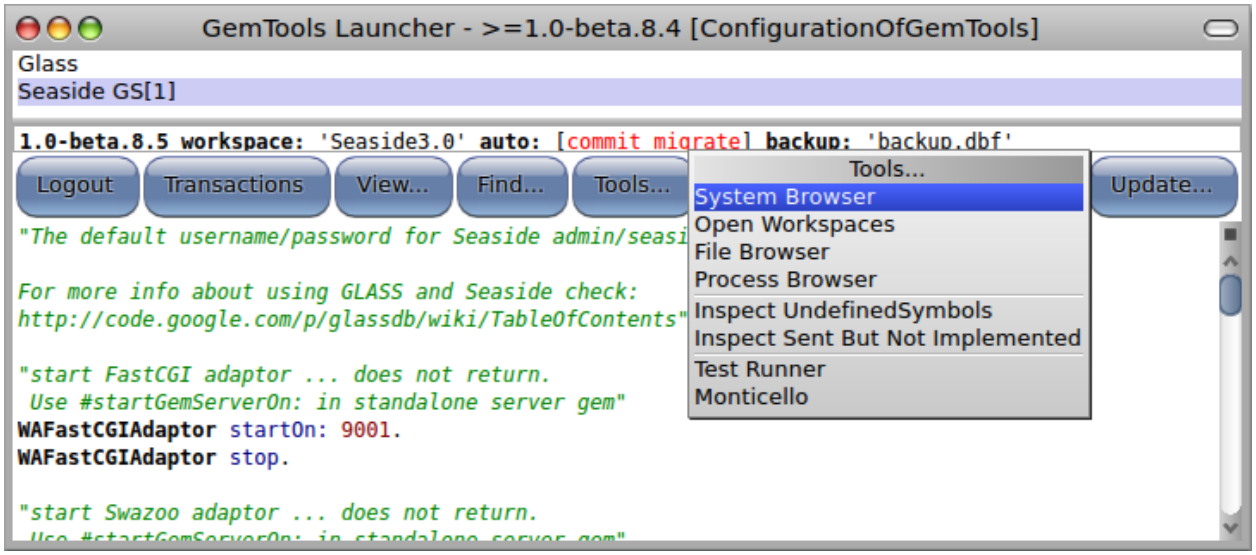
When you are done, stop the Seaside gems (from the Admin menu) and click the 'Logout' button. Then in a terminal you can stop GemStone with the following line:

```
source /opt/gemstone/product/seaside/defSeaside; stopGemstone; stopnetldi
```

You can learn more about GemTools at http://code.google.com/p/glassdb/wiki/GemTools.

Let's get started using GemStone.

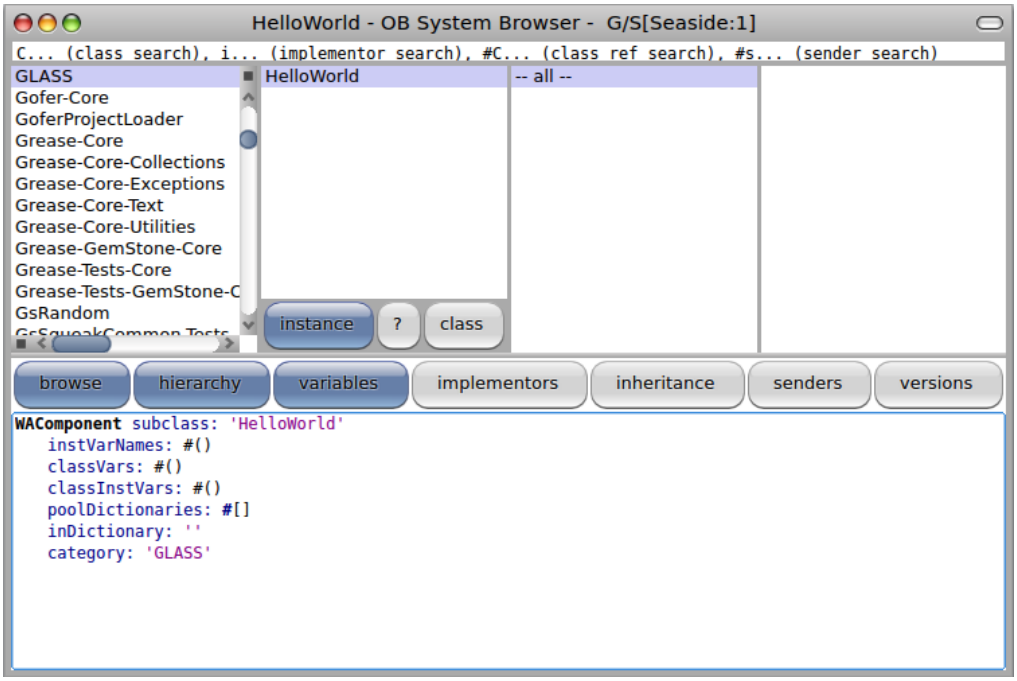
1. First we will quickly create a 'Hello World' application in GemStone.
    a. Start GemStone and the Seaside gems using the instructions from Chapter 16 (if it is not running) and launch GemTools.
    b. In the GemTools launcher, select Seaside, and click the 'Login' button. Enter your name if requested.
    c. Once logged in, click the 'Tools…' button and select 'System Browser'.



    d. This will open an OB System Browser showing GemStone code. Click in the first column to get a class creation template and enter the following in the text area:

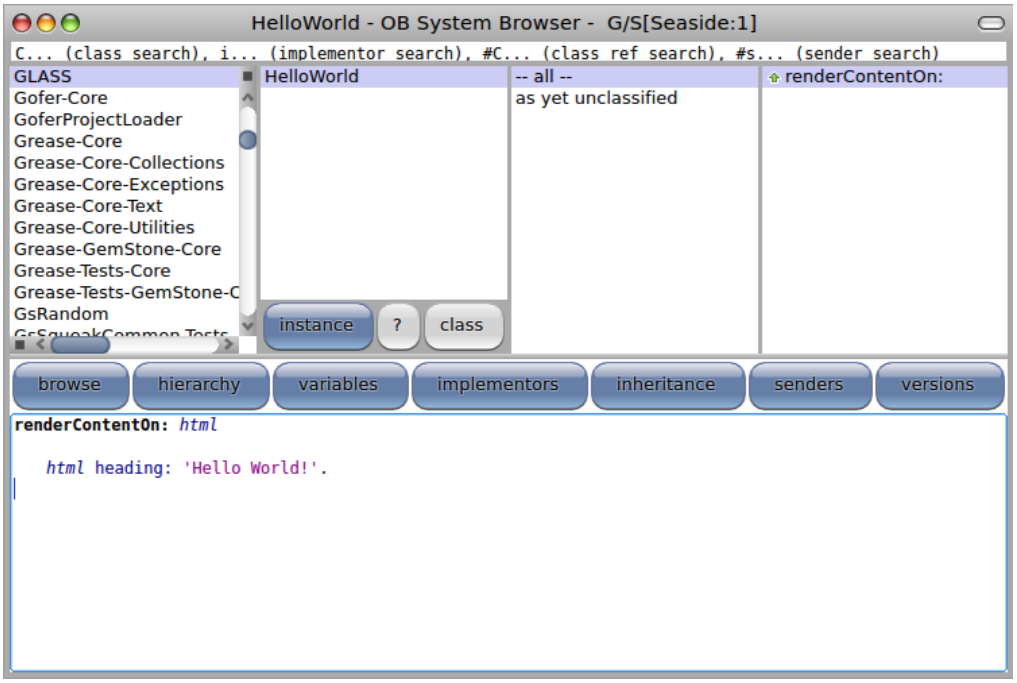
```
WAComponent subclass: 'HelloWorld'
   instVarNames: #()
   classVars: #()
   classInstVars: #()
   poolDictionaries: #[]
   inDictionary: ''
   category: 'GLASS'
```


e. This should update your System Browser to show the new class.



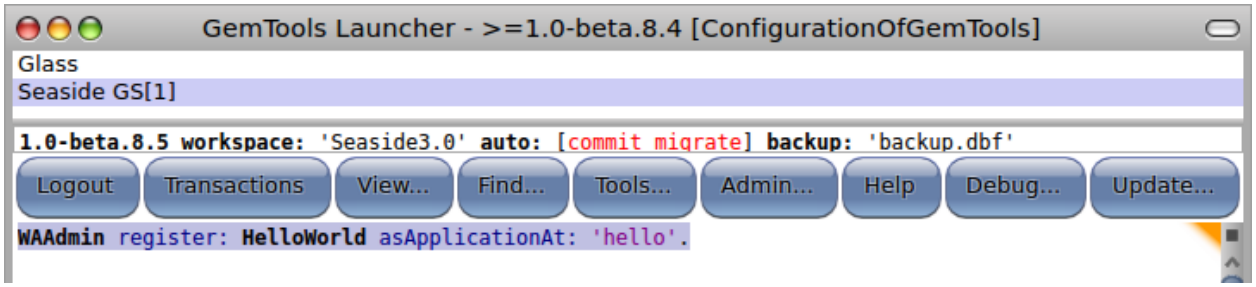
f. Click in the third column to change the text area from a class definition to a method template. Enter and save the render method.

```
renderContentOn: html

   html heading: 'Hello World!'.
```

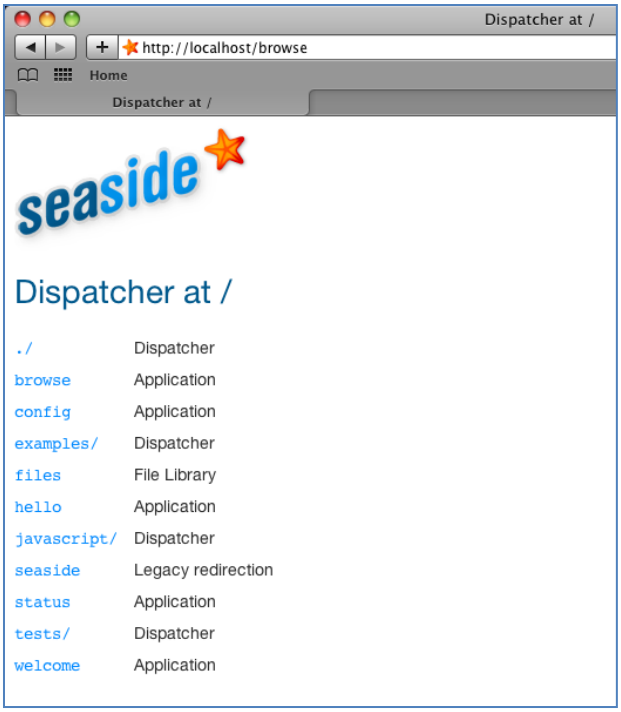


g. Register the component as an application. Select the GemTools Launcher, click in the text area below the buttons and enter the expression to register the application. Press <Ctrl>+<D> (for 'do-it') to evaluate the expression.

