Obtaining these Instructions and Software

1. Copy instructions and software
    a. Open a web browser on http:// 192.168.1.13/ to see these instructions.
    b. Copy from DVD, USB flash drive, or web.

GemStone/S 64 Bit Setup Options

2. **Native install** on 64-bit Linux or MacOSX 10.5. This is somewhat more complex, but does not require VMware. Follow these steps:
    a. Obtain a DVD or USB flash drive with the components.
    b. Make a directory /opt/gemstone.
    c. Copy GemTools, GemStone64*.zip, and installGemstone* to that directory.
    d. Run the installGemstone* script appropriate to your OS.
    e. Once the server is started, use 'startSeaside_Hyper 8000' to start a web server.
    f. In a web browser go to http://localhost:8000/seaside.
3. **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 (30-day trial 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). Follow these steps:
    a. Obtain a DVD or USB flash drive with the components
    b. Copy one of the VMware-guest64check* to a temp directory (if you are using Microsoft Windows, select the one ending with .exe; if you are using Linux, select the other one).
    c. Run the guest64check executable and verify that your hardware supports virtualization. If it does not, select another option.
    d. Reboot your machine into the BIOS setup utility. Ensure that hardware virtualization has been enabled (it is generally disabled by default).
    e. Copy GemTools.zip and GLASS-Appliance-1.0beta10.zip to a temp directory. Unzip the Appliance into your Virtual Machines directory and open it from VMware.
    f. Once started, you should see a web page with the guest's IP address. You can use the tools inside the guest or you can use the tools from your host system.
4. **Client-only connection**. This approach should work for anyone that can connect to a network and run Squeak. The disadvantage is that everyone using this approach will be using the same server on a single laptop.
    a. Obtain a user ID, password, and port number from the instructor.
    b. Copy GemTools from a DVD, from a USB flash drive, or from this link.

Introduction to Seaside and the Tools

The goal of this exercise is to help you become familiar with the tools and with Seaside.

a. If you are using a native or VMware server, start the server and open a web browser on seaside.
b. If you are using the shared server, open a web browser on http://192.168.1.13/ and download GemTools. Select your assigned name, enter your password, select the "Start Server" radio button, click "Submit," and click on the "Seaside" link.

5. Explore the 'Counter Application', the most famous demo of Seaside.
    a. Select examples / counter, and try clicking the ++ link and the -- link.
    b. Right-click on the ++ link and select the menu item to open the link in a new tab or window.
    c. Click the ++ link a couple times in the second tab then return to the first tab.
    d. Guess what will be displayed when you click on the ++ link, and then try it.
6. Explore the options made available with the Halos.
    a. Return to the second tab and click the 'Toggle Halos' link at the bottom.
    b. Click the 'S' link in the top right to see the HTML source.
    c. Click the 'R' link to return to the rendered component.
7. Edit code using a web browser.
    a. Click on the notepad icon (the leftmost icon) to open a class browser.
    b. View the methods in the 'actions' category (click on 'actions' in the third column).
    c. View the code for the 'increase' method (click on 'increase' in the fourth column).
    d. Change the code to add two (2) instead of adding one (1).
    e. Click the [Accept] button at the bottom.
    f. Return to the first tab and click the ++ link to see the value increase by 2.

8. Edit code in a debugger.
    a. In the second tab edit the code to add 'self halt.' (without the quotes) before the count assignment and save the method.
    b. In the first tab click the ++ link to bring up walkback.
9. Note that by default Seaside shows the top five (5) frames on the stack.
    a. Click the <Full Stack> link and take a look at the stack.
    b. You could click the <Proceed> link to resume from the halt.
    c. Click the <Remote Debug> link to save this continuation in the database for future debugging.
10. Open a debugger on this error:
    a. Launch GemTools
    b. Login to the database.
        i. In the Gem groupbox, edit the text entry field to change 'glass' to the server running GemStone/S.
            1. If you are running GemTools from within the VMware Virtual Appliance, then you can leave it as 'glass'.
            2. If you are running GemTools outside the VMWare Virtual Appliance, then you can get the IP address from the browser inside the appliance.

3. If you are running the server with a native install, then you can use 'localhost'.
4. If you are using the shared server, then use '192.168.1.13'.

ii. Change 'gs64ldi' to '50377' (or edit your services file to add this port).
iii. If you have a local install (either native or VMware), the user ID should be left as 'DataCurator' with a password of 'swordfish'.
iv. If you are using the shared server, then use the provided User ID and password.
v. Click the [Login] button.

c. From the Transcript window, open a debugger:
   i. Click the [Debug] button.
   ii. Select [Okay] when asked to abort the transaction. This gives GemTools access to the latest database view.
   iii. If you are presented with a pop-up menu of errors, select the bottom one.
   iv. Scroll down till you get to 'WACounter | increase' with the red flag and select it (this will be about three pages if you haven't changed the window).

11. Edit the code and proceed.
    a. Remove the 'self halt.' line and change the two (2) back to a one (1), then save your changes (using <Ctrl>+<S>).
    b. If the text area refreshes with the old text, click on a different line in the upper list and click back on the original line. This should show the new code.
    c. Click the [proceed] button in the debugger.
    d. Return to the first tab of your web browser and click the <resume> link.
    e. Note that the values now increase by one.

12. Try various other pieces of the system (using your history to go back to the Dispatcher Viewer or starting over).
13. Note that you get an error in the 'Mini Calendar' (tests / alltests).
    a. Click the <Remote Debug> link.
    b. Return to GemTools (in Squeak) and open a debugger.
    c. Select the last continuation with the label '^_MessageNotUnderstood 2010: No method was found for the selector <#'year'> when sent to <6>…"
    d. Select any stack frame in the list and click [inspect context]
    e. Examine various stack frames to get an idea of what Seaside does. Which frame has the error?
14. The problem appears to be that the object in the instance variable 'month' in WAMiniCalendar is an integer and does not understand the message #'year'. What classes implement the method #'year'?
    a. From the Transcript, click [Find Method…]
    b. Enter 'year' (without the quotes) and click Accept
    c. Select the first item in the list, 'year'.

15. While any of these classes could be what is needed, it is also possible that a subclass would do. What subclasses exist for Timespan?
    a. From the Transcript, click [Browse…]
    b. Enter 'Timespan' and click Accept
    c. Note that the class 'Month' is a subclass of Timespan, so should implement #'year'.
16. Where is the instance variable 'month' in WAMiniCalendar referenced?
    a. From the Transcript, click [Browse…]
    b. Enter 'calendar' and click Accept
    c. Select 'WAMiniCalendar'
    d. In the second list, right click on 'WAMiniCalendar' and select 'chase variables'. (If this does not pop up a menu try the <ESC> key.)
    e. In the new window, click on 'month' in the first list.
17. How does Date>>#month differ in Squeak from GemStone?
    a. In a Squeak workspace, inspect (Date today month) using <Ctrl>+<I>
    b. Do the same in a GemStone workspace (opened from the Transcript)

    It appears that in Squeak the method returns an instance of the class Month, while in GemStone the method returns an instance of the class SmallInteger.
18. In the "Chasing Browser" (from #16 above), examine each of the five methods in WAMiniCalendar. Note that when you click on a method in the second list, a third list is added (look for the horizontal scroll bar).
19. Which methods are reading the 'method' instance variable and which method(s) are setting the 'method' instance variable?
20. What methods are available to construct a Month object?
    a. Select the System Browser on Month (it should still exist)
    b. Click the [Class] button in the second column
    c. Select the #'month:year:' method in the fourth column
    d. Examine the source code for this method
21. Edit WAMiniCalendar>>#'initialize' so that the month assignment reads as follows (save with < Ctrl>+<S> or <right click>+<accept> or <Esc>+<accept>):

```
month := WAValueHolder with: (Month
        month: Date today month
        year: Date today year).
```

