Seaside – Dynamic Language Power for Web Development

Stéphane Ducasse[a] Adrian Lienhard[b] Lukas Renggli[b]

[a]*Language and Software Evolution Group, Université de Savoie, France*
[b]*Software Composition Group, University of Bern, Switzerland*

Abstract

Nowadays, many complex applications are built with a web browser as their main user interface. However, despite the increasing popularity of the web as an application platform, implementing and maintaining web applications still remains difficult and lags behind conventional desktop application development.

The underlying technologies such as HTTP for the interaction and XHTML/CSS for the presentation were originally built to display and link static documents. Unfortunately, most mainstream frameworks provide only little abstraction over the page-oriented structure imposed by those technologies. Inevitably, the goto-like manner of how pages are linked leads to spaghetti code and hampers reuse.

In this article we present Seaside, a web application framework that provides an uniform and pure object-oriented view on web applications. In this way, Seaside avoids the unwieldily goto-like style. Exploiting the reflective features of Smalltalk, Seaside reintroduces procedure call abstraction in the client-server context.

Seaside's key concepts are: (i) a component architecture supporting multiple, simultaneously active control flows, (ii) a programmatic XHTML generation, and (iii) fully supported on-the-fly debugging, code-editing, and recompilation. In this article we discuss these key features of Seaside and explain how they are made possible by the dynamic nature and the reflective capabilities of Smalltalk.

Keywords: J.8 Internet Applications, D.1.5 Object-Oriented Programming, D.3.2.i Extensible languages

Email addresses: `stephane.ducasse@univ-savoie.fr` (Stéphane Ducasse), `lienhard@iam.unibe.ch` (Adrian Lienhard), `renggli@iam.unibe.ch` (Lukas Renggli).

1 Introduction

Page-centric web development structures an application into individual scripts, each responsible for processing a user request and generating a response. Links pass the control from one script to the next. This imposes a goto-like hard-wiring of the control flow because each page is required to know what comes next. Although goto statements have been considered harmful for a long time [1], unknowingly they are still used with today's main stream frameworks and hamper the reuse of pages in different parts of the application.

Another issue of web application frameworks is the limited support they provide for composing multiple parts on the same page. The stateless nature of the client-server relationship imposes the requirement that the current state is passed back and forth between browser and server, inevitably leading to an undesired coupling between these parts.

Both problems have been tackled by more recent web application frameworks. JWIG [2], RIFE, Jakarta Struts and JBoss SEAM have proposed solutions which model control flow explicitly. However, most of these approaches do not integrate well with the rest of the code as the flow has to be specified in external XML pageflow specifications or using a different programming language. Continuation-based approaches provide the mechanisms to model control flow over several pages with one piece of code [3–8]. WebObjects, Ruby on Rails, Google Web Toolkit and .NET, on the other hand, offer better abstractions by composing an application from components, but 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 we briefly present the key challenges of modern web application development, then we present Seaside[1], a highly dynamic framework for developing web applications in Smalltalk. By exploiting the dynamic nature of Smalltalk and its reflective capabilities, Seaside offers a unique way to have *multiple control flows* active simultaneously on the same page. Furthermore, Seaside application servers do not need to be recompiled and restarted after each modification. Applications can be debugged and updated on-the-fly, which results in a considerable reduction in development time.

Seaside is open source and in productive use in many commercial applications since 2002. Seaside was originally written by Avi Bryant, but is now under active development by a growing community of contributors, among them the third author of this article.

The contributions of this article are: (1) we outline the benefits of using a

[1] http://www.seaside.st

dynamic language for advanced web development and (2) we describe the key aspects of Seaside from this perspective.

2 Web Development Challenges

We identify the key features that a web development environment or language should offer to bring web application development to the same level as a desktop application development. For a more in-depth presentation of the problems, we refer the reader to our previous work [9].

