Seaside: A Flexible Environment for Building Dynamic Web Applications

Stéphane Ducasse, *Université de Savoie*
Adrian Lienhard and Lukas Renggli, *University of Bern, Switzerland*

The Seaside framework provides a uniform, pure object-oriented view for Web applications. Exploiting Smalltalk's reflective features, Seaside reintroduces procedure call abstraction in a client-server context.

Page-centric Web development structures an application into individual scripts, each responsible for processing a user request and generating a response. Links pass control from one script to the next. This imposes a go-to hardwiring of the control flow because each page must know what comes next. Although software developers have long-considered go-to statements harmful,[1] they are still present in today's mainstream frameworks, and they hamper the reuse of pages in different parts of the application. Another issue with Web application frameworks is the limited support they provide for composing multiple parts on the same page. The client-server relationship's stateless nature requires passing the current state back and forth between browser and server, inevitably leading to undesired coupling between these parts.

Recent Web application frameworks have tackled both of these problems. JWIG (Java Extensions for High-Level Web Service Development),[2] RIFE (Full-Stack Open Source Java Web Application Framework), Jakarta Struts, and JBoss SEAM (Web 2.0 Application Framework) solutions model control flow explicitly. However, most of these approaches don't integrate well with the rest of the code, because it's necessary to specify the flow in external XML page-flow specifications or to use a different programming language. Continuation-based approaches provide mechanisms to model control flow over several pages with one piece of code.[3–5] WebObjects, Ruby on Rails, Google Web Toolkit, and .NET, on the other hand, offer better abstractions by composing an application from components, but they fail to model control flow at a high level. With any solution, combining multiple simultaneous flows inside the same page is difficult.

In this article, after briefly discussing the key challenges of modern Web application development, we present Seaside (www.seaside.st), a highly dynamic framework for developing Web applications in Smalltalk. By exploiting Smalltalk's dynamic nature and reflective capabilities, Seaside offers a unique way to have multiple control flows active simultaneously on the same page. Furthermore, there's no need to recompile and restart Seaside application servers after each modification. Web developers can

 0740-7459/07/$25.00 © 2007 IEEE

debug and update applications on the fly, thus reducing development time considerably.

Web development challenges

To bring Web application development to the same level as desktop application development, a Web development environment or language should offer the following key features:

First, it should have *reusable and composable components*. A Web application logically contains reusable components—input forms, sortable reports, and so on. Building applications from reusable parts is natural. In addition, these components should define their own control flow. Composing new components out of several other components, each having a different flow, should be easy to achieve.

Second, it needs to *produce valid XHTML*. Producing valid XHTML code and connecting it to the application logic is difficult, especially if the XHTML code and application logic must be developed in different environments. Template-based XHTML generation lacks the host language's power and expressiveness. Seamless integration of development tools is often missing.

Third, it should *enable hot debugging*. Efficiently identifying bugs is a major challenge for Web developers. Web applications are becoming increasingly more complex, and the characteristics of client-server interaction make debugging more complicated, especially because advanced debugging support is often missing in today's mainstream approaches. The ability to hot-debug exceptions and use break points would speed up productivity considerably. Once a problem was fixed, the session could resume at the place where it had left off. There should be no need to restart and navigate to the problematic part of the application continually.

Fourth, it should support *hot recompilation*. On-the-fly method recompilation—that is, recompiling a method while the application is running—is important because it lets developers update the code without restarting the session. This is especially beneficial for Web development because it avoids time-consuming edit-compile-run cycles. Hot recompilation is a necessary requirement for hot-debugging support.

For a more in-depth presentation of these problems, please see our previous work.[6]

Smalltalk's reflective capabilities

Seaside introduces an abstraction layer over the asynchronous interaction protocol between the client and server to provide the illusion of developing a desktop application. Smalltalk's reflective capabilities make this high abstraction level possible. Smalltalk is a uniform language that applies simple principles systematically. In Smalltalk, everything is an object—an instance of a class. Objects exclusively communicate through message passing. In addition, Smalltalk supports *closures*, anonymous functions that can be passed around, stored in variables, or executed at a later time.