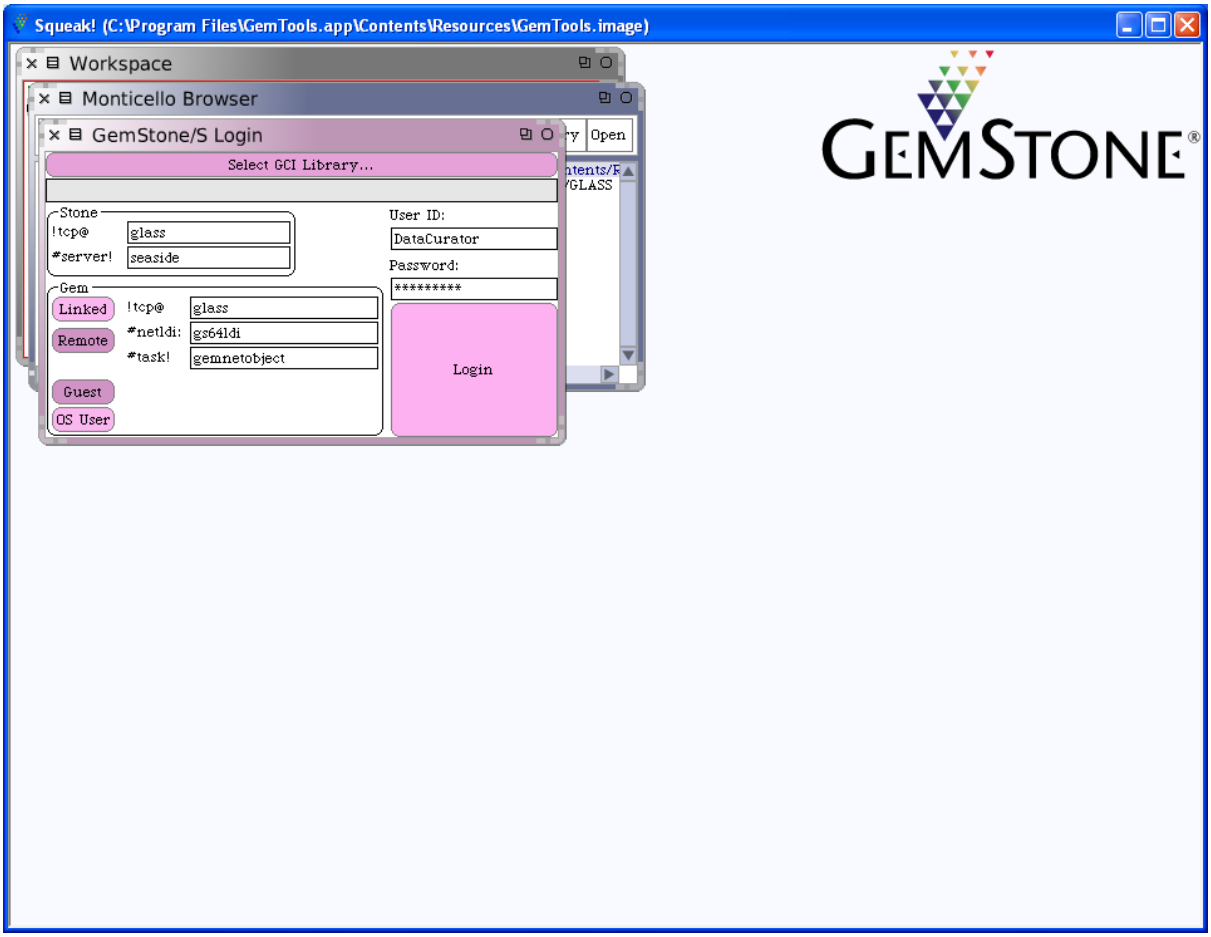
22. Test the changes.
    a. Return to your web browser, and start over with tests / alltests / Mini Calendar
    b. Return to Squeak and click the [Logout] button on the Transcript and note that the other windows close automatically.
23. Clear the Object Log
    a. From the first Dispatcher Viewer, select tools / objectLog
    b. Review some of the information stored in the Object Log
    c. Delete individual lines, delete down to particular lines, and then groups (e.g., fatal or error) till they are all gone.

This exercise has introduced you to running Seaside, editing code, and debugging an error. The Calendar error is a real-world example of one found when porting a Seaside application from Squeak to GemStone. The problem is that in these two dialects of Smalltalk the base class library has a

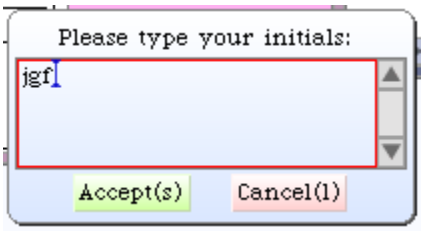
different implementation of Date. GemStone keeps the month as an integer, while Squeak has a more sophisticated Month class. While there are not too many such inconsistencies, there are a few and knowing what to watch for can make it easier to port a Seaside application from Squeak to GemStone.

A First Seaside Component

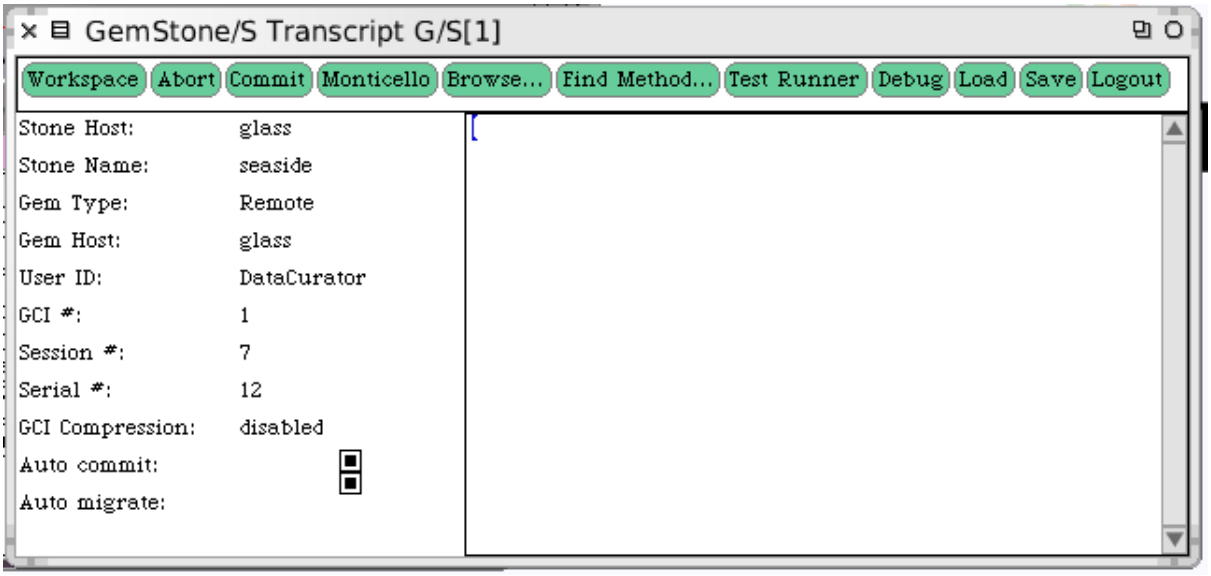
1. Locate and unzip GemTools.zip then launch the application. It should open a window that looks like the following (the title bar will look different on Mac or Linux, but the contents should be the same):



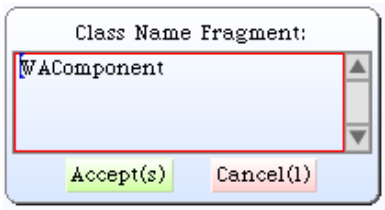
2. If you have not modified your hosts file you will need to replace the second 'glass' (the one in the Gem box) with the proper IP address. Change the User ID and Password to the ones provided by the instructor and the click Login. When asked to type your initials, do so and click Accept(s). The '(s)' means that you can type <Ctrl>+<S> instead of clicking on the button.