Reusable and composable components. A web application logically contains reusable components, for example input forms, sortable reports, etc. It is natural to build applications from reusable parts. In addition these components should be able to define their own control flow. Composing new components out of several other components each having a different flow should be easy to achieve.

Producing valid XHTML. Producing valid XHTML code and connecting it to the application logic is hard, especially if they have to be developed in different environments. Template based XHTML generation lacks the power and the expressiveness of the host language. A seamless integration of development tools is often missing.

Hot debugging. The efficient identification of bugs is a major challenge for web developers. Web applications become increasingly complex and the characteristics of the client-server interaction make debugging more complicated, especially since advanced debugging support is often missing with today's mainstream approaches. The ability to hot-debug exceptions and use break points would speed-up productivity significantly. Once a problem is fixed, the session can be resumed right at the place where it left off. There should be no need to restart and navigate to the problematic part of the application over and over again.

Hot recompilation. On-the-fly method recompilation, *i.e.,* recompiling a method while the application is running, is important since it supports updating code without having to restart the session. This is especially beneficial for development since it avoids time consuming edit-compile-run cycles. *Hot recompilation* is a necessary requirement for hot debugging support.

3 Smalltalk Reflective Capabilities in a Nutshell

Seaside introduces a layer of abstraction over the asynchronous interaction protocol between the client and server to provide the illusion of developing a desktop application. This high level of abstraction is made possible by the reflective capabilities of Smalltalk.

Smalltalk is a uniform language where simple principles are applied systematically. In Smalltalk everything is an object, an instance of a class. Objects exclusively communicate via message passing. In addition, there is support for *closures*, anonymous functions which can be passed around, stored in variables, and executed at a later time.

Smalltalk is written in itself and offers powerful reflective capabilities: structural reflection as well as behavioral reflection [10–12]. We limit our brief overview to the reflective capabilities that make it possible to create 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 CLOS [13], except that Smalltalk offers full access to the execution stack as we explain in the following.

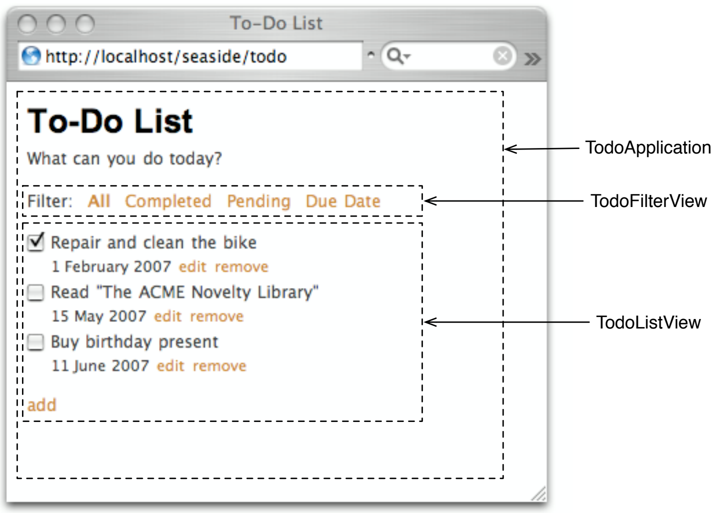
Shape and class changing. Objects are instances of classes. When the class changes, *i.e.,* instance variables are added, renamed or removed, all instances of the class are automatically migrated.

On-the-fly recompilation. Methods can be defined and re-compiled on the fly. Currently active execution contexts of a changed method run to completion using the old definition.

Stack reification. In addition to self and super, Smalltalk offers a third pseudo-variable thisContext which represents the execution stack on demand. For efficiency, the stack is reified (*i.e.,* made an object) only when the pseudo-variable is used. This object is *causally connected* to the stack it represents. This means that it not only represents the stack but any change made to the object is reflected to the execution stack itself. Hence, we are able to manipulate the stack to change the execution of the program.

4 Seaside Key Features

In this section we discuss the key features and characteristics of Seaside and how they address the web development challenges identified in Section 2. For space reasons, we do not discuss the advanced AJAX (Asynchronous JavaScript and XML) support offered by Seaside[2].