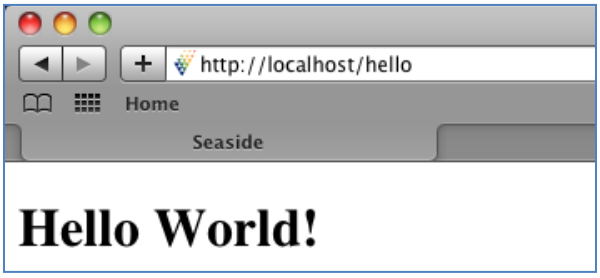
```
WAAdmin register: HelloWorld asApplicationAt: 'hello'.
```



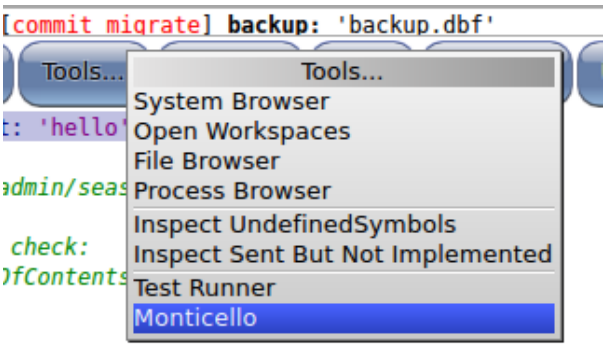
h. Open a web browser on http://glass/browse and note that 'hello' is listed.



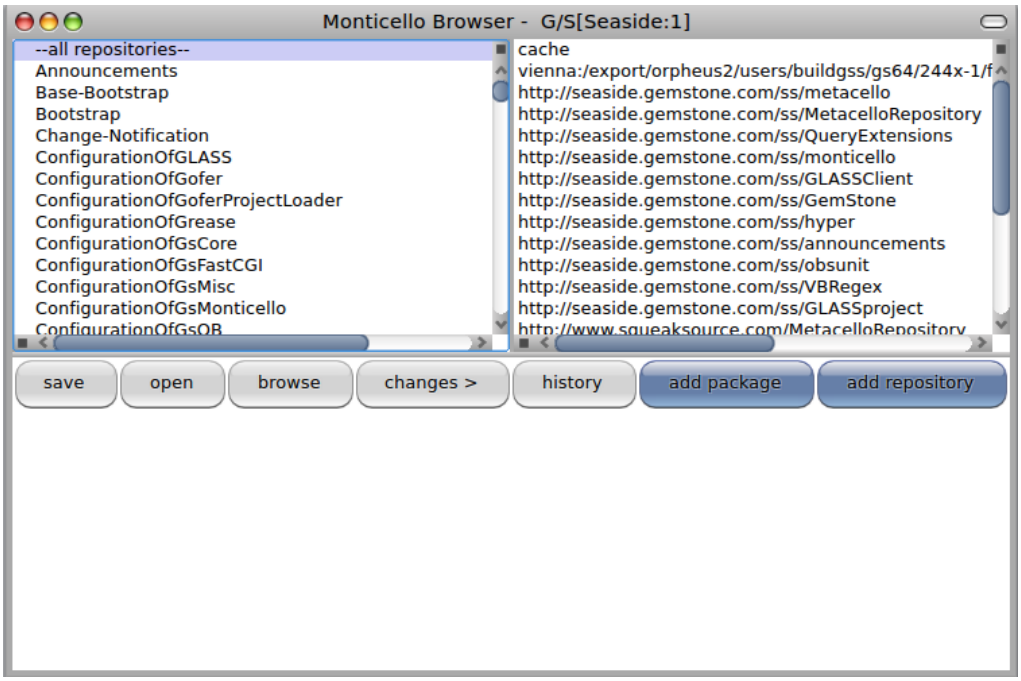
i. Click on the 'hello' link to get the application.



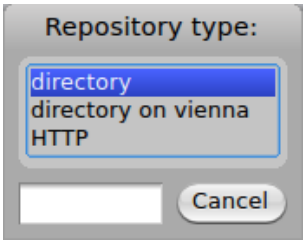
2. Now we will port the Boquitas application from Squeak.
    a. In Chapter 14 we versioned our code to a local disk-based Monticello repository. From the GemTools Launcher, click the 'Tools…' button and then select the 'Monticello' item.



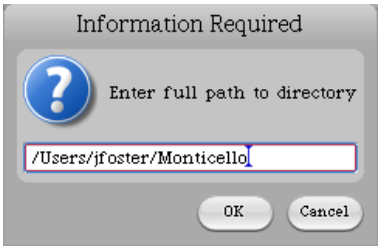
    b. This will open a Monticello Browser viewing GemStone code. Select '--all repositories--' in the left column and then click the 'add repository' button.



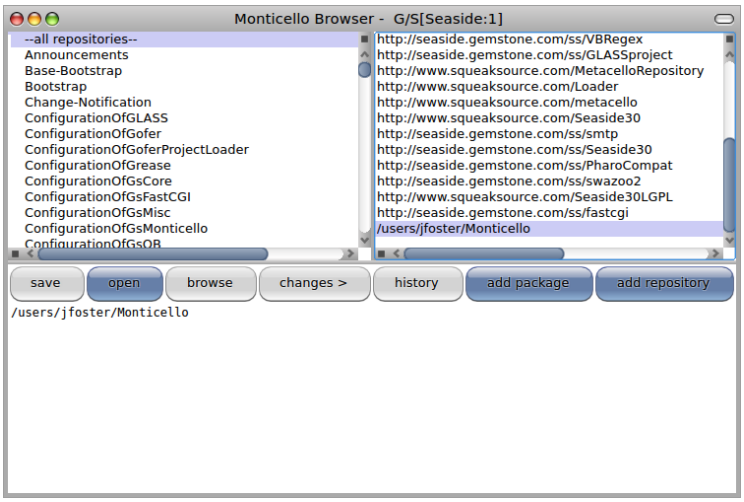
    c. This will open a dialog where you need to identify the repository type. Select the 'directory' option.