Smalltalk is written in itself and offers powerful reflective capabilities: structural as well as behavioral reflection.[7] We limit our brief overview to the reflective capabilities that enable creating a powerful Web development framework that offers multiple control flow composition, hot debugging, and hot recompilation. The reflective features of Smalltalk are comparable to those of the Common Lisp Object System (CLOS),[8] except that Smalltalk offers full access to the execution stack. These features include the following:

- *Shape and class changing*. Objects are instances of classes. When the class changes (that is, when instance variables are added, renamed, or removed), Smalltalk automatically migrates all instances of the class.
- *On-the-fly recompilation*. Developers can define and recompile methods on the fly. A changed method's currently active execution contexts run to completion using the old definition.
- *Stack reification*. In addition to `self` and `super`, Smalltalk offers a third pseudovariable, `thisContext`, which represents the execution stack on demand. For efficiency, Smalltalk reifies the stack (that is, makes it an object) only when the pseudovariable is used. This object is *causally connected* to the stack it represents. Therefore, not only does this object represent the stack, but any change made to the object reflects onto the execution stack itself. Hence, developers can manipulate the stack to change the program's execution.

Seaside's key features

Seaside is open source, and many applications have used it since the first version of it was released in 2002. Originally written by Avi Bryant (a consultant at the time, and now

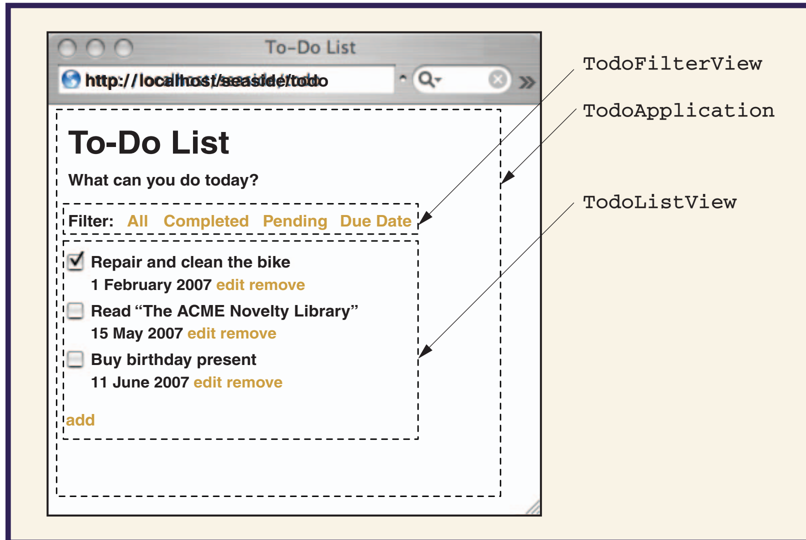


Figure 1. The to-do application, built from three different Seaside components.

owner of dabbledb.com), Seaside is continually under active development by a growing community of contributors, including the third author of this article. Here, we discuss the key features and characteristics of Seaside and how they address the Web development challenges that we identified earlier. Because of space limitations, we won't discuss the advanced AJAX (Asynchronous JavaScript and XML) support offered by Seaside (see http://scriptaculous.seasidehosting.st for details). In the remainder of this article, we use a simple application that lets users manage a to-do list of pending tasks. Figure 1 shows the main view, which contains three components. The main component is `TodoApplication`, which consists of the title and two subcomponents. One subcomponent, `TodoFilterView`, implements filters applied to the list of to-do items that the other subcomponent, `TodoListView`, applies. A to-do item consists of a title, a due date, and a completion status. Users can change an item's status by selecting its corresponding checkbox.

Components are the main building blocks in a Seaside application. A component is an instance of a user-defined subclass of `Component` and defines the look and behavior of a portion of a page. For each request, the session lets the components evaluate their *action callbacks* and render their current visual representation. In action callbacks, components can define control flow, for example, to temporarily pass control to another component. The following summarizes the three key features of Seaside and identifies the dynamic Smalltalk capabilities that make them possible:

- *Programmatic XHTML generation.* Following Smalltalk's principle of treating everything as an object, Seaside provides a very different approach to XHTML generation compared to template systems. Seaside generates XHTML through an object layer using pure message sending. Using block closures, Seaside binds action callbacks to components.
- *Multiple simultaneous control flows.* Each component can define its own control flow independently from the other components displayed on the same page. This makes it possible to implement business logic spanning multiple pages as one continuous piece of code. The enabling feature is the concept of *continuations*, which harness Smalltalk's ability to access and manipulate an execution stack.
- *Hot debugging, code-editing, and recompilation.* Seaside offers excellent development support; most notably, it makes the debugger work seamlessly with Web application code. Smalltalk enables this through hot recompilation and its treatment of exceptions as first-class objects, which can be inspected and resumed.