To-Do List

What can you do today?

Filter: All Completed Pending Due Date

- [x] Repair and clean the bike
  1 February 2007 edit remove
- [ ] Read "The ACME Novelty Library"
  15 May 2007 edit remove
- [ ] Buy birthday present
  11 June 2007 edit remove

add

Fig. 1. The Todo application built from different Seaside components.

4.1 An Overview

In the remainder of this article we use a simple application that allows a user to manage a todo-list of pending tasks. Figure 1 shows the main view which is composed of three components. TodoApplication is the main component which consists of the title and the two sub-components TodoFilterView and TodoListView. TodoFilterView implements filters which are applied to the list of todo items displayed by TodoListView. A todo item consists of a title, a due date, and a completion status. The status of an item can be changed by ticking its corresponding checkbox.

Components are the main building blocks in a Seaside application. A component is an instance of an user-defined subclass of Component and defines the look and the behavior of a portion of a page. On 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, *e.g.,* to temporarily pass control to another component.

The following list summarizes the three key features of Seaside and identifies the dynamic capabilities of Smalltalk that make them possible.

Programmatic XHTML generation. Following Smalltalk's principle of everything is an object, Seaside provides a very different approach to XHTML generation compared to template systems. XHTML is generated through an object layer using pure message sending. *Action callbacks* are bound to com-

[2] http://scriptaculous.seasidehosting.st

ponents using block closures.

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 technique are *continuations*, possible through Smalltalk's openness to access and manipulate the 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 features like hot-recompilation and exceptions being first-class.

In the following sections we explain how XHTML generation and action callbacks are used in Seaside and how control flow works. Debugging support and implementation points are discussed in Section 5.

4.2 A Domain Specific Language (DSL) for generating XHTML

In Seaside, XHTML code fragments are not kept in external files and are not manipulated directly, *i.e.,* there is no templating system. Instead, a high-level interface reliefs developers from checking correct tag nesting and attributes. The Smalltalk block closure syntax is used to define a domain specific language for XHTML rendering. Thus, through this dedicated language defined in Smalltalk itself, XHTML is generated programmatically.

Rendering. When the framework generates a response, each component visible on the page invokes its hook method `renderContentOn:`. It can then *render* a representation of itself on a `RendererCanvas` instance passed as method argument, by convention named `html` [3].

[3] In Smalltalk, attributes and local variables are read simply by using the name in an expression. They are written using the := construct. In a first approximation, messages follow the pattern `receiver methodName1: arg1 name2: arg2`, which is equivalent to the Java syntax `receiver.methodName1Name2(arg1, arg2)`.

```
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?' ]
```

The implementation

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

The generated XHTML

The rendering canvas returns instances of XHTML tags, so-called *brushes*. In the above code, 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. The contents, *i.e.,* nested tags or strings, is passed as argument to the with: message.

The correct nesting of tags is enforced by the closures. It is not possible to forget closing a tag, as the compiler complains about the invalid source code. Strings, as in the above code the title or notes, can be passed directly and are automatically encoded. Since the attributes are set trough accessor methods, the framework ensures the validity of the tag declaration.

Action Callbacks. So far we only discussed how a component renders itself. But components can also react to actions triggered by the user by means of *action callbacks.* Action callbacks are defined on anchors and buttons as well as on form elements such as text input fields, check boxes, select boxes, etc.

Action callbacks are defined using closures. Their execution is delayed until the action is triggered by the user. Buttons and anchors use closures without arguments. Other form fields use closures that expect one argument, where the current value of the element is passed in upon activation. The following code snippet from the class TodoListView displays the checkbox of an item and lets the user check or un-check a todo item.

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

The statement html checkbox 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 gets executed when the user clicks the checkbox, expects one argument, a boolean, reflecting the new status of the item.