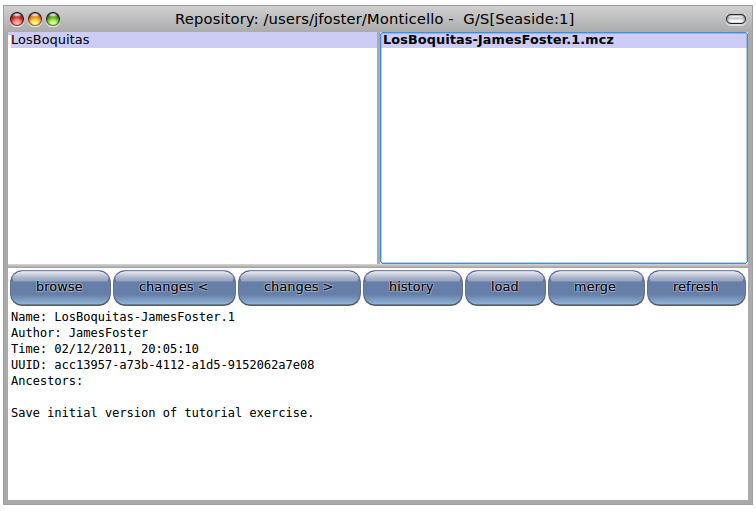
d. When asked to enter the path to the repository, enter the path you used in Chapter 14. You can use a file browser to look for a file named 'LosBoquitas-*.1.mcz'.



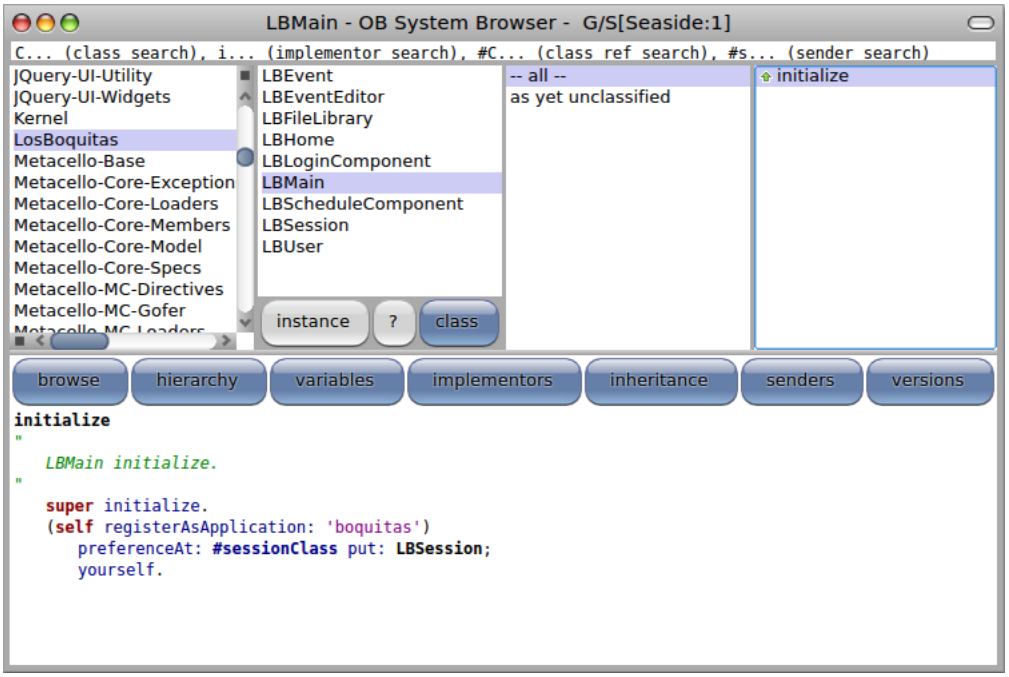
e. Once the repository has been added, scroll to the bottom of the repository list (the second column) and select your repository.



f. With the repository selected, click the 'open' button. This will open a Repository Browser. Select 'LosBoquitas' in the left column (a list of all the packages), and then click the first entry in the right column (a list of the package versions). Click the 'load' button and after the load note that the version name is underlined and no longer bold.

