Seaside – A Multiple Control Flow Web Application Framework ⋆

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

[a] *Software Composition Group*
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

[b] *netstyle.ch GmbH*
Bern, Switzerland

Abstract

Developing web applications is difficult since (1) the client-server relationship is asymmetric: the server cannot update clients but only responds to client requests and (2) the navigation facilities of web browsers lead to a situation where servers cannot control the state of the clients. Page-centric web application frameworks fail to offer adequate solutions to model control flow at a high-level of abstraction. Developers have to work manually around the shortcomings of the HTTP protocol. Some approaches offer better abstractions by composing an application out of components, however they still fail to offer modeling control flow at a high level. Continuation-based approaches solve this problem by providing the facilities to model a control flow over several pages with one piece of code. However combining multiple flows inside the same page is difficult.

This article presents Seaside. Seaside is a framework which combines an object-oriented approach with a continuation-based one. A Seaside application is built out of components (*i.e.,* objects) and the logic of the application benefits from the continuation-based program flow infrastructure. Seaside offers a unique way to have *multiple control flows* on a page, one for each component. This enables the developer to write components that are highly reusable and that can be used to compose complex web applications with higher quality in less time.

⋆ We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1) and netstyle.ch.

Email addresses: `ducasse@iam.unibe.ch` (Stéphane Ducasse), `alienhard@netstyle.ch` (Adrian Lienhard), `renggli@netstyle.ch` (Lukas Renggli).

1 Introduction

With the victory of the World Wide Web as an information platform the Internet has become one of the most important platforms for applications. When the WWW was invented in 1989 its intended purpose was to link static documents. But soon simple forms processed by cgi-scripts [1] enabled the user to enter data: Web applications were born. However many of the web applications that exist today are much more complex than those based on the original form technology.

One of the major advantages of web applications, compared to conventional client-server applications, is that the user does not need to install a special program for each application he likes to use. Furthermore web applications are easier to deploy – they just have to be installed on a single server. A web application is an application running on a web server which interacts with web browsers over the Internet. HTTP is the protocol used for communication: An interaction between server and client over HTTP is a sequence of requests from the client and responses from server. The server usually responds with a HTML document that is then rendered in the web browser on the client side. Since HTTP is stateless, the server cannot identify the request associated with the response and the server can only respond to requests from the client but cannot update the client spontaneously.

A major difficulty of web applications is that they are based on the asymmetric design of the HTTP Protocol and the fact that it is stateless [2]: The server is unable to send updates to the client and has to wait for incoming requests. Moreover, the web server is unable to control the navigation facilities in web browsers, like the back- and forward-buttons or the capability to open new windows of the same page. These navigation facilities lead to synchronization problems with the state of the server and its clients. This means that the server has to deal with the fact that for one question asked (*e.g.,* filling in an order form) there may be more than one answer. This happens if the user uses the back button or clones a window and submits a form a second time. Thus a user can follow several paths of interaction in a session at the same time.

Another major problem arises from the architecture of web pages: Each HTML link or form-action encodes in its URL the file which handles the request – and additional parameters that can be passed with the query string. Whereas this scheme is appropriate for static documents it is not ideal for programming complex applications. Without an appropriate abstraction it leads to the problem that control flow logic which would ideally be implemented in a single piece of code has to be split into different parts – one for each request sent by the client. From this point of view URLs with their query strings are procedure calls that do never return: control flow has to be defined in a goto-like

manner which leads to poor designs [3] [4]. This problem remained unsolved since the existence of cgi-scripts – even by widely used page-centric frameworks or newer technologies such as WebObjects [5] or ASP.NET [6]. Several frameworks like Jakarta Struts [7], JWIG [8,9] or RIFE [10] have proposed solutions that model control flow explicitly on a higher level of abstraction.

Continuation-based frameworks propose a more innovative approach to web-serving [11] [12] [13] [14] [15] [2]. Using continuations enables the web application server to offer a procedural view to the programmer that makes it very convenient to model a flow of pages using a single piece of code. Continuations increase application control flow abstraction. However, current continuation-based frameworks make it difficult to combine several control flows into a single page. This hampers the design of truly reusable and composable components from which web application could be built.