Domain-specific language for generating XHTML

Seaside doesn't keep XHTML code fragments in external files; there is no template system. Instead, a high-level interface relieves developers from checking correct tag nesting and attributes. The Smalltalk block closure syntax defines a domain-specific language (DSL) for XHTML rendering. Thus, through this dedicated language defined in Smalltalk, Seaside lets developers programmatically generate XHTML code.

Rendering. When the framework generates a response, each component visible on the page invokes its hook method, `renderContentOn:`, so that it can render a representation of itself on a `RendererCanvas` instance passed as a method argument by a convention named `html`. (In Smalltalk, developers can read attributes and local variables by using the name in an expression. They can write them by using the `:=` construct. In a first approximation, messages follow the `methodName1: arg1 name2: arg2` pattern `receiver`, which is equivalent to the Java syntax `receiver.`

methodName1Name2(arg1, arg2).)

Figure 2 shows a sample rendering of an implementation involving the `TodoApplication` component.

The `render` canvas returns instances of XHTML tags, called *brushes*. In the code given in figure 2, for example, the object returned from `html div` is a brush for the `div` tag. Brushes define the interface to specify attributes and the tag's contents. Seaside passes the contents (that is, nested tags or strings) as arguments to the `with:` message. The closures enforce the correct nesting of tags. It's not possible to forget a closing tag, because the compiler complains about the invalid source code. Strings, such as the `title` in figure 2, can be passed directly to the `render` canvas and are automatically encoded. Because the attributes are set through accessor methods, the framework ensures the validity of the tag declaration.

Action callbacks. So far, we have discussed only how a component renders itself. But components can also react to actions triggered by users via action callbacks. Seaside lets users employ closures to define action callbacks on anchors and buttons, as well as on form elements such as text input fields, check boxes, and select boxes. Execution of these closures is delayed until the user triggers the action. Buttons and anchors use closures without arguments. Other form fields use closures that expect an argument that is bound to the current value of the element upon activation. The following code snippet from the `TodoListView` class displays an item's checkbox and lets users check or uncheck a to-do item.

```
html checkbox
      submitOnClick;
      value: anItem done;
      callback: [:value | anItem
                   done: value].
html span: anItem title
```

The `html checkbox` statement returns a checkbox brush, which automatically submits the form when clicked. Furthermore, it sets the checked Boolean value, depending on the item's current status. Finally, the action callback, which executes when the user clicks the checkbox, expects one argument, a Boolean value, reflecting the item's new status.

These kinds of callbacks provide a high abstraction level over the low-level HTTP protocol. Application developers need not manually fetch submitted parameters from HTTP requests or validate and parse these strings with every request. Instead, Seaside automatically calls the appropriate handlers with real objects as parameters.

```
TodoApplication>>renderContentOn: html
      self renderTitleOn: html.
      ...

TodoApplication>>renderTitleOn: html
      html div class: 'title'; with: [
        html heading
          level: 1;
          with: self model title.
        html paragraph
          with: 'What can you do today?']
```

(a)

```
<div class="title">
      <h1>Todo List</h1>
      <p>What can you do today?</p>
</div>
```

(b)

Figure 2. Sample rendering involving the `TodoApplication` component in Seaside: (a) the implementation; (b) the generated XHTML.

Control flow

Whenever the user requests a new page by clicking on a link or a button, Seaside executes an action callback of a component. In this callback, the component can change the application mode's state, as we discussed in the previous section. Moreover, callbacks can serve to define a control flow while other components on the page remain unchanged. Every component can have its own control flow, which describes the sequence in which it is temporarily replaced by other components. Control flows in Seaside need not be linear; developers can mix control statements, loops, method calls, and domain code with messages to display components. All this is possible with plain source code; there is no need to build complex state machines.