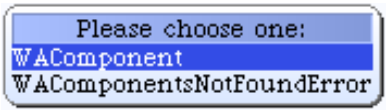
3. This should give you a Transcript window.



4. From the Transcript, open a Class Browser by clicking the 'Browse…' button. When prompted for a class name fragment, enter 'WAComponent' and click Accept(s).



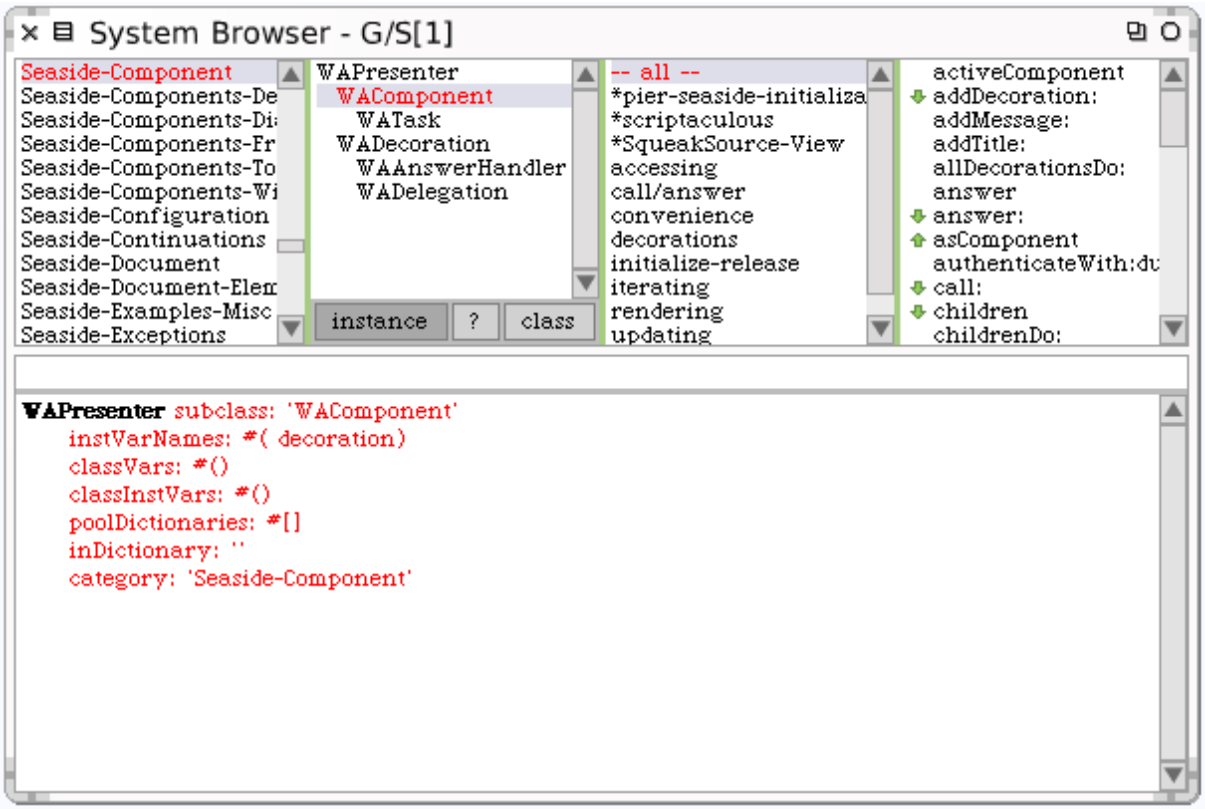
5. You will be prompted to select among two classes that include these letters in their name. Select the first item.



6. This should open a System Browser (see below) with four columns in the top section and a text edit area in the bottom section. The first two columns help select the class to show in the browser. The first column (the system categories) contains a list of names used to categorize classes; the category 'Seaside-Components' is selected. The second column (the class list) shows a hierarchy of the six classes in the Seaside-Components category, WAPresenter and five classes that inherit (directly or indirectly) from WAPresenter; WAComponent is selected. Two if the buttons below the class list specify whether we are looking at methods implemented "on the instance side" of WAComponent (the instance button is selected) or "on the class side" (if the class button were selected).

   The third and fourth columns help identify the method to show in the browser. The third column (the method categories) provides a grouping for methods and '--all--' is selected. (The method categories that begin with an asterisk indicate methods implemented in WAComponent that will be packaged elsewhere for purposes of source code control.) The fourth column shows a list of methods implemented in the class (either all of them or those with a specified method category). There are icons to the left of some of the method names to give additional information about certain methods. The green down arrow indicates that a particular method is overridden in a subclass (see #'children' for example). The green up arrow indicates that a particular method is also implemented in a superclass (see #'asComponent' for an example).

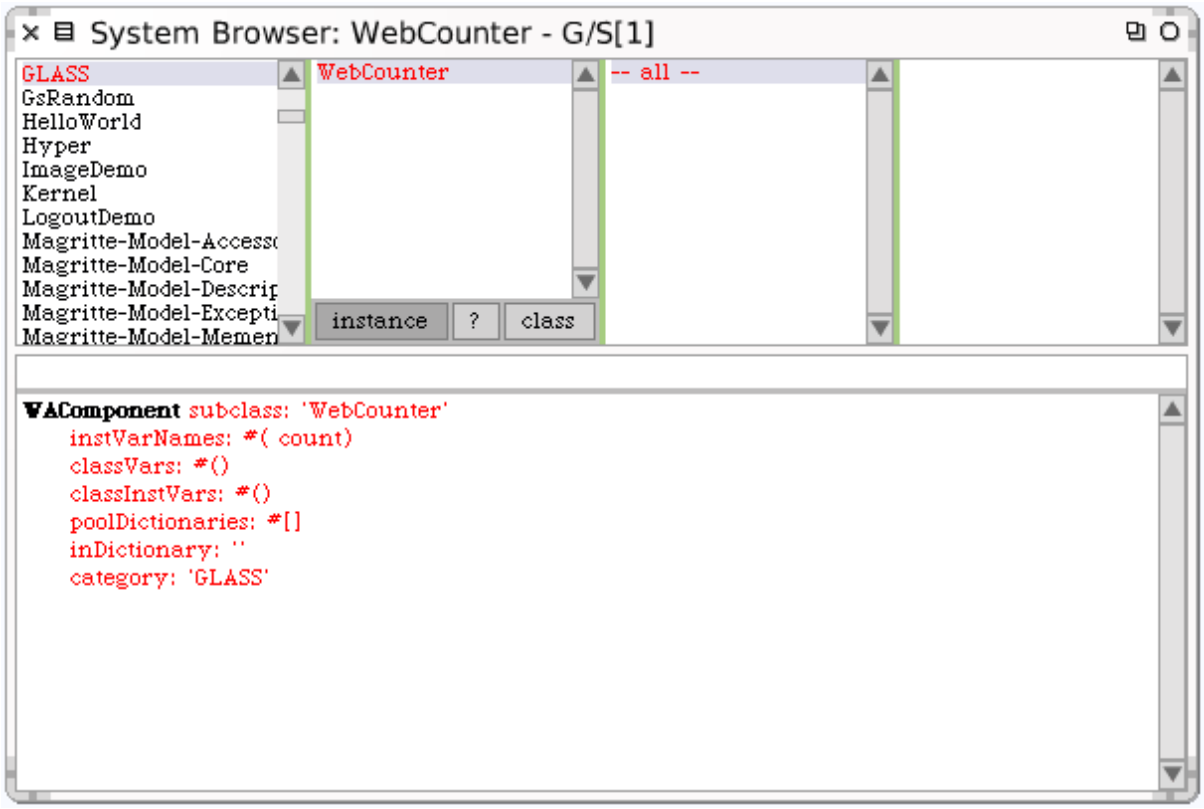
   Take a few moments to explore the System browser.