Seaside [16] is a mature framework which combines an object-oriented approach with a continuation-based one. A Seaside application is constructed out components (*i.e.,* objects) and the logic of the application benefits from the power provided by continuation-based program flow infrastructure. This combination enables a framework in which applications are built out component each having its *own* control flow. This unique *multiple control* flow enables the definition of highly reusable components that can be even used multiple times on the same page and the definition of control flow at a high level of abstraction. Seaside is developed in Squeak [17], an open-source platform and development environment for Smalltalk, by *Avi Bryant* and *Julian Fitzell*. The last two authors are using Seaside professionally.

The contributions of the paper are: (1) a description of the most common problems in web application development, (2) the description of the Seaside framework a combination of object-oriented application composition with continuation-based flow and (3) the description of the multiple control flow provided by this combination. Before presenting the core concepts of Seaside and going into some details of its implementation, we describe the problems that a web application has to face. As a motivating example and for illustration purposes we use a simple web shop example throughout the rest of the article.

2 A Web Shop as a Motivating Example

Our shop that comes as an example application with Seaside, selling sushis, is composed of various elements: it provides a search on the left, a batched list of products with a detail view of a selected product in the middle, and a shopping cart on the right (see Figure 1). The title bar, the search and the

shopping cart are displayed on every page.

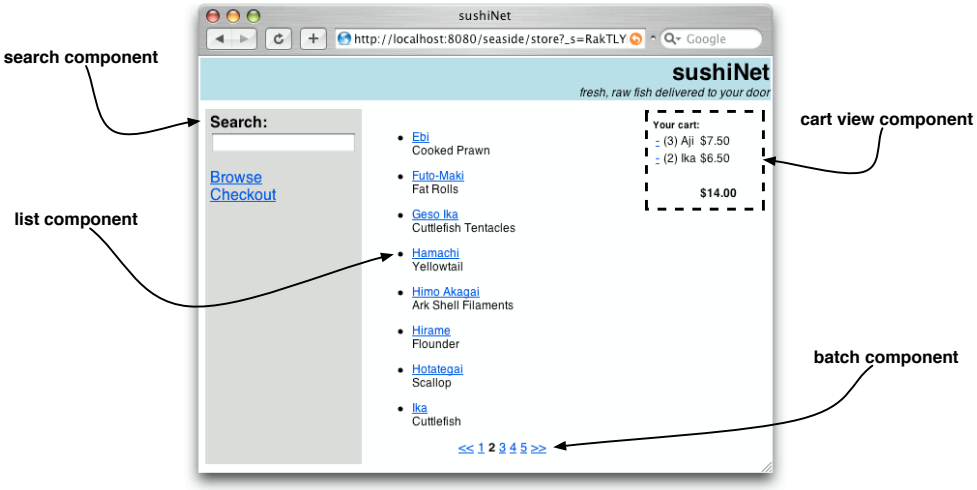

Fig. 1. The Sushi Web Shop and its components.

The shop application also defines several control flows that the user can follow. For example the checkout process, which is the following (see Figure 2): When the user wants to check out, he first has to confirm the contents of the cart, and if he agrees, he is asked for the shipping address. Subsequently, a dialog asks him if he wants to have a separate billing address. If he answers with yes, an additional address dialog is displayed. After having entered the payment information the order is finished and a confirmation page is displayed. Between each of these steps there is a validation logic that may decide to redisplay the previous dialog with an error message.

3 Current Limits of Web Application Development

There are basically two kinds of problems: the ones related to the control flow logic and the ones concerned with the state which has to be remembered during user interaction.

3.1 Control Flow Problems

Many of today's frameworks (such as Servlets/JSP [18] [19], PHP [20], ASP [21], JSP [19] or Zope [22]) fail to provide a high-level abstraction over how pages are linked. Indeed, a web application has to model *control flow*.