Users can employ the `TodoFilterView>>due` method to set a due-date filter. The `TodoFilterView>>due` method displays two calendar components in sequence, letting users select start and end dates as the range to filter the items. This method first instantiates and calls a

```
TodoFilterView>>due
        | start end |
        start := self call: (Calendar new
          addMessage: 'Select Start Date').
        end := self call: (Calendar new
          canSelectBlock: [:date | date > start];
          addMessage: 'Select End Date').
        self filter: [:item | item due between:
               start and: end]
```

Figure 3. Using the `TodoFilterView>>due` method to set a due-date filter in Seaside.

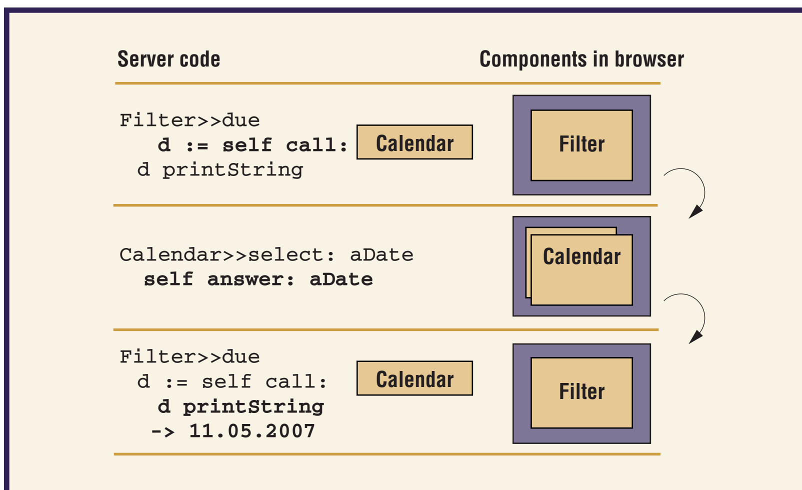


Figure 4. The basic elements of control flow in Seaside: the `call:` and `answer:` methods.

calendar component to let the user select a start date. When selected, this date is returned and stored in the `start` local variable. Because `start` is a date object, developers can employ it right away in the second calendar component to ensure that the end date is after the start date. Eventually, in the method's last line, a closure referencing the start and end dates defines a filter. This closure then serves to filter the to-do items, as shown in figure 3.

As figure 3 shows, there is no need to serialize states into strings and pass information from one page to another. Temporary variables serve to keep track of the state between the different steps of the flow. Control flows in Seaside are based on the interplay between the `call:` and `answer:` methods, as figure 4 shows. The framed `Calendar` in the `due` method is an instance of a Seaside component. Seaside invokes the `select:` method in `Calendar` from an action callback—that is, when the user confirms the date selection.

Call. One component can pass control to another component. Thus, the latter temporarily replaces the former. The developer achieves this by sending the `call:` message to the old component with the new component as the argument. In figure 4, sending the `call:` method with an instance of a calendar component installs this calendar atop the filter component and passes it control. Other components elsewhere on that page remain functional and can be used independently of the new component.

Answer. A component can return control to the component from which it was called by using the `answer:` method. When returning, a component can additionally return an object to the caller. In figure 4, the expression `self answer: aDate` makes the calendar component return a date object to the filter component.

Composing components

A user interface is built mostly from different parts visible on the same screen as our example illustrates. Seaside implements these parts as components that can be composed from other components. The following code describes how the to-do application's top-level component plugs together the filter and the to-do list. Seaside stores the two components in instance variables of the `TodoApplication` component instance and places them inside XHTML `div` elements to be rendered one after another. The title displays above the two components.

```
TodoApplication>>
      renderContentOn: html
   self renderTitleOn: html.
   html div
      class: 'filter';
      with: filterView.
   html div
      class: 'list';
      with: listView
```

Earlier, we illustrated a control flow of the filter component, and the selection of start and end dates. The list view component, however, also defines several control flows—for example, a confirmation dialog box displayed when the user is about to delete a to-do item, and dialog boxes for adding or editing items. Without additional changes to the code, both