7. Our first Seaside component will be a counter. In the text area of the Class Browser, replace the current class definition with the following and then save the text (<Ctrl>+<S>):

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

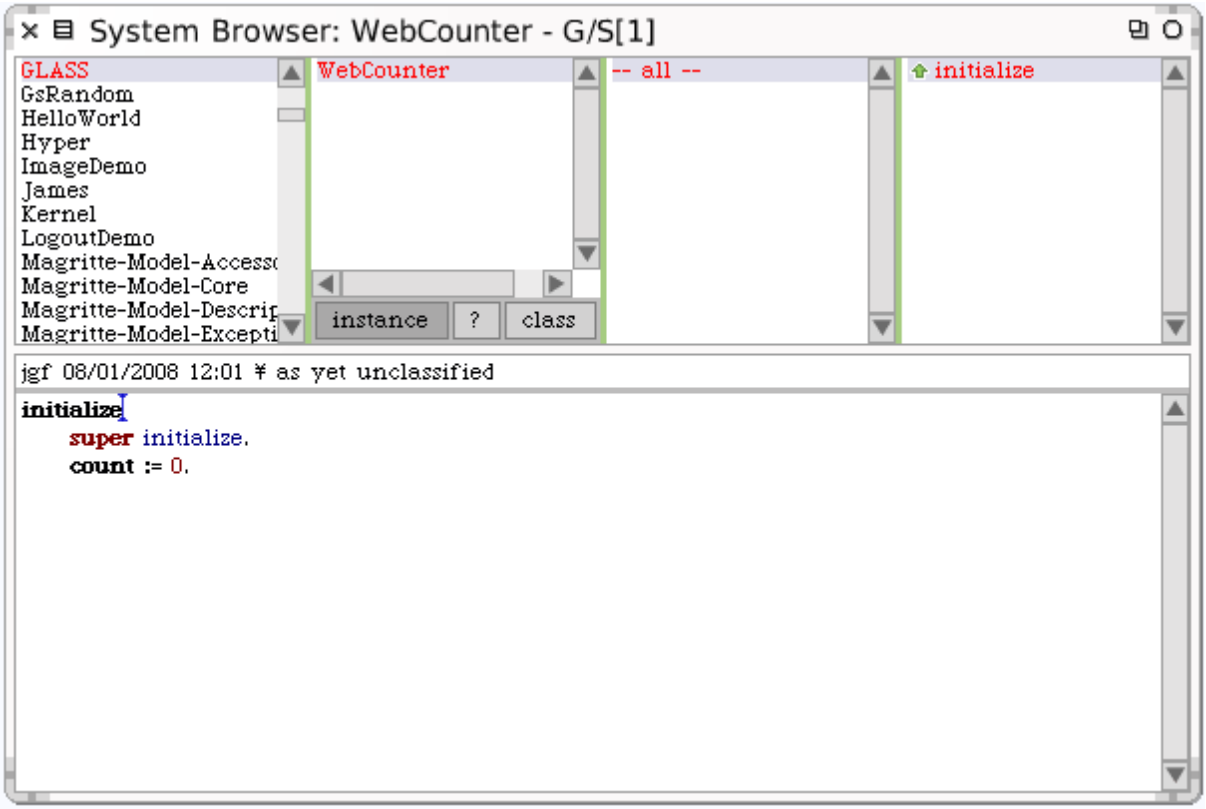
The result should look like the following:



8. Add an #'initialize' method to the instance side of WebCounter. Click on the method category '---all---' and then replace the text entry field with the following and save it:

```
initialize
   super initialize.
   count := 0.
```

The result should look like the following:



9. Create an #'increase' method:

```
increase
   count := count + 1.
```

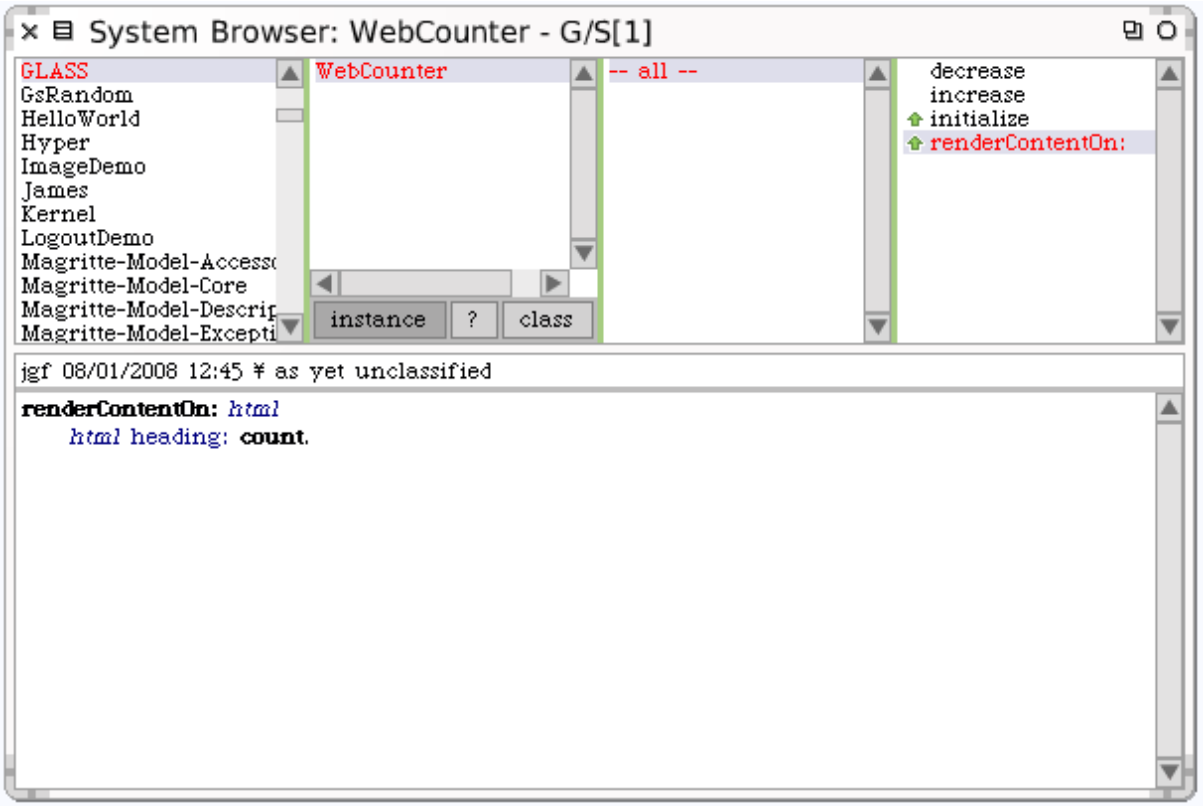
10. Create a #'decrease' method:

```
decrease
   count := count - 1.
```

11. Now we define the method #'renderContentOn:' to display the counter as heading. Seaside will call such a method when needed and display it in the web browser. In the following method we just say that we want to display the value of the variable count using a heading HTML tag.

```
renderContentOn: html
   html heading: count.
```