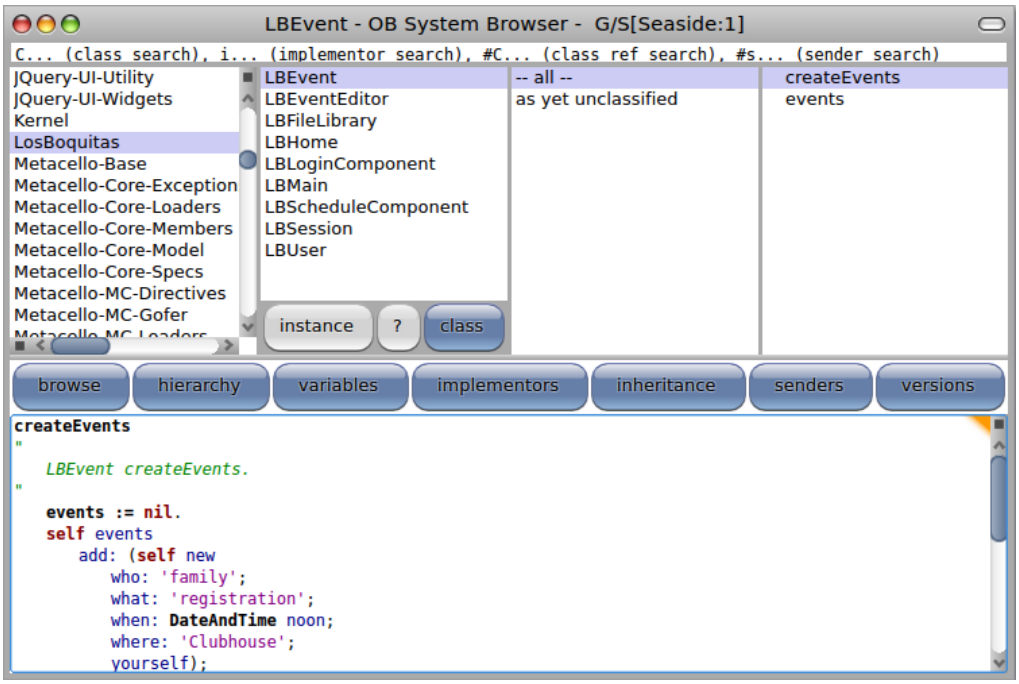


g. Close the Repository Browser and the Monticello Browser. Return to the System Browser and select the 'LosBoquitas' class category in the first column. This shows that the various classes and methods you defined in Squeak have been imported into GemStone. In fact, if you defined #'initialize' as a class-side method on LBMain, then the application will be registered when the class is first loaded.

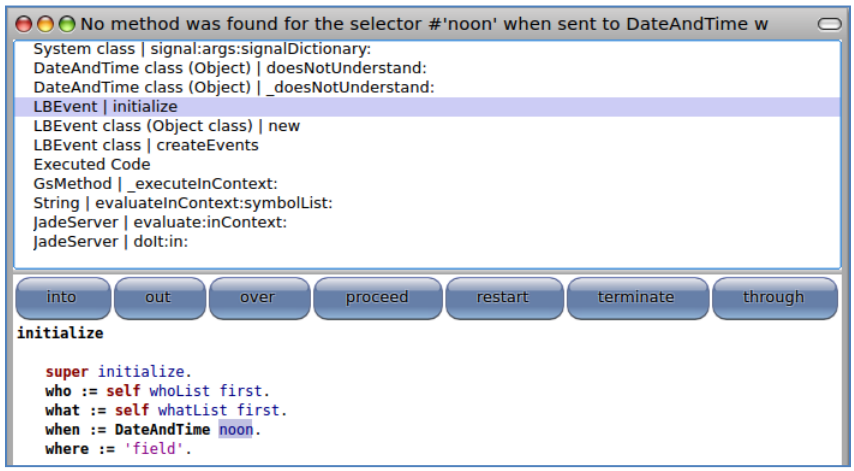


h. Return to your web browser, and navigate to http://localhost/browse and note that 'boquitas' is now available. You should be able to navigate the application and see the features.

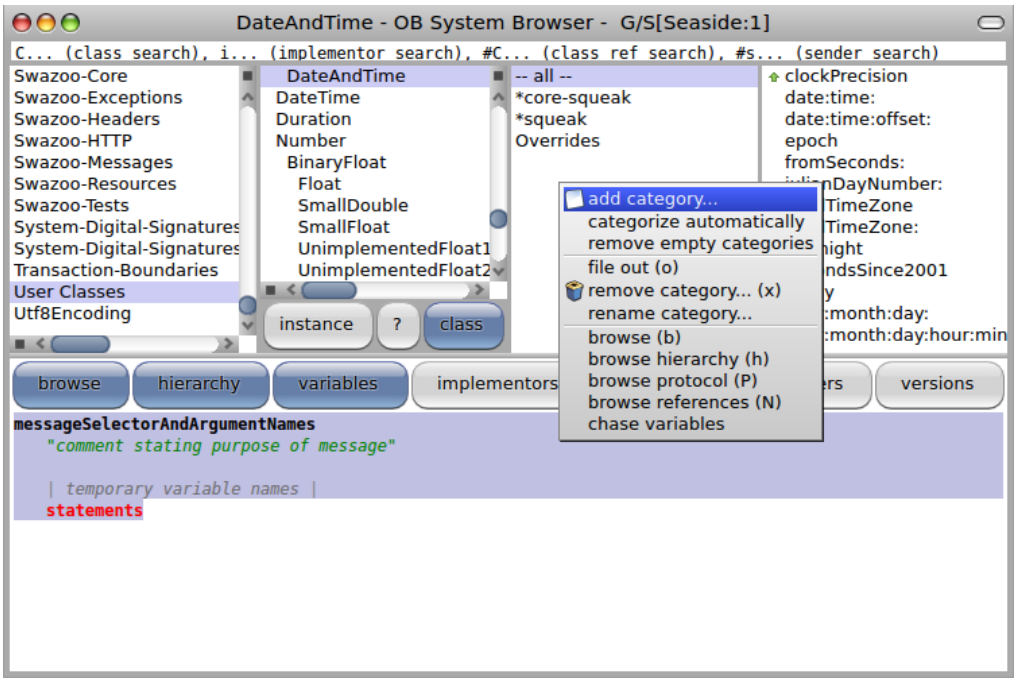
3. For the most part GemStone Smalltalk is compatible with Pharo, but sometimes we port an application and find that something doesn't work.
    a. Note that there are no events. Recall that we have a method to create events. In the System Browser, select the LBEvent class, switch to the class side, select the #'createEvents' method, click anywhere in the third line, and type <Ctrl>+<D> (for 'do-it') to evaluate the expression in the comment.