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

4.3 Control Flow

Whenever a new page is requested by clicking an a link or a button, an action callback of a component is executed. In this callback, the component can change the state of the application model, as we have seen in the previous section. Moreover, callbacks can be used 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 with other components. The control flow in Seaside does not have to be linear: control statements, loops, method calls and domain code can be mixed with messages to display components. All this is done by writing *plain* source code. There is no need to build complex state machines.

The method TodoFilterView≫due is used to set a due date filter. It displays two calendar components in sequence, allowing the user to select a start and an end date as the range to filter the items. It first instantiates and *calls* a calendar component to let the user select a *start date*. When selected, this date is returned and stored in the local variable start. Since start is a date object, we can use it right away in the second calendar component to assure that the end date is after the start date. Eventually, in the last line of the method a filter is defined by a closure referencing the start and end date. This closure is then used to filter the todo items.

```
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 ]
```

As we see in the example above, there is no need to serialize state into strings and pass information from one page to another. Temporary variables are used to remember the state between the different steps of the flow.

The control flow in Seaside is based on the interplay between the methods call: and answer:, as illustrated in Figure 2. The framed Calendar in the method due is an instance of a Seaside component. The method select: in Calendar is invoked from an action callback, *i.e.,* when the user confirms the selection of the date.

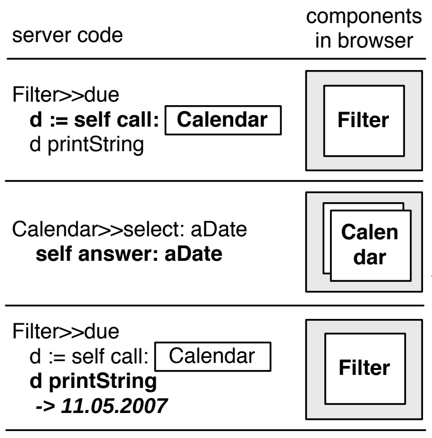


Fig. 2. call: and answer:, the basic elements of control flow in Seaside.

Call. A component can pass control to another one. During this time it is temporarily *replaced* with another component. This is achieved by sending the message call: to the old component with the new component as argument. In Figure 2 sending the method call: with an instance of a calendar component installs this calender on top of the filter component and passes it the control. Other components elsewhere on that page stay functional and can be used independently of the new component.

Answer. A component can give back control to the component from which it was called using the method answer:. When returning, a component can additionally return an object to the caller. In Figure 2, the expression self answer: aDate makes the calendar component return a date object to the filter component.

4.4 Composing Components: Multiple Control Flows

A user interface is mostly built from different parts visible on the same screen like our example illustrates. In Seaside, these parts are implemented as components that can be composed from other components.

The following code describes how the top level component of the todo application plugs together the filter and the todo list. The two components are both stored in instance variables of the component instance TodoApplication and are placed inside XHTML div elements to be rendered one after another. The title is displayed above the two components.

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

In Section 4.3 we illustrate a control flow of the filter component, the selection of a start and an end date. The list view component, however, also defines several control flows, such as a confirmation dialog displayed when a todo item is about to be deleted or dialogs for adding or editing items. Without additional changes to the code, both components can have simultaneously active control flows.