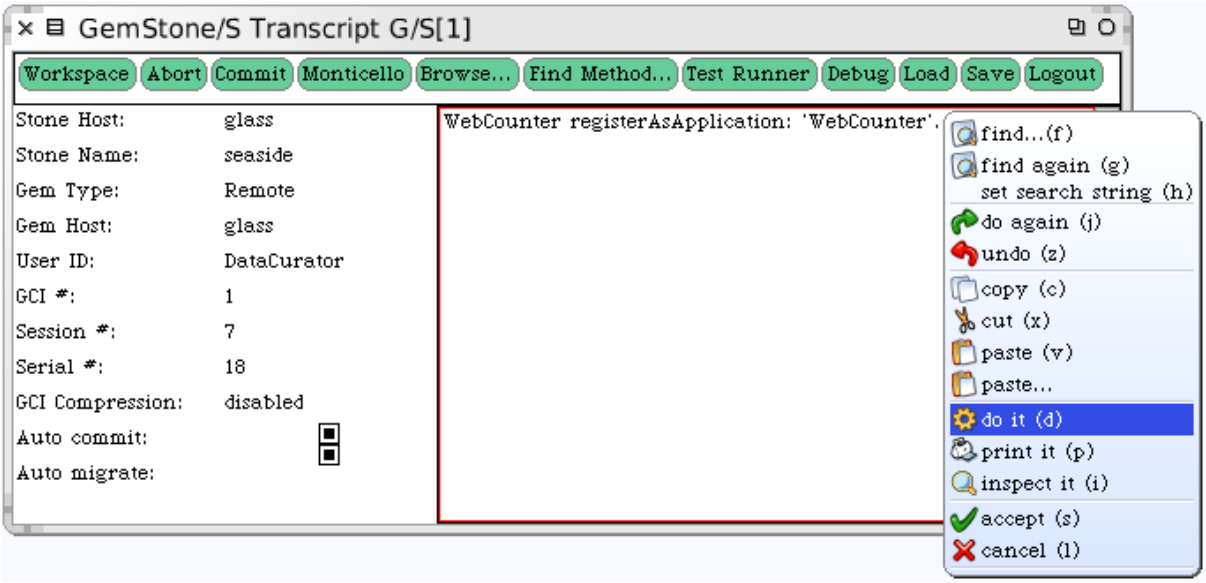
The result should look like the following (note that there are four names in the method list):



12. Now we should register a component as an application so that we can access it directly from the url path that will be associated with it. To register a component as an application, we send the message registerAsApplication: to the class we created and specify a path string that will be used to access the component from the web browser. The following code snippet registers the component WebCounter as the application named 'WebCounter' and can be executed in the Transcript.

```
WebCounter registerAsApplication: 'WebCounter'.
```

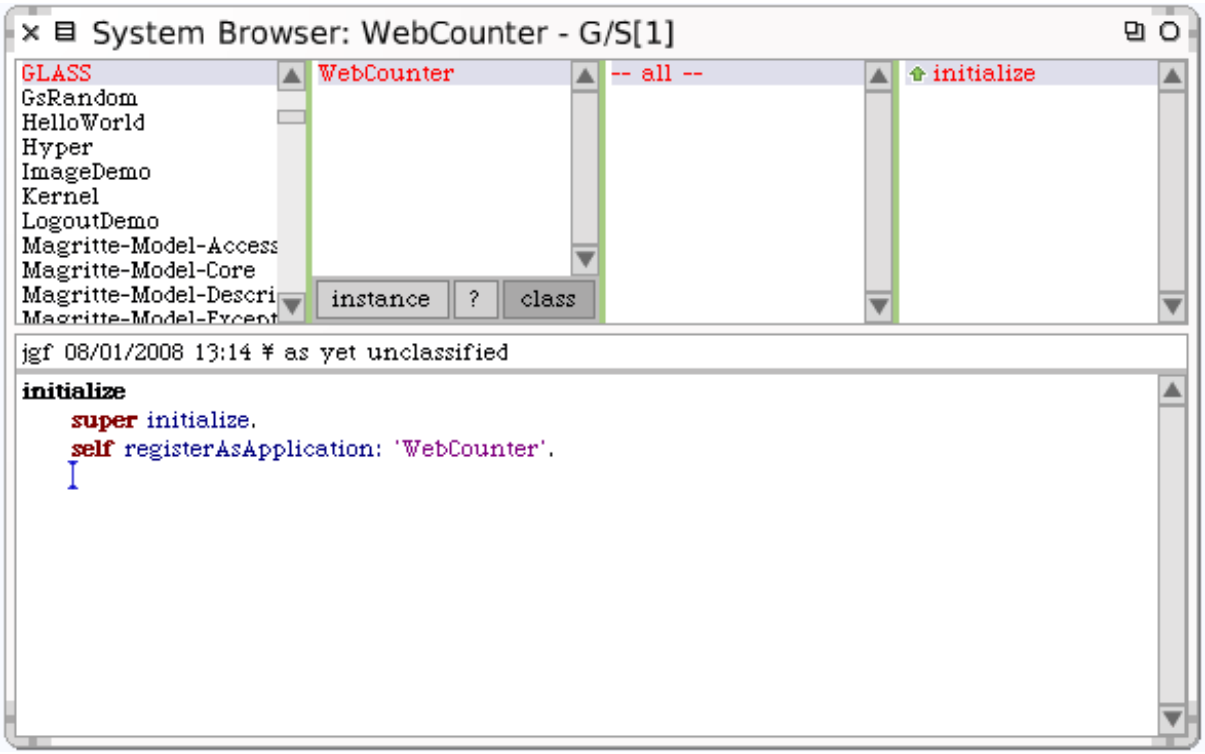
The following shows the transcript with the cursor at the end of the line. To get the pop-up menu you can press <Esc> or use the right-click on the mouse. Alternatively, you can type <Ctrl> + <D> to take the shortcut for 'do it'.



13. This expression can also be added in the class #'initialize' method which is invoked when the class is loaded into the system. In the Class Browser, switch to the class-side methods (click the [Class] button), select the method category '---all---', and then enter the following method:

```
initialize
   super initialize.
   self registerAsApplication: 'WebCounter'.
```