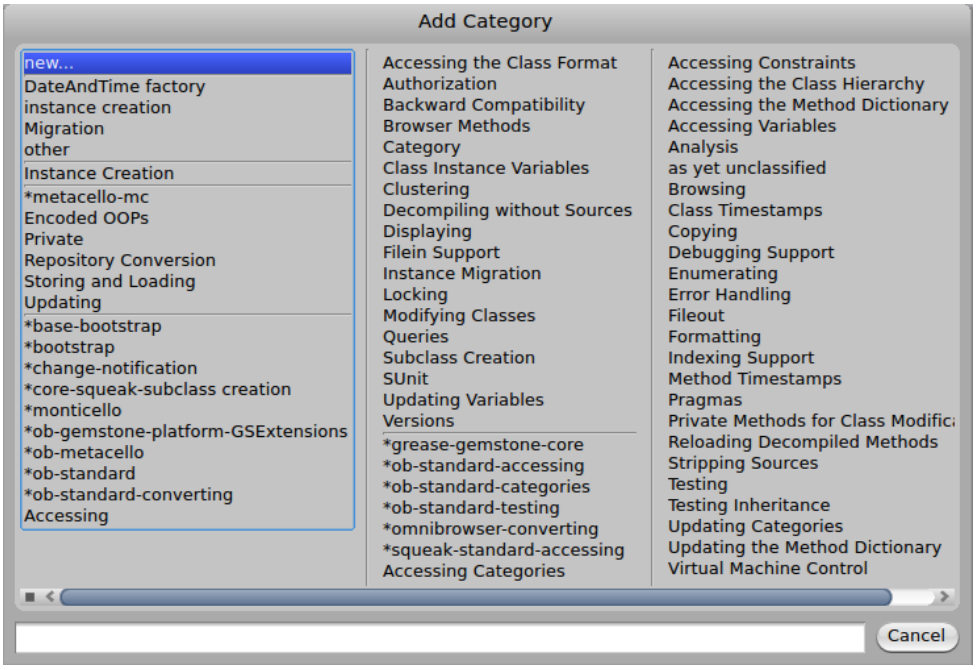
    b. In Pharo this expression generated three events. Unfortunately, in GemStone this generates a walkback reporting that the message #'noon' was not understood by the DateAndTime class. We could, of course, modify our application to not send that message. This would be a relatively simple and safe process. Instead, we will use this opportunity to examine the alternative of adding DateAndTime class>>#'noon' to GemStone. Close the walkback window.



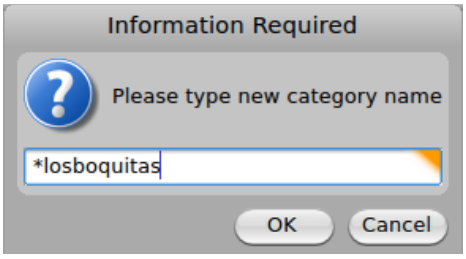
c. In Smalltalk it is customary for all Smalltalk source code to be included in the distribution image and any add-on packages. Furthermore, Smalltalk does not 'close' libraries in the way that, say, Java allows preventing any modification of source code. In the GemTools Launcher, click the 'Find' button, enter 'DateAndTime' in the text entry field, and select the 'DateAndTime' class. In the new System Browser, switch to the class side and right-click on the third column and select the 'add category…' menu.



d. In the 'Add Category' dialog, select the 'new…' menu item.

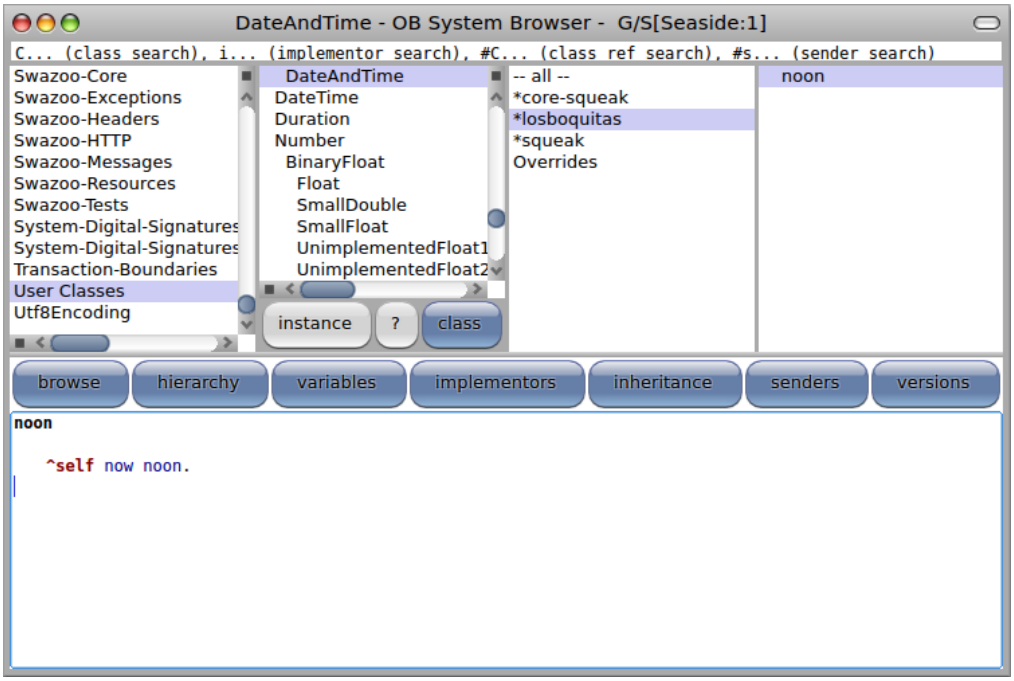


e. When asked for a new category name, enter '*losboquitas' and click the 'OK' button. Note that the category name starts with an asterisk, has no spaces, and is all lowercase. The '*' at the beginning means that methods in this category will not be packaged with the class for purposes of version control, but will be included with the package that has a name whose lower-case is 'losboquitas'.