The control flow logic (as illustrated by the check-out in the shop example) would ideally be implemented in one single piece of code with common pro-

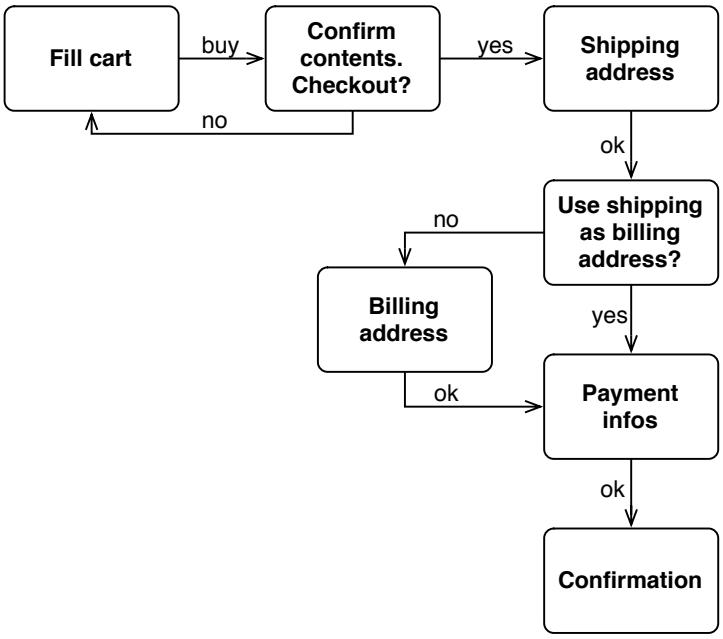

Fig. 2. The checkout process in our shop.

gram statements. Unfortunately, this way of modeling web applications is inverse to what the HTTP protocol with its request/response model implies. As long as the framework does not provide the abstractions, the developer is forced to use unintuitive workarounds. Here is a list of the most important problems related to control flow.

Mixing Application Logic and Component Logic. The user session in a in a web application can be seen as two repeating tasks that take place: The first is generating the page and the second is processing data when the user submits it. In many frameworks these two parts are disconnected and the processing takes place in two separate executions of the server program. The first part produces the HTML representation of the page. The latter processes the data and starts the generation of the next page, *e.g.,* by doing a validation and by deciding what to do next *i.e.,* which rendering code to execute to produce the next page. It might be the same page with the previously entered values and additional error messages in case of a validation failure *or* the next step in the sequence of control flow. Thus, the decision on "what to show next" is coupled with the processing of the data of the last page rather than being defined in a control flow logic at a higher level. In page-centric frameworks plugging together components of the application and defining how they should interact with each other (application logic) has to be done in each component itself (component logic): each page in a sequence of pages has a hardcoded pointer to the next one. This is comparable to programming with goto-statements which never return.

Difficult Composition of Control Flows. Continuation-based web servers were a big step forward to support a better abstraction of application control flow [12] [14] [15]. However, they do not allow an easy composition of multiple control flows coming from the different components that composed the application within the same page. For example, the user can be browsing a detailed description of an item while on the same page, it can get prompted to know whether the numbers of item he selected is correct, and he does not have to be forced to answer the question but can perform all kinds of other tasks before and in parallel. Not being able to easily combine multiple control flows hampers the definition of truly reusable and composable components.

Controlling Program Flow. In some situations we have to strictly control which requests are valid and which should not be processed anymore. Because the user can clone browser windows and go back in the history of pages he is able to send the same or different requests more than once. In our shop, the user should not be able to add additional items to his shopping cart after having payed for it.

3.2 State Problems

A web application typically has to deal with the following kind of states: (1) user interface state (*e.g.,* remember which is the current page number in a batched list) (2) domain model state which has to make persistent before it gets stored in a database (*e.g.,* the billing address in the shop checkout process) and (3) state related to the user session (*e.g.,* customer id of the user). But handling state in the web's client-server context is hard because the control flow is not linear [23] [24]. This is mainly caused by the fact that HTTP is stateless and by the capabilities of web browsers: The users are used to going back in the browser or cloning windows to undo their last step or to interact with the application with several windows in parallel: A form in a web application can be submitted more than once and maybe even at a later time. Since the server cannot update clients, the page shown in the browser may be out of date.

What makes handling state even more complicated is the fact that the facilities to pass state from the client to the server or vice versa are very limited: Information can be encoded into URLs, submitted in hidden form fields or stored in cookies. Cookies only hold information per session and not per browser window and are therefore not suited to store state that is individual for each path of user interaction.

