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:

Copyright © 2016 by GemTalk Systems LLC
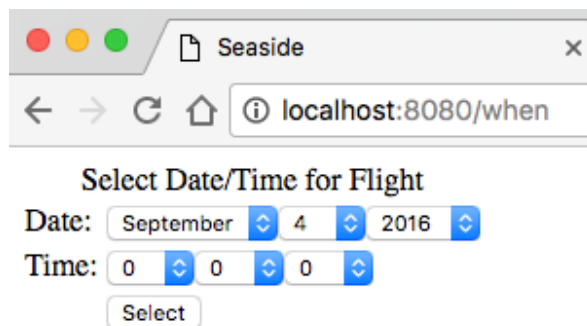
8. In the System Browser in Pharo, switch back to the FlightInfoComponent (we are done with the FlightInfoWhenComponent for the moment) and click in the third column to get a method creation template.
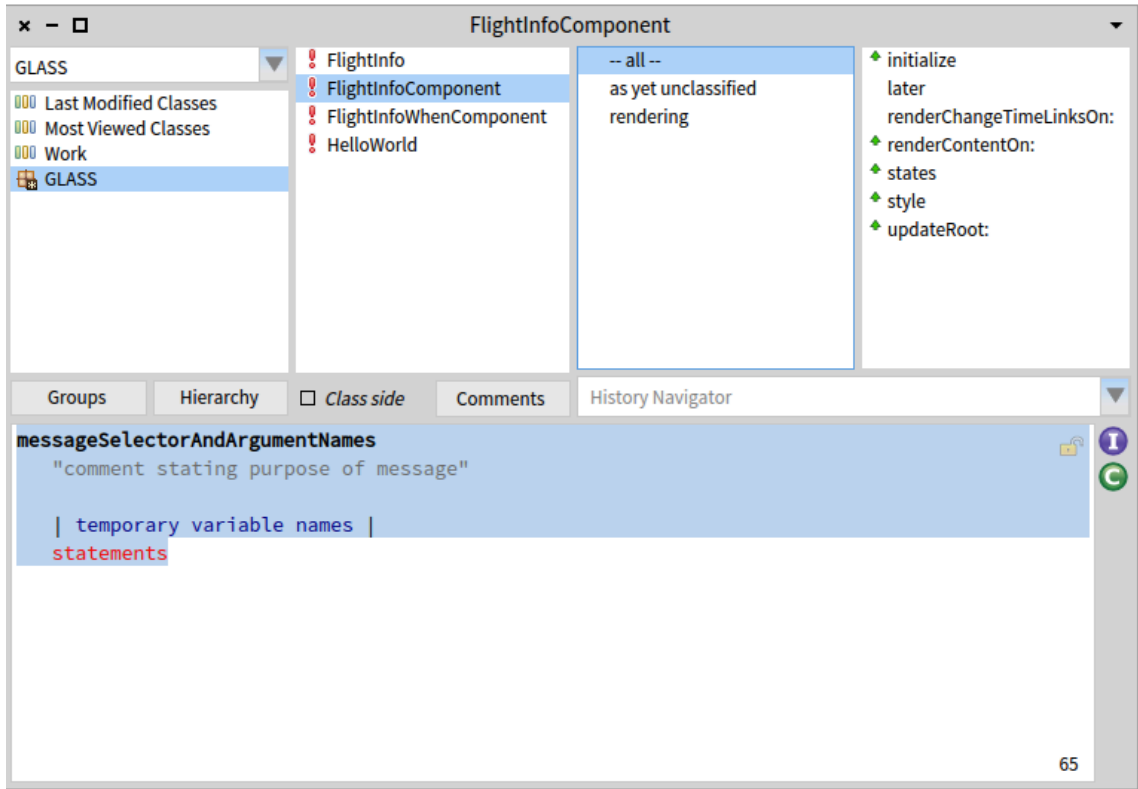
Now we will add a method to FlightInfoComponent to call the new component (if you get an error it is likely because you are still on the FlightInfoWhenComponent; make sure to select the FlightInfoComponent class):

```
        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.

Copyright © 2016 by GemTalk Systems LLC

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.