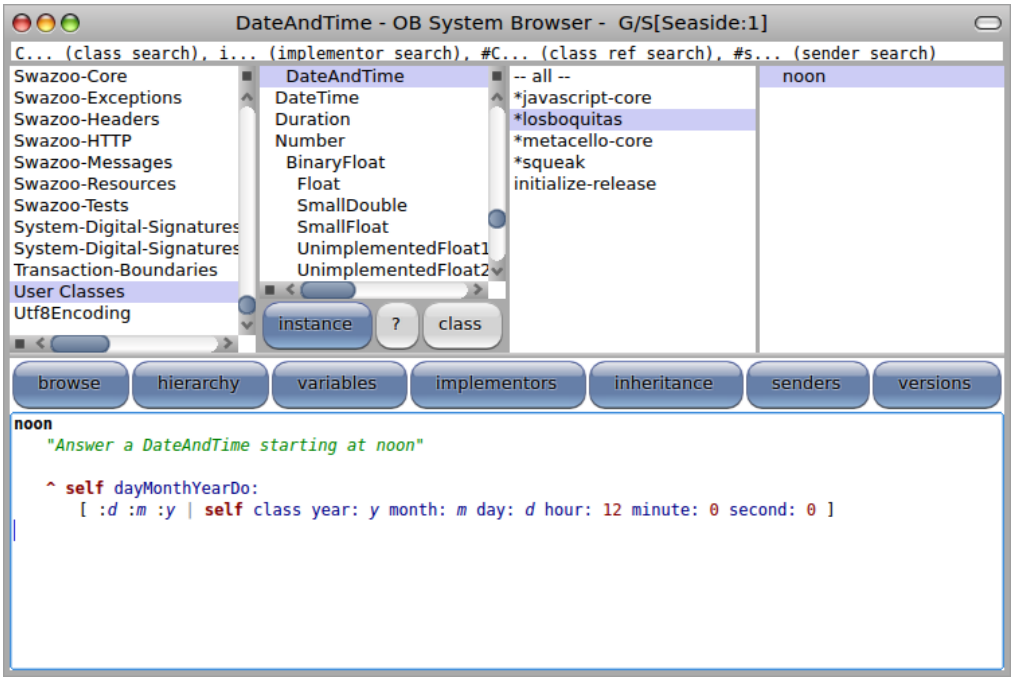
f. Once the category is defined, make sure it is selected and then enter a new method.

```
noon

   ^self now noon.
```



g. Now, switch from the class side to the instance side (click the 'instance' button), create a '*losboquitas' method category on the instance side, and enter a #'noon' method.

```
noon
   "Answer a DateAndTime starting at noon"

   ^ self dayMonthYearDo:
      [ :d :m :y | self class year: y month: m day: d hour: 12 minute: 0
second: 0 ]
```

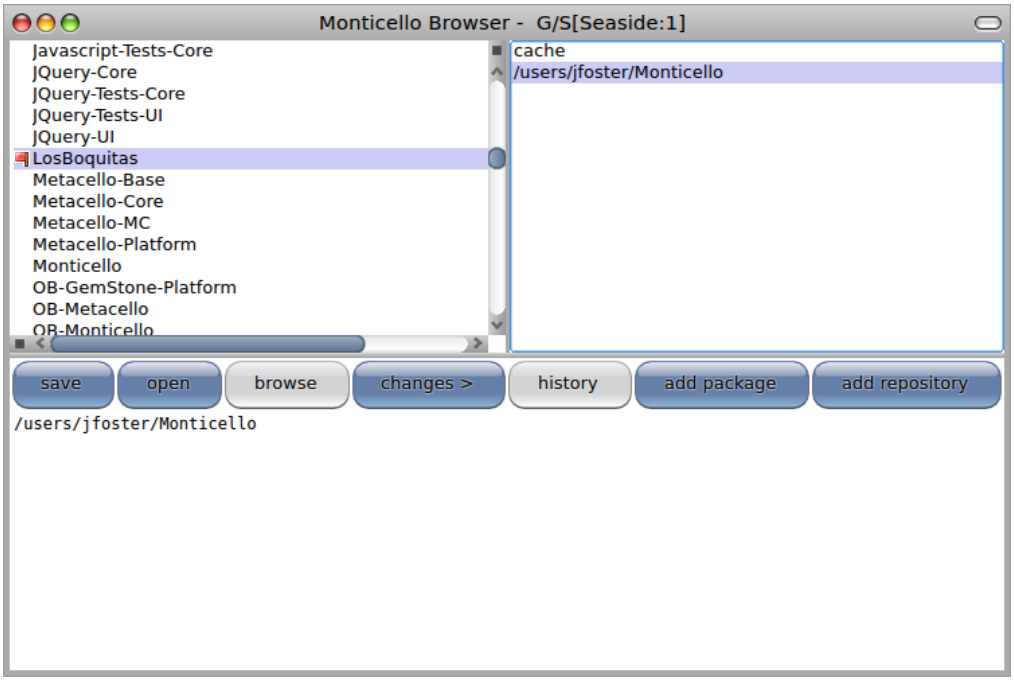


h. Now we will try again to create events. In a workspace, evaluate the following expression. This time we should not get an error.