Instead of passing information back and forth from the client to the server, storing state on the server in the session object would be an obvious solution.

However, this approach fails because there can be more than one path of user interaction in parallel in the same session. Thus, especially in page-centric frameworks the developer is forced to encode state in the responses sent to the client by manipulating query strings or using hidden form fields. Those values then have to be decoded from the next request of the user. However, this solution is not only cumbersome, it also leads to the following problems:

Encoding State in Pages. When domain model state should exist over a sequence of pages, each page has to pass the information from all previous pages to the next page. This makes code almost not reusable since one page depends on the previous ones.

Name Clashes. There can be name clashes in URLs or in form fields. The programmer has to take care that identifiers are unique across each page. This gets especially bad if we like to reuse code. For example, it is not easily possible to write reusable user interface components such as batched lists or tab widgets and use them more than once on the same page.

Mixing Presentation and Domain Logic. To encode state in the pages, the developer has to mix the generation of HTML with domain logic using string concatenation or templates. This is cumbersome and leads to unreadable code.

To summarize, developing web-applications face the following problems: it is difficult to (1) define *reusable* components with their *own* logic, (2) compose the logic of an application out of component logic, and (3) represent the state of an application.

4 Seaside Main Building Blocks: Components

The main entities in a Seaside application are objects, but called components in Seaside parlance. They are responsible of defining the user interface and the control flow of application part. A component is an instance of a user-defined subclass of Component that defines the look and the behavior of a portion of a page. Components therefore can be seen as views and controllers of the MVC triad [25]. Note that contrary to file-based frameworks component instances often exist during the whole lifetime of a user session, *e.g.,* the component displaying the cart in our shop. On each request the session lets the components evaluate their *callbacks* and *render* their current state by writing onto the response stream as presented hereafter.

Rendering. Each component which is visible in an application gets its hook method renderContentOn: invoked to render itself, *i.e.,* to generate an XHTML representation of itself. The component's method renderContentOn: is invoked with as argument an instance of HtmlRenderer, named by convention *html*.

Such an instance is a stream-like object that understands different messages to conveniently create most of the XHTML tags [26].

The following example shows the shop's root component render method which defines a table with one row containing the main title and the subtitle, both embedded in div-tags with specific CSS classes (see the result in Figure 1). Nesting of XHTML tags (*e.g.,* table:) is done by using Smalltalk blocks. The last call to html passes the root component's task which defines the body of the application. This is discussed in the following paragraphs.

```
Store ≫ renderContentOn: html
    html cssId: 'banner'.
    html table: [
        html tableRowWith: [
            html divNamed: 'title' with: self title.
            html divNamed: 'subtitle' with: self subtitle ] ].
    html divNamed: 'body' with: self task.
```

The example shows how XHTML code is generated programmatically. This is very convenient because control- and loop-statements can be defined uniformly without the need to switch between Smalltalk code and HTML definitions.

Embedding Components. To compose an application out of different components, components can be embedded into each other. This is what the last line of the previous example is doing. The method divNamed:with: – as well as most of the other methods of HtmlRenderer – takes as second argument a component or any other object that can be rendered. In our example it is a *task* that the store component holds as an instance variable. A task is a special kind of component that only defines control flow. It will be discussed in more detail in Section 5.

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

Action callbacks are defined using blocks: For buttons and anchors blocks without an argument are used, for form fields blocks are evaluated with one argument, the current value of the element. The following code snipped (from StorePaymentEditor) shows the definition of a select box which lets the user choose the credit card he likes to pay with:

```
    html
        selectFromList: self cardTypes
        selected: self cardType
        callback: [ :value | self cardType: value ]
```

```
labels: [ :each | each abbreviation ]
```

The first argument passed to the method selectFromList:selected:callbacl:labels: is a collection of classes, one for each supported credit card type. The second argument defines which object, if any, of the previous list should be selected. StorePaymentEditor holds an instance variable cardType to remember the selection and defines the accessors cardType and cardType:. The third argument is the action callback block which itself takes one argument. When the user submits the form which holds the selected value, the block is evaluated with the selected value. The value is one of the objects of the provided list, in our case a class. In the example the value is stored for later usage. The last argument specifies how the items are labeled in the select box, in this example each class from the list of possible card types responds to abbreviation returning a description string.