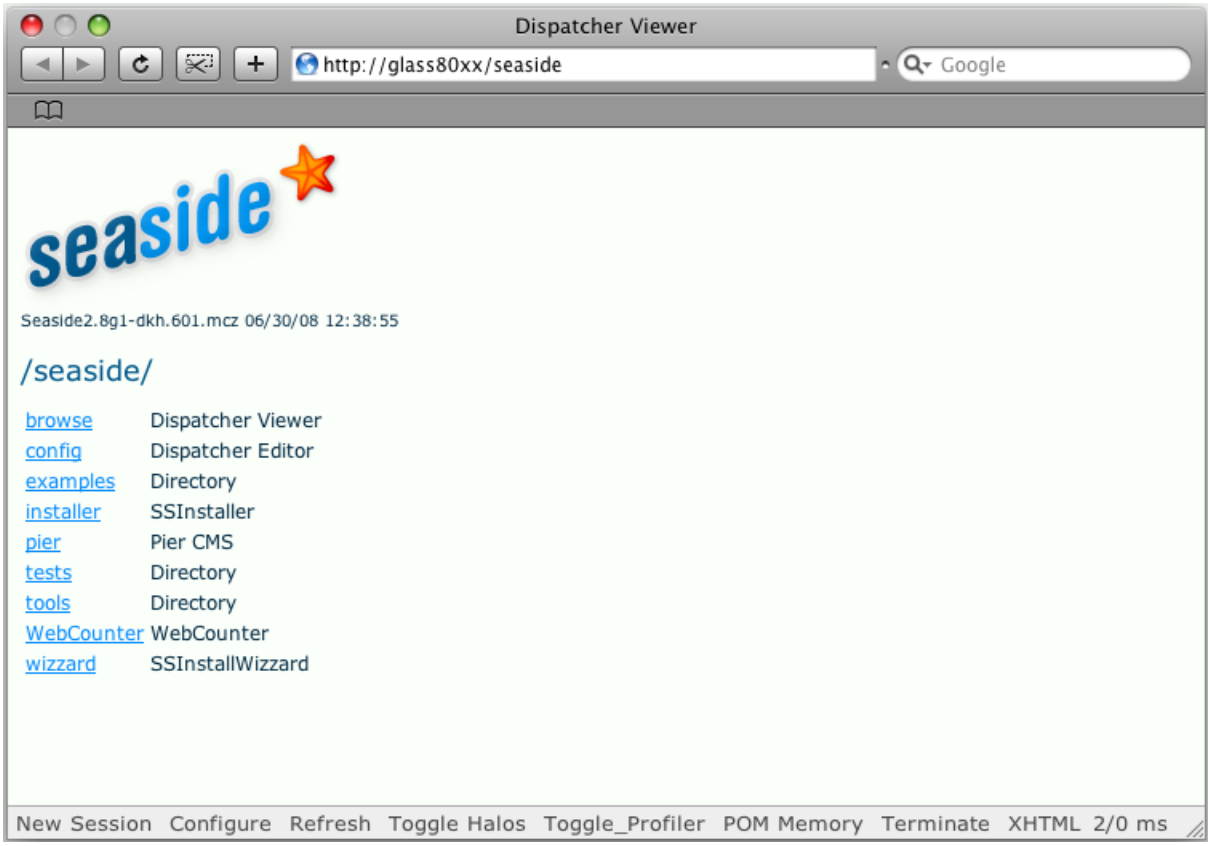
This should look like the following:



14. Now you can launch a web browser and look at the Seaside application list by going to the base Seaside dispatcher. For this class we have a web server for each participant, but they are each listening on different ports. You have been assigned a specific port that should be used in the web browser. In these instructions we have specified it as '80xx', but you need to substitute your assigned port. Following is a template of the URL (which you might want to bookmark to return to easily):

    http://glass:80xx/seaside

    This should bring up a browser like the following:

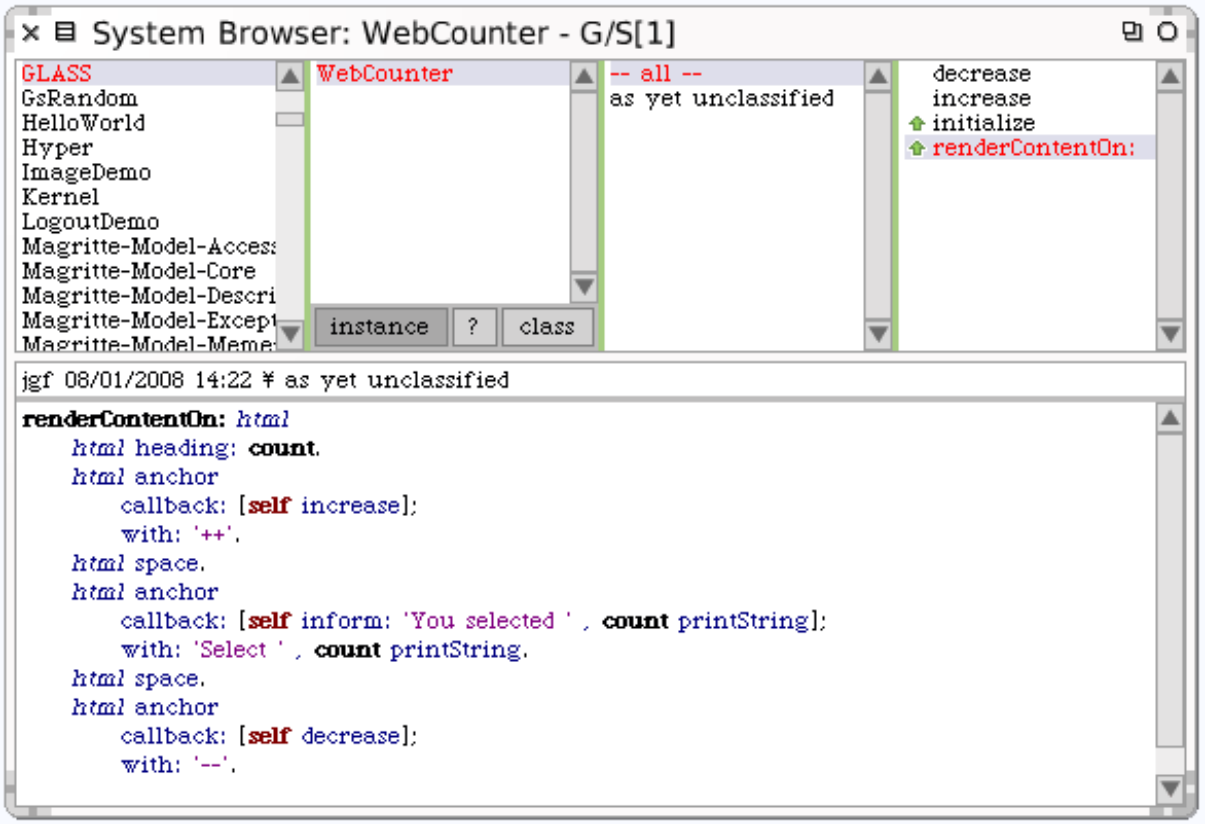


15. If WebCounter does not show up in this list, the application was probably not properly registered. Return to step #12. Click on the 'WebCounter' link and you should see the following:

16. Now we can add some actions by defining callbacks attached to anchors. A callback is a piece of code that will be executed when a link is clicked. Modify the instance method #'renderContentOn:' as follows:

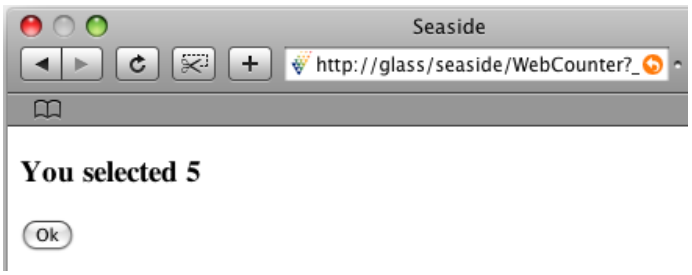
```
renderContentOn: html
   html heading: count.
   html anchor
      callback: [self increase];
      with: '++'.
   html space.
   html anchor
      callback: [self inform: 'You selected ' , count printString];
      with: 'Select ' , count printString.
   html space.
   html anchor
      callback: [self decrease];
      with: '--'.
```

This should look like the following (make sure you switch back to the instance side after step#13):