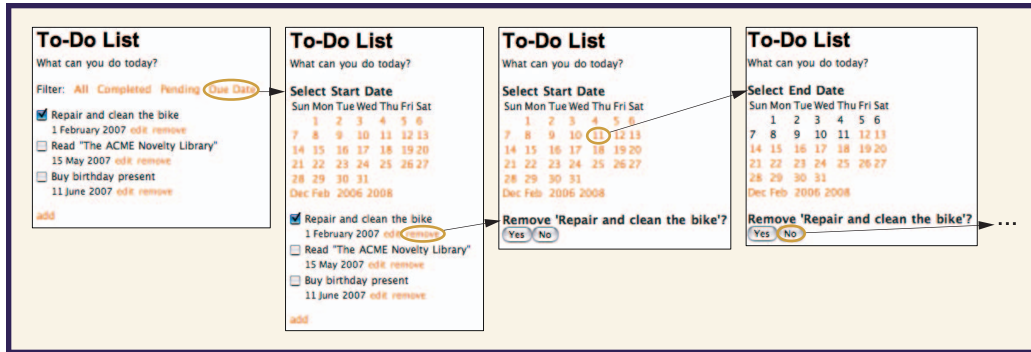


Figure 5. Multiple flows on the same page in the to-do application user interface. The user can interact freely with different components and their flows.

components can have simultaneously active control flows.

Figure 5 shows this process in the Web browser, presenting four states of the to-do application user interface. First the user decides to define a filter on the items. Then the user decides to remove the first to-do item and clicks on *remove*, which displays a confirmation dialog box. Now there are two flows simultaneously active, and the user can continue with either one. In the example, the user selects the start date, and hence sees the end date calendar next.

Implementation points

Seaside inherits most of its dynamic capabilities directly from Smalltalk's strong reflective capabilities. Here, we discuss the key implementation points that enable Seaside debugging support and the call-answer protocol.

Hot debugging and recompilation

Most of today's Web frameworks provide only weak support for Web application debugging. Typically, only a line number and a stack trace results from an unhandled exception. Fixing and finding bugs is tedious and often requires introducing log statements to gather additional data points.

Seaside adopts Smalltalk's philosophy of incremental programming in an interactive environment. Developers can add or edit code while the Web application is running. This greatly eases development. Figure 6 illustrates Seaside's debugging process—in this case, fixing an array index out-of-bounds bug without restarting the application.

Smalltalk exceptions are first-class objects that reference the original execution context from which they were raised. Exceptions are not unwound until they are properly handled. Therefore, Seaside can remember an earlier raised exception in an instance variable, and later open a debugger for this exception if the developer so desires. The developer can then fix the problem within the integrated development environment and resume the application at the point where he left off. This feature makes debugging Web applications very powerful. Recompiling and restarting the Web server isn't necessary. The developer can go right back to the questionable page to see if he fixed the error correctly and then resume the testing session.

Stack reification for call and answer

Seaside implements a call-and-answer mechanism using *continuations*, which are immutable representations of the execution stack at a certain point in time. A continuation is basically a suspended process that is resumable from the stored state several times. The following Seaside code excerpt shows a slightly simplified implementation of this call-and-answer mechanism:

```
Component>>call: aComponent
   ^ Continuation currentDo:
           [:continuation |
       self replaceWith:
           aComponent.
       aComponent onAnswer:
           [:result |
         aComponent
           replaceWith: self.
         continuation value:
           result].
       WARenderNotification
           raiseSignal]
```

Calling a component—that is, invoking the `call:` method with an instance of a component—captures a continuation and passes it