Most of the time action callbacks to anchors or buttons call methods that define component or application control flow. This is a very central mechanism in Seaside: the specification of *control flow* by means of temporarily passing control from one component to another in a non-goto like manner (*i.e.,* in a procedural or method invocation manner). This facility is provided on component level which enables to define multiple control flows independently from each other. This is the subject of the following sections.

5 Multiple Control Flows in Seaside

As each component defines its own control flow independently of the other components displayed on the same page and a component can be composed of multiple other components, a component or an application has multiple control flows. Whenever a new page is requested by hitting a link or a button, one of the components is able to go one step further in its own flow, while the other remain in the same state. Furthermore, control flow in Seaside does not have to be sequential: control statements, loops, function calls and domain code might be mixed with messages to display new web pages. All this is done simply by writing *plain* Smalltalk code, there is no need to build state machines like in Struts [7], JWIG [8] or RIFE [10].

In the following we present how a component defines its own control flow, then we describe how a component can pass its control to another component and finally how multiple components can be assembled together to create page with multiple control flows.

Each component can have its own control flow that may describe simple widget logic or more advanced control flow. The method StoreTask ≫ go below defines the control flow of the StoreTask component. It specifies the central logic of our shop: the sequence of pages which are shown when browsing through the shop and performing the checkout. It models the process shown in Figure 2 precisely and it almost reads like a piece of pseudo-code. There are different helper-methods called and each of them displays information, offers choices or collects data from the user. Users are able to browse products and put them into their shopping cart: this subprocess is implemented by another component which is invoked by fillCart:. Finally the user orders the products by providing a shipping and, if necessary, a billing address and payment information. When everything has been completed correctly, the method ship:to:billTo:payWith: executes the final ordering and displayConfirmation confirms the order to the customer.

```
StoreTask ≫ go
    | cart shipping billing creditCard |
    cart := StoreCart new.
    [   self fillCart: cart
        self confirmContentsOfCart: cart ] whileFalse.
    shipping := self getShippingAddress.
    billing := (self useAsBillingAddress: shipping)
        ifFalse: [ self getBillingAddress ]
        ifTrue: [ shipping ].
    creditCard := self getPaymentInfo.
    self ship: cart to: shipping billTo: billing payWith: creditCard.
    self displayConfirmation.
```

This method models the shop control flow at a high level of abstraction: It defines how parts of the application are composed. Information from one part can be:

(1) passed to the next part, *e.g.,* for example the instance of StoreCart, cart is passed to be filled, then passed to the shipping part,
(2) used to decide what to do next, *e.g.,* useAsBillingAddress: asks the user if he wants to use the shipping address as billing address) or
(3) stored in a temporary variable for later use, *e.g.,* shipping.

There is neither a need to pass information from one page to another, so that it is available later one, nor the need to model logic that encodes which component to display next in a called component.