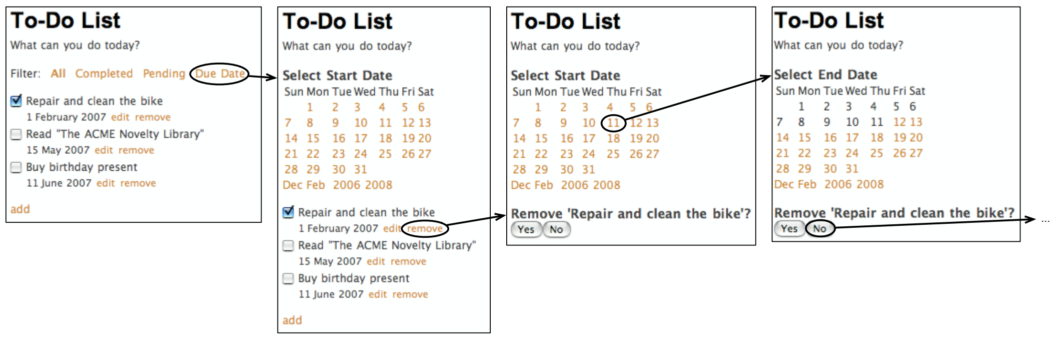

To-Do List
What can you do today?
Filter: All Completed Pending Due Date
Repair and clean the bike
1 February 2007 edit remove
Read "The ACME Novelty Library"
15 May 2007 edit remove
Buy birthday present
11 June 2007 edit remove
add
Select Start Date
Sun Mon Tue Wed Thu Fri Sat
Dec Feb 2006 2008
Remove 'Repair and clean the bike'?
Yes No
Select End Date
...

Fig. 3. Multiple flows on the same page in the todo application. The user can interact freely with different components and their flows.

Figure 3 shows this process in the web browser. It presents four states of the todo application user interface. First the user decides to define a filter on the items. Then he remembers that he wanted to remove the first todo item and clicks on *remove* which displays a confirmation dialog below. Now there are two flows active at the same time, the user can continue with either one. In the example the user selects the start date and hence gets the end date calendar presented next.

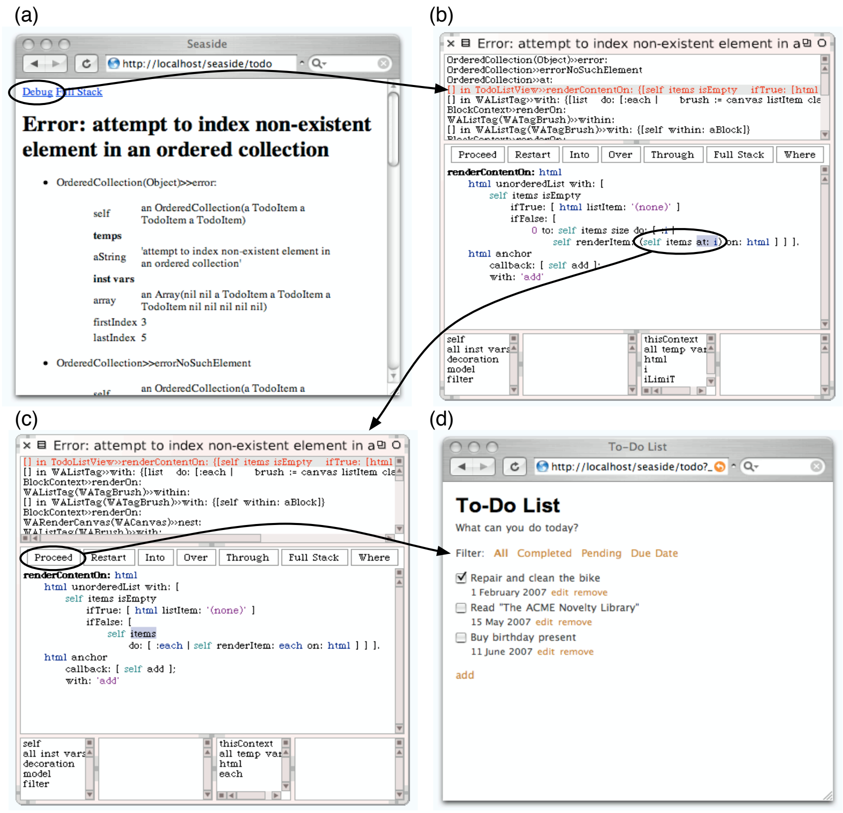
5 Implementation Points

Most dynamic capabilities of Seaside are directly inherited from the strong reflective capabilities of Smalltalk [10–12]. In this section we discuss the key implementation points which enable Seaside debugging support and the call/answer protocol.