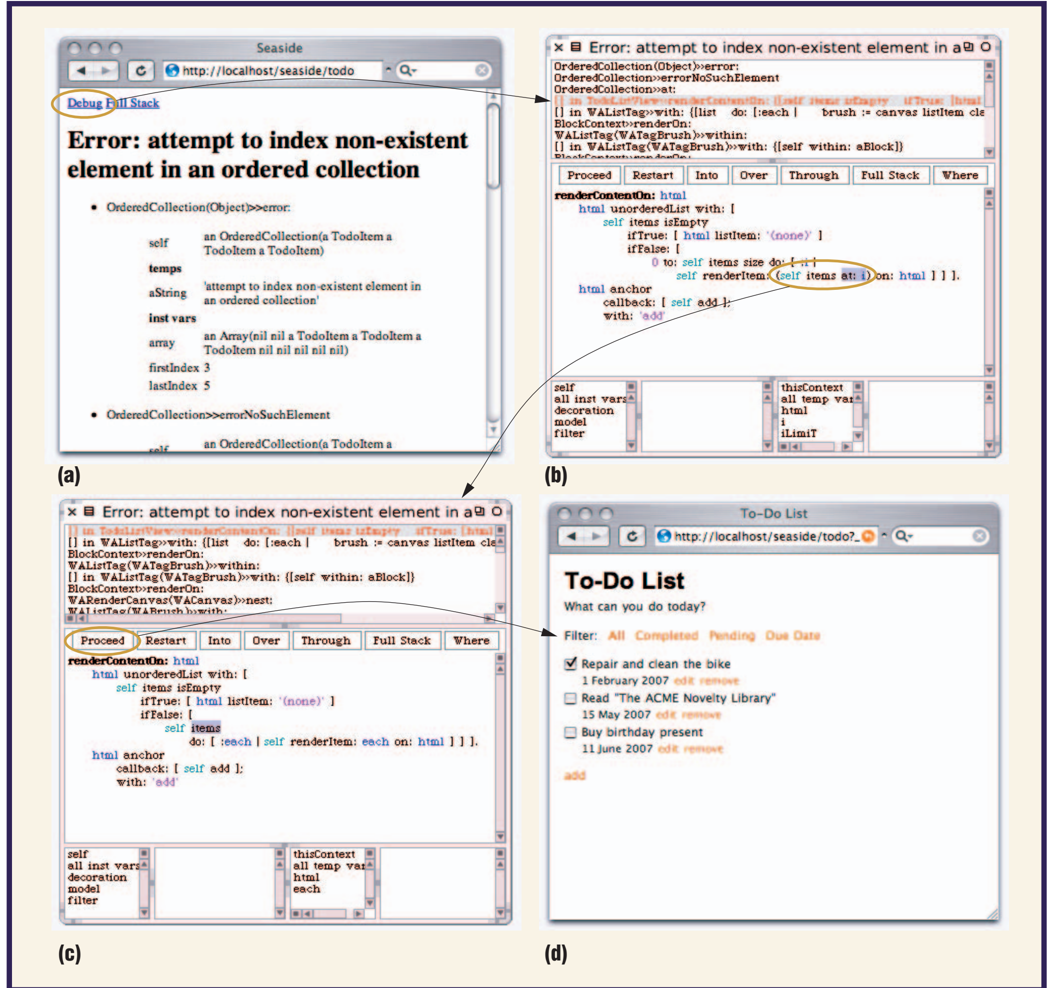


Figure 6. Debugging a Seaside application: (a) when an unhandled exception occurs, the Web browser displays a stack trace with a link called *debug*; *(b)* the developer clicks on this link to activate a debugger within the development environment to inspect variables and modify the code on the fly; (c) the debugger now displays the recompiled method, during which the Web browser waits for the server's response; (d) when the developer clicks on *proceed*, Seaside resumes processing the request that had caused the error, and the resulting page displays in the Web browser.

into a closure as the `continuation` variable. This closure replaces the current component, `self`, with the one passed to the `aComponent` method and assigns an event handler to the called component, which Seaside will evaluate when sending `answer:`. The last statement raises an exception, causing Seaside to stop the control flow evaluation and redisplay the page with the new `aComponent` component in place. This means sending `call:` to a component will not immediately return an `answer:` message to the caller after the method executes.

Later, during a subsequent HTTP request, `aComponent` might send the `answer:` message, and Seaside will evaluate the closure defined as the answer handler. This message swaps back the called component with the original one and evaluates the `continuation` with the answer argument that has been passed to the handler. This causes the `continuation` to return to the point where it had been captured (that is, to the top of the `call:` method), and return the `result` to the caller.

Thanks to Smalltalk's reflective nature, implementing a continuation in Smalltalk takes less than 30 lines of code. A continuation captures the current execution stack by fetching the active context with the `thisContext` pseudovariable, and then proceeding up the execution stack and copying all the frames.

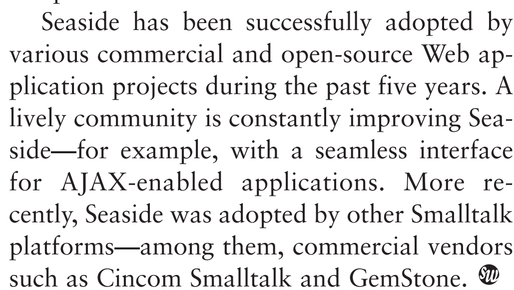
When evaluating a continuation later on, Seaside discards the active execution stack and restores and reactivates the captured one by iterating through the stored stack frames and chaining them together again. As a final step, the argument passed into the continuation is answered as a return value, making it possible to return the answer to the `call:` statement.