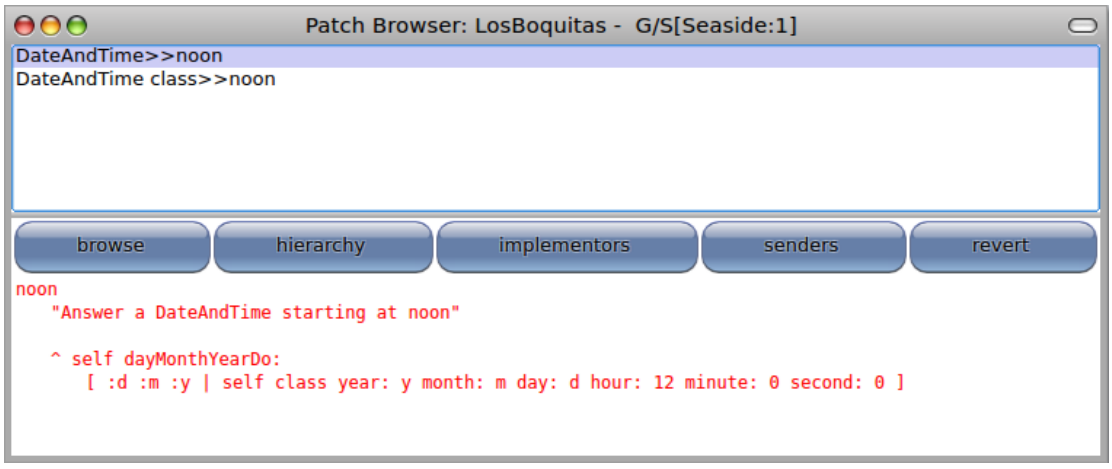
```
LBEvent createEvents.
```

i. Now return to your web browser and observe that the three default events have been created. The approach we have taken of modifying a base class is sometimes called 'Monkey Patching' (see http://en.wikipedia.org/wiki/Monkey_patch) and being able to do this in Smalltalk is considered a powerful feature. Our decision to put the extra methods in our 'LosBoquitas' package is a bit more questionable. A better place for this might be a package like 'Squeak-Kernel-Chronology' so that it can be shared by other packages and so that our LosBoquitas package can be loaded back into Pharo without unnecessary monkey-patching the Pharo methods. The reason we choose not to do so now is because the Squeak extensions for GemStone package is shared (see http://seaside.gemstone.com/ss/monticello) and any changes we made would be lost when we get a new version. (Also, if the main package were fixed, then we wouldn't have this error to use as an example of porting problems!)

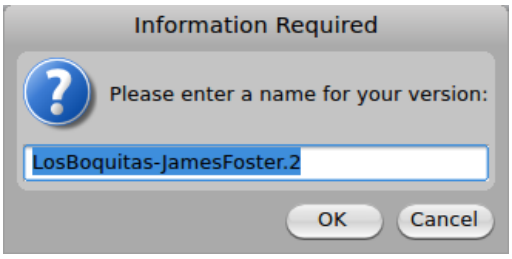
j. The changes we made are part of our GemStone object space, but we should save the code outside GemStone for purposes of source control. From the GemTools Launcher, click the 'Tools' button and select the 'Monticello' menu option. This will open a Monticello Browser. Scroll down till you see the 'LosBoquitas' line with a flag next to it. The flag means that the package has been modified.



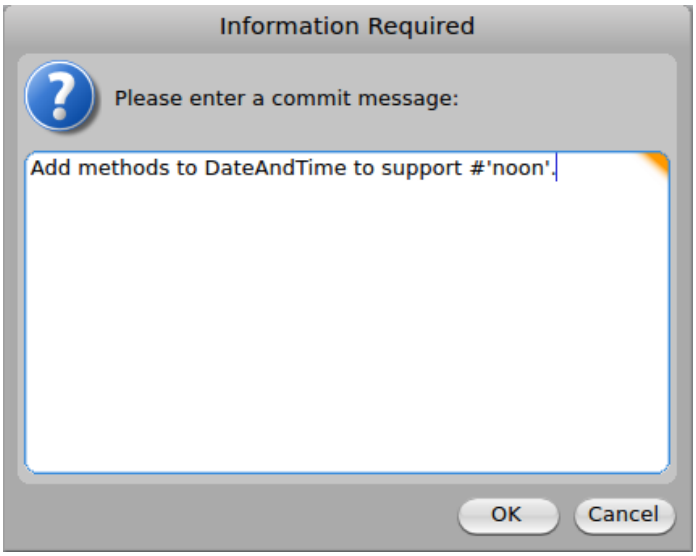
k. Select the local repository from the right-hand list and click the 'changes' button. This will give you a list of the changes in the local environment compared with the version in the repository. In our case, we have added two methods.



l. Close the Patch Browser and in the Monticello Browser click the ‘save’ button. In the dialog will be a default name for the new version. If you entered your initials when prompted earlier then the default name should be correct. Click the ‘OK’ button.



m. Before committing the modified package to the repository, Monticello will ask for a commit comment. Enter something meaningful.



4. From the GemTools Launcher click the ‘Logout’ button and select ‘Yes’ when asked to confirm. You can quit GemTools without saving. Finally, stop GemStone using the instructions from chapter 16.

With this Chapter we have demonstrated how to create an application in GemStone as well as how to load and modify an application saved in Monticello.