5.1 *Hot Debugging and Recompilation*

Most of today's web frameworks have weak support for web application debugging. Most of the time, only a line number and a stack trace results from an unhandled exception. Fixing and finding bugs is tedious and often has to be accomplished introducing log statements to gather additional datapoints.

Smalltalk's philosophy of incremental programming in an interactive environment is supported by Seaside. Code can be added and edited while the web application is running. This greatly eases development.



(a)

Error: attempt to index non-existent element in an ordered collection

- OrderedCollection(Object)>>error:

self	an OrderedCollection(a TodoItem a TodoItem a TodoItem)
temps	
aString	'attempt to index non-existent element in an ordered collection'
inst vars	
array	an Array(nil nil a TodoItem a TodoItem a TodoItem nil nil nil nil nil)
firstIndex	3
lastIndex	5

- OrderedCollection>>errorNoSuchElement

self	an OrderedCollection(a TodoItem a

(b)

Error: attempt to index non-existent element in a

```
OrderedCollection(Object)>>error:
OrderedCollection>>errorNoSuchElement
OrderedCollection>>at:
[] in TodoListView>>renderContentOn: {[self items isEmpty   ifTrue: [html
[] in WAListTag>>with: {[list   do: [:each |     brush := canvas listItem cla
BlockContext>>renderOn:
WAListTag(WATagBrush)>>within:
[] in WAListTag(WATagBrush)>>with: {[self within: aBlock]}
```

```
renderContentOn: html
    html unorderedList with: [
        self items isEmpty
            ifTrue: [ html listItem: '(none)' ]
            ifFalse: [
                0 to: self items size do: [ :i |
                    self renderItem: (self items at: i) on: html ] ] ].
    html anchor
        callback: [ self add ];
        with: 'add'
```

```
self              thisContext
all inst vars     all temp vars
decoration        html
model             i
filter            iLimiT
```

(c)

Error: attempt to index non-existent element in a

```
[] in TodoListView>>renderContentOn: {[self items isEmpty   ifTrue: [html
[] in WAListTag>>with: {[list   do: [:each |     brush := canvas listItem cla
BlockContext>>renderOn:
WAListTag(WATagBrush)>>within:
[] in WAListTag(WATagBrush)>>with: {[self within: aBlock]}
BlockContext>>renderOn:
WARenderCanvas(WACanvas)>>nest:
```

```
renderContentOn: html
    html unorderedList with: [
        self items isEmpty
            ifTrue: [ html listItem: '(none)' ]
            ifFalse: [
                self items
                    do: [ :each | self renderItem: each on: html ] ] ].
    html anchor
        callback: [ self add ];
        with: 'add'
```

```
self              thisContext
all inst vars     all temp vars
decoration        html
model             each
filter
```

(d)

To-Do List

What can you do today?

Filter: All Completed Pending Due Date

- [✓] Repair and clean the bike
  1 February 2007 edit remove
- [ ] Read "The ACME Novelty Library"
  15 May 2007 edit remove
- [ ] Buy birthday present
  11 June 2007 edit remove

add

Fig. 4. Debugging a Seaside application – a continuous loop fixing a bug without restarting the application.

With Seaside, when an unhandled exception occurs, as shown in Figure 4, the web-browser displays a stack trace (a) with a link called *debug*. Clicking this link, the developer activates a debugger (b) within the development environment which lets him inspect variables and modify the code on-the-fly. The developer replaces the code part that caused the *index out of bounds* exception with a more elegant loop construct. The debugger now displays the recompiled method (c). During this time, the web browser keeps waiting for the response of the server. On hitting *proceed*, the processing of the request, which had caused the error, is resumed and the resulting page is finally displayed in the web browser (d).