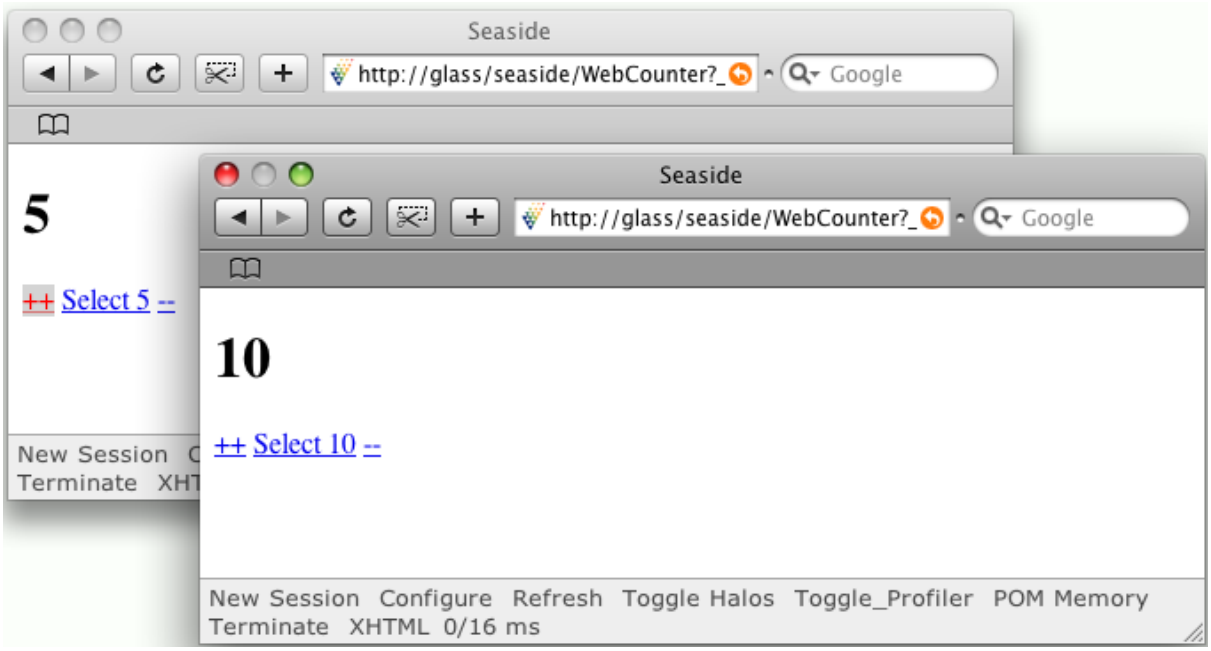


17. Refresh the page in your browser and click the ++ link till you have 5 displayed.

18. Click the 'Select 5' link and notice that you get a dialog telling you what you selected.



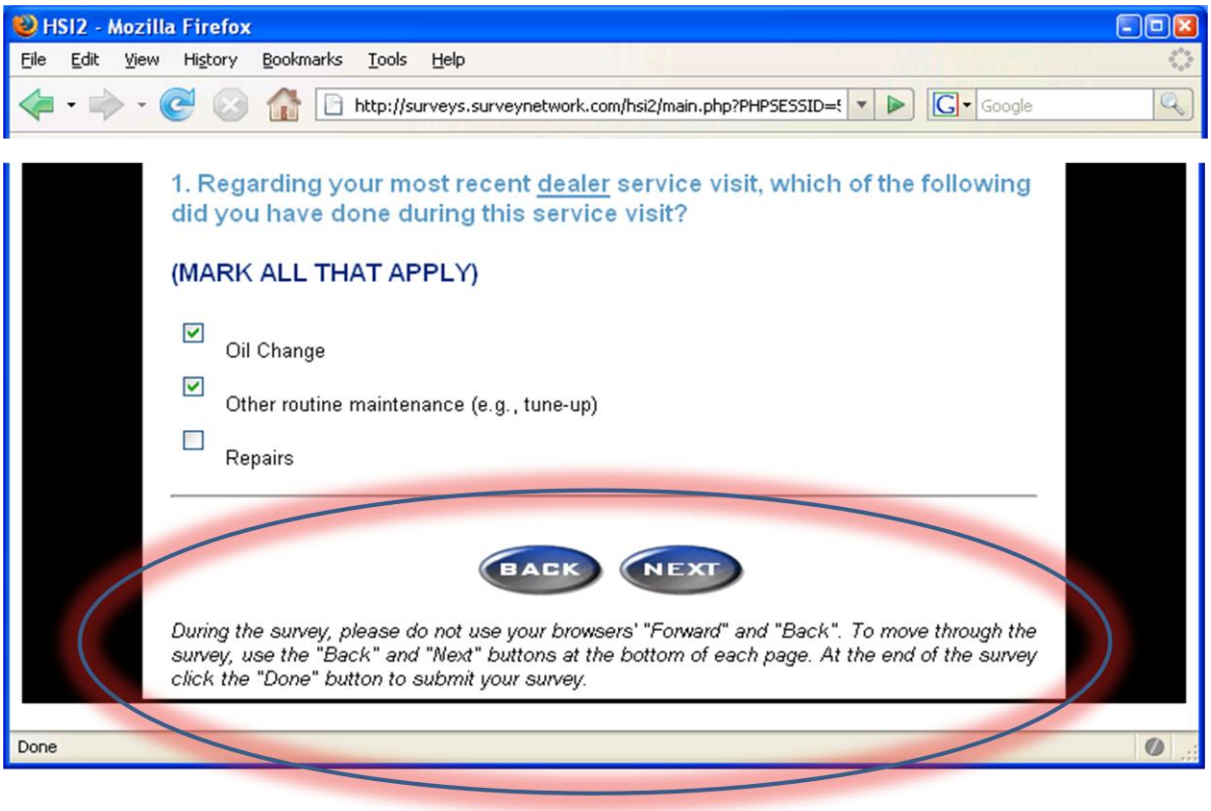
19. Click OK which should take you to a page with 5. Click the Back button till you get to 4. At that point, click on the 'Select 4' link. Note that it tells you that you selected 5!
20. Click OK which should take you to a page with 5. Click the Back button till you get to 4. At that point click the Refresh button. Note that it again tells you that you selected 5!
21. Click OK which should take you to a page with 5. Then try opening the ++ link in a new tab or window (by right-clicking on a link or the equivalent; do not start a new session from the beginning). Click the ++ link a few times in the second tab. This should give you something like the following:



22. Return to the first tab/window and guess what will happen when you click the 'Select 5' link. Click the link. Did it behave the way you expected?

    Here the application is saving the latest count on the server. This is typical of how most web frameworks handle session state on the server—there is one copy that reflects the most recent changes made on the server. Any interaction with the server will work with that latest value, not the one displayed on the browser!

    This trivial example is provided to demonstrate a larger problem that most web frameworks do not address. The user expects the data on the page to be the data on the server. Many otherwise sophisticated applications address this problem by instructing the user not to click the back, forward, refresh, or print buttons on the browser. Consider the following example from a survey commissioned by a car dealership:

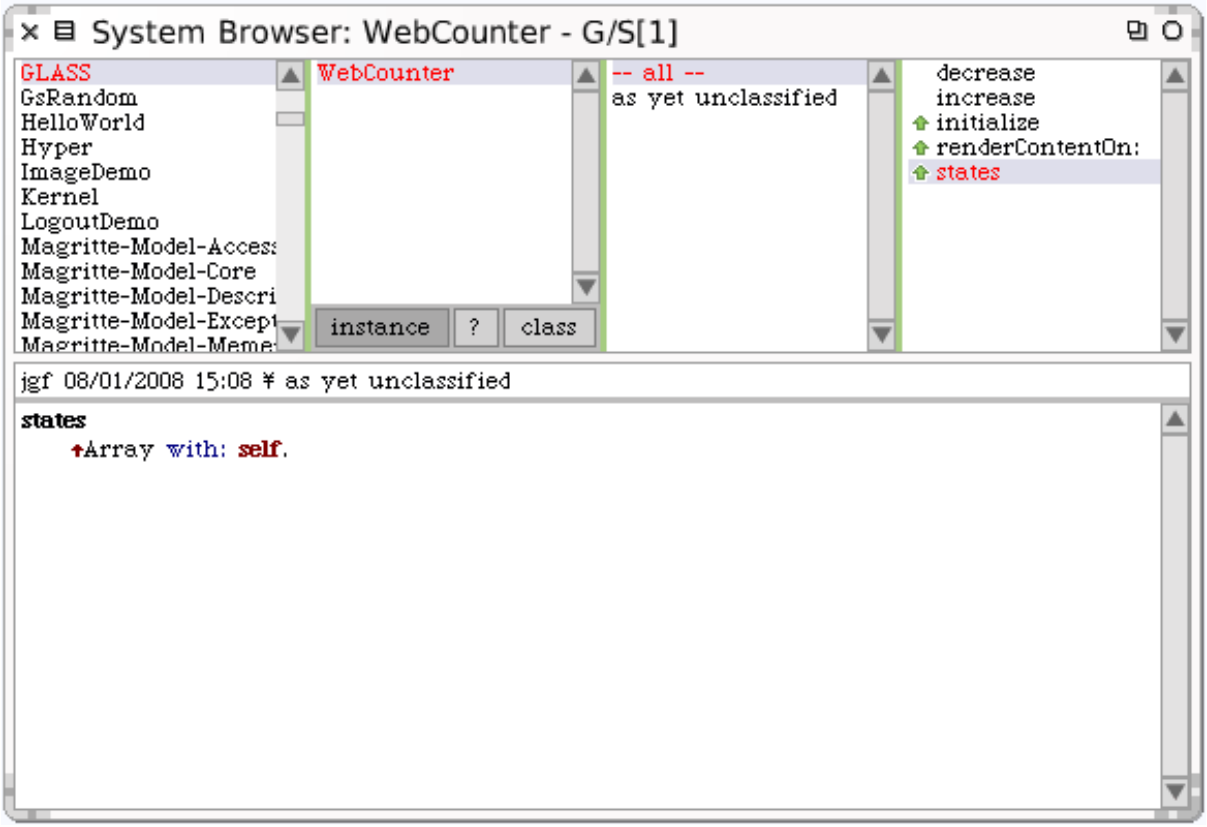


    Imagine you were using a web-based airline reservation system where you looked at flights for Monday then looked at flights for Tuesday. If you use the back button to go back to Monday's flight and click "Book this flight", which flight will be booked? Likewise, if you opened a second tab or window to look at flights for Tuesday then returned to the tab or window for Monday, which flight would you get? A web framework that remembers only the most recent information (as most will do) will exhibit this problem and Seaside offers a solution.

23. Seaside provides a way to tell the framework that a component is showing information that should be included in session state. You do this by adding a #'states' method on the instance side of your component, in this case WebCounter:

```
states
   ^Array with: self.
```

Your class browser should look like the following:



24. Restart the application in your web browser clicking the "New Session" link at the bottom. Try the exercise again, opening the ++ link in a new tab and see if the two web pages now share state. Try the back button, the forward, button, and the refresh button. Note that Seaside preserves the state associated with the page being displayed.

25. 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 re 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 a program flow that looks like it would if the program were handling a rich (or fat) client application.

    Not, 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 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 templating framework, processing starts at the beginning of the page with each request. There is not so easy a way of calling 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.

    We will demonstrate this with a link to the WebCounter application to ask for a new value.

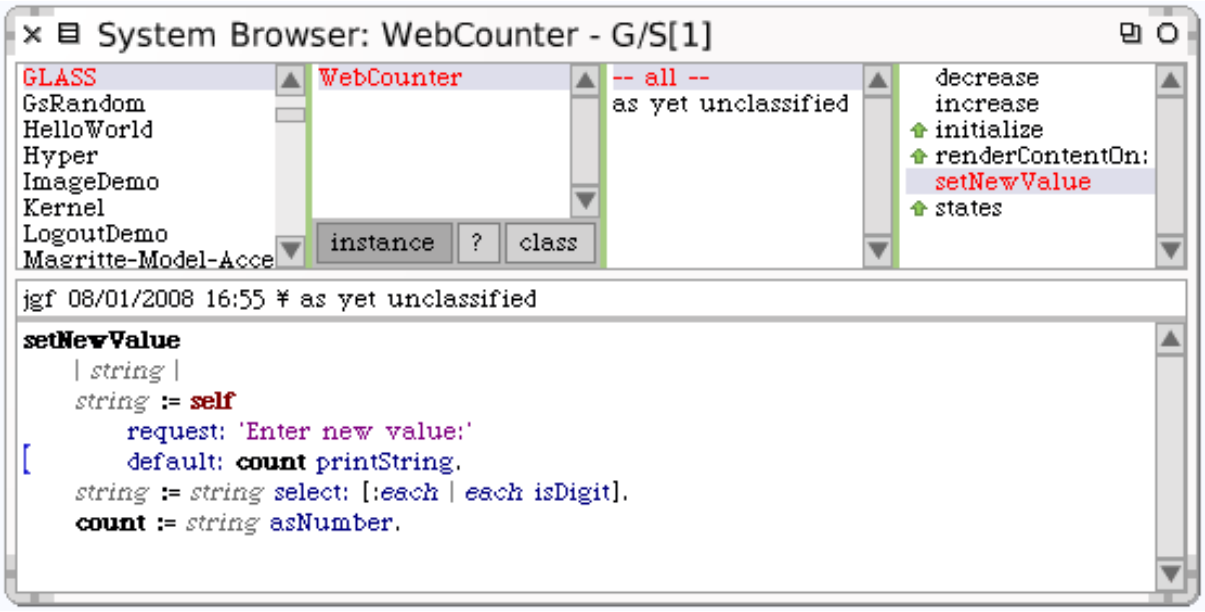
26. In the System Browser, edit the render method to add the last four lines as follows:

```
renderContentOn: html
   html heading: count.
   html anchor
      callback: [self increase];
      with: '++'.
   html space.
   html anchor
      callback: [self inform: 'You selected ' , count printString];
      with: 'Select ' , count printString.
   html space.
   html anchor
      callback: [self decrease];
      with: '--'.
   html space.
   html anchor
      callback: [self setNewValue];
      with: 'Set new value'.
```

27. Then add a new method to set the value:

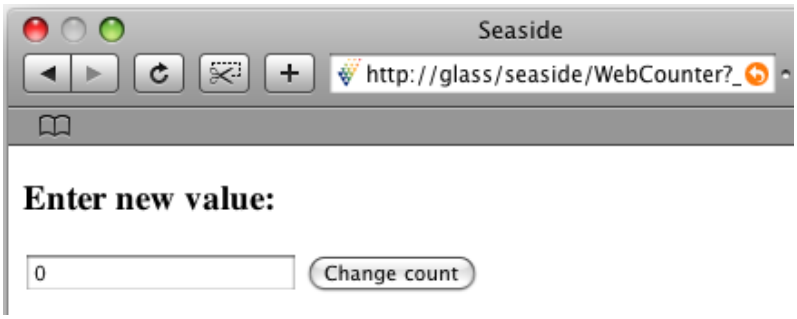
```
setNewValue

   | string |
   string := self
      request: 'Enter new value:'
      default: count printString.
   string := string select: [:each | each isDigit].
   count := string asNumber.
```

Your browser should look like this:



28. Now return to your web browser and refresh your page. There should be a link for "Set new value".

29. Click on the "Set new value" link and see that a new page is presented with a form for entering a value:



30. Enter a value (say, 42), and click the "Change count" button. Note that the value is changed. The thing to be noted here is how easy it was to treat a web page as a subroutine call and get back an answer (the string). While the particular example is trivial (and would be handled easily with JavaScript), it demonstrates the power of Seaside in allowing complete web pages to be handled as a subroutine by code. The code from the previous page is reproduced here:

```
setNewValue

   | string |
   string := self
      request: 'Enter new value:'
      default: count printString.
   string := string select: [:each | each isDigit].
   count := string asNumber.
```

31. This ends the prepared tutorial. Please feel free to stay and ask questions and experiment!