Seaside keeps all captured continuations within a cache, so using the Back button in the Web browser isn't a problem. For example, when defining a filter in the `TodoApplication`, the user can change the start date after setting it by pressing the Back button until the

dialog box for choosing the start date reappears. Because it's possible to invoke continuations multiple times, the `send` message to the `call:` method returns a second time, but now with a different start date, and the flow continues from there on.

Seaside provides a high-level abstraction for implementing Web applications, which alleviates developers from dealing with the page-oriented client-server interaction model imposed by HTTP. Seaside lets developers build applications as interacting components that are freely composable, and expresses an application's control flow as plain method invocations without requiring special facilities such as state machines or configuration files. The dynamic and reflective capabilities of Smalltalk are enabling factors for the Seaside core as well as for Seaside-enabled development tools.

Seaside has been successfully adopted by various commercial and open-source Web application projects during the past five years. A lively community is constantly improving Seaside—for example, with a seamless interface for AJAX-enabled applications. More recently, Seaside was adopted by other Smalltalk platforms—among them, commercial vendors such as Cincom Smalltalk and GemStone.



References

1. E.W. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM*, vol. 11, no. 3, 1968, pp. 147–148.
2. A.S. Christensen, A. Moller, and M.I. Schwartzbach, "Extending Java for High-Level Web Service Construction," *ACM Trans. Programming Languages and Systems*, vol. 25, no. 6, 2003, pp. 814–875.
3. C. Queinnec, "The Influence of Browsers on Evaluators or, Continuations to Program Web Servers," *Proc. ACM Sigplan Int'l Conf. Functional Programming*, ACM Press, 2000, pp. 23–33.
4. P. Graunke et al., "Programming the Web with High-Level Programming Languages," *Proc. European Symp. Programming* (ESOP 01), LNCS 2028, Springer, 2001, pp. 122–136.
5. C. Queinnec, "Inverting Back the Inversion of Control or, Continuations Versus Page-Centric Programming," *Sigplan Notices*, vol. 38, no. 2, 2003, pp. 57–64.
6. S. Ducasse, A. Lienhard, and L. Renggli, "Seaside—A Multiple Control Flow Web Application Framework," *Proc. 12th Int'l Smalltalk Conf.* (ISC 04), European Smalltalk User Group, 2004, pp. 231–257; www.iam.unibe.ch/~scg/Archive/Papers/Duca04eSeaside.pdf.
7. F. Rivard, "Smalltalk: A Reflective Language," *Proc. Reflection Conf.*, Xerox, 1996, pp. 21–38; www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.html.
8. G. Kiczales, J. des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

About the Authors

Stéphane Ducasse is a full professor in Listic (Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance) at Université de Savoie, where he leads the Language and Software Evolution Group. He is also the president of the European Smalltalk User Group. His research interests include dynamic languages, reflective systems, reengineering of object-oriented applications, program visualization, and maintenance. Ducasse received his PhD in computer science from the University of Nice-Sophia Antipolis. Contact him at Listic, Université de Savoie, B.P. 80439, 74944 Annecy le Vieux Cedex, France; stephane.ducasse@gmail.com.

Adrian Lienhard is a doctoral candidate in computer science at the University of Bern and cofounder of netstyle.ch, a startup company specialized in business Web application development. His research interests include reengineering of object-oriented systems, dynamic languages, language design, and Web applications. He received his MSc in computer science from the University of Bern. Contact him at Software Composition Group, Institut für Informatik, Neubrückstrasse 10, 3012 Bern, Switzerland; lienhard@iam.unibe.ch.

Lukas Renggli is a doctoral candidate in computer science at the University of Bern. He is also a core developer of Seaside and is an independent software consultant. His research interests include programming languages, software engineering, software design, meta programming, and Web application development. He received his MSc in computer science from the University of Bern. Contact him at Software Composition Group, Institut für Informatik, Neubrückstrasse 10, 3012 Bern, Switzerland; renggli@iam.unibe.ch.