Smalltalk exceptions are first class objects that reference the original execution context they have been thrown from. Exceptions are not unwound until they are properly handled. Therefore, Seaside is able to remember an earlier raised exception in an instance variable and, later on, it can open a debugger on this exception if the user decides to do so. The developer is then able to fix the problem within the IDE and after fixing the problem, resume the application at the point where he left off. This feature makes debugging web applications very powerful. There is no manual recompilation and restarting of the web server required. The developer is put right back into the questionable page where he is able to see if he fixed the error correctly and is able to resume the testing session.

5.2 Stack reification for Call and Answer

Seaside's Call and Answer mechanism is implemented using continuations, immutable representations of the execution stack at a certain point in time. A continuation can be seen as a suspended process which is resumable from the stored state several times. The following code excerpt of Seaside shows a slightly simplified implementation of the 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, *i.e.,* invoking the method call: with an instance of a component, captures a continuation and passes it into a closure as the variable continuation. This closure replaces the current component self with the one passed to the method aComponent and assigns an event handler to the called component, which will be evaluated when sending answer:. The last statement raises an exception, causing Seaside to stop the evaluation of the control flow and redisplay the page with the new component aComponent in place. This means that sending call: to a component will not immediately return after executing the method like normal ones would.

Later on, during a subsequent HTTP request, aComponent might send the message answer: and the closure defined as the answer handler will be evaluated. It swaps back the called component with the original one and evaluates the continuation with the answer argument which has been passed into the handler. This causes the continuation to return from the point it has been captured, *i.e.,* the top of the method call: and return the result to the caller.

Due to the reflective nature of the language continuations can be implemented with less than 30 lines of Smalltalk code. A continuation captures the current execution stack by fetching the active context with the pseudo-variable thisContext and walking up the execution stack and copying all the frames.

When a continuation is evaluated later on, the active execution stack is discarded and the captured one is restored and reactivated. This is done by iterating trough the stored stack frames and chaining them together again. As a last step, the argument passed into the continuation is answered as a return value, making it possible to return the answer to the call: statement.

Since Seaside keeps all captured continuations within a cache, the use of the back button in the web browser causes no problems. For example, when defining a filter in the Todo application the user decides to change the start date *after* already having set one. He does so by pressing the back button until he gets back to the dialog to choose the start date. Since continuations can be invoked multiple times, the send to the method call: returns a second time, but now with a different start date, and the flow continues from there on.

6 Conclusion

In this article we presented the most important functionality offered by the Seaside web application framework. We identified the following dynamic language capabilities of Smalltalk that are key to facilitate these features.

- *Block closures* are used for the XHTML rendering protocol and for the action callback mechanism.
- *Full stack reification* allows one to implement continuations within Smalltalk. Continuations facilitate per component control flows to be implemented in one continuous piece of code.
- *Reflective capabilities* such as on-the-fly recompilation and automatic migra-

tion of instances enable seamless integration of the Smalltalk IDE, namely the debugger.

Summarizing, with Seaside a developer builds applications as interacting components that are freely composable. The control flow of an application is expressed as plain method invocations without requiring special facilities like state machines or configuration files – all the power of object-oriented programming is at hand. The fulfillment of today's web application demands with one uniform dynamic language pays off with ease of use, rapid development, and excellent debugging support.

References

[1] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[2] Aske Simon Christensen, Anders Moller, and Michael I. Schwartzbach. Extending java for highlevel web service construction. *ACM Transaction on Programming Languages and Systems*, 25(6):814–875, 2003.

[3] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, may 2000.

[4] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, pages 23–33, 2000.

[5] Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *Proceedings of ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 122–136, 2001.

[6] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[7] Christian Queinnec. Continuations and web servers. *Higher-Order and Symbolic Computation: an International Journal*, pages 1–16, 2004.

[8] Paul Graham. Beating the averages. http://www.paulgraham.com/avg.html.

[9] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC'04)*, pages 231–257, September 2004.

[10] Fred Rivard. Pour un lien d'instanciation dynamique dans les langages à classes. In *JFLA96*. INRIA — collection didactique, January 1996.

[11] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[12] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[13] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.