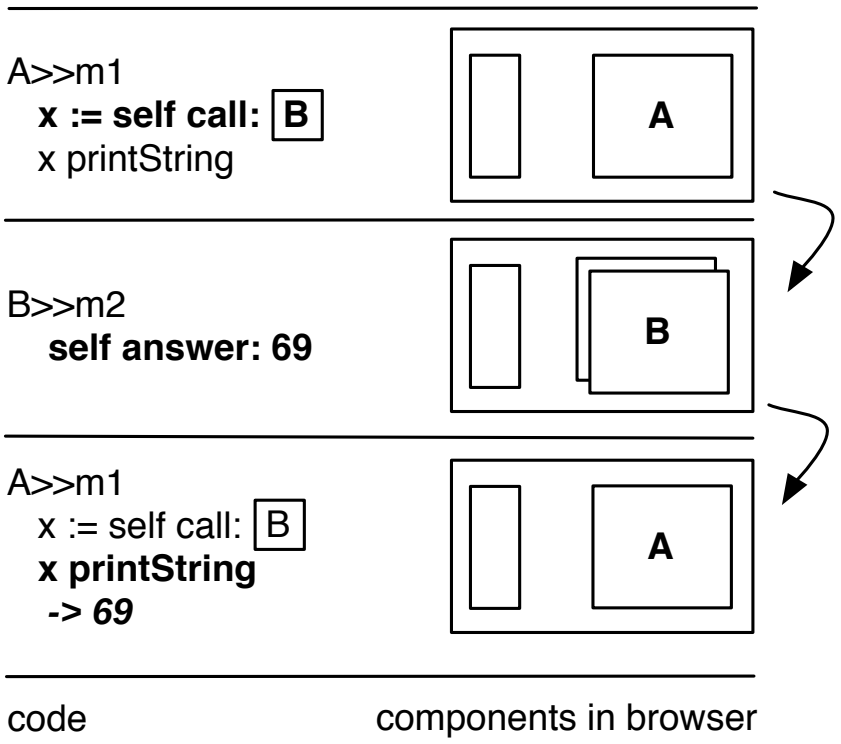

Fig. 3. call:/answer: the basic elements of control flow in Seaside. The framed B in the method m1 is a graphical object displayed as the window B in the web browser. m2 is a method that is invoked in a callback *i.e.,* when an action on the component B is invoked such as a button pressed or a link clicked.

5.2 Call and Answer: Passing Control to Another Component

In Seaside the control flow is based on the interplay between the methods call: and answer: (see Figure 3). Several other methods are built on top of these two messages, such as inform: to display an information dialog or request: to ask the user for a string and make the developer's work more convenient.

Call. At any time a component can pass control to another one. During this time it is temporarily *replaced* by the other component. This is achieved by sending the message call: with the new component as parameter to the component that should be replaced. In Figure 3 sending the method call: to the currently displayed component with the component B installs the component B and passes it the control. Other components elsewhere on that page stay functional and can be used independently of the new component.

The sushi listing in our web shop illustrates call: use. The link's action handler of each sushi item is implemented as shown in the following piece of code. The effect of this code is to replace the current main component (the batched list)

by a component displaying detailed information about the chosen product.

```
StoreFillCart ≫ displayItem: anItem
   main call: (StoreItemView new
                        item: anItem;
                        cart: cart;
                        yourself)
```

Answer. At any time a component can give back control to the component from which it was called using the method answer:. Every call in Seaside eventually returns at some point and it is even able to return a value to the caller. This makes it possible to pass resulting objects from called components back into the control flow and avoids the necessity to pass around strings with requests and responses.

To pass the control back, a called component should send to itself the message answer: aValue. In Figure 3, the expression self answer: 69 makes the component B return the number 69 to component A. As argument of the answer: message, any object can be given and this object will be handed back to the caller of the method call:. For example after the expression x := self call: B, the value of x is the value passed as argument in the expression self answer: 69 of the method m2. For convenience, if there is no return value needed, one might also call answer that is implemented as self answer: nil.

It is then possible to collect information by calling a component which will return an object. In case of a confirmation dialog this might just be a boolean – but it can also be a business object as the following example of our shop demonstrates: Whenever a user has finished selecting his sushis, he has to provide valid payment information. Within the component StoreTask the method getPaymentInfo calls StorePaymentEditor and returns the result of this message send.

```
StoreTask ≫ getPaymentInfo
   ^self call: StorePaymentEditor new.

StorePaymentEditor ≫ ok
   self answer: (cardType new
                        name: name;
                        number: cardNumber;
                        expiry: (Date newDay: 1 month: month year: year)).
```

The message call: replaces the current component instance of StoreTask with a new instance of StorePaymentEditor and stops the execution until the user has provided valid payment information. The class StorePaymentEditor implements a method called ok, which is evaluated when the user is hitting the okay button. The method ok creates a new instance of the selected cardType and passes the

information collected in the dialog (the name, cardNumber, month and year) to the newly created object. The method answer: returns this object to its caller (StoreTask ≫ getPaymentInfo) where Seaside resumes the control flow which has been previously stopped at this position.

The real power of the call and answer mechanism relies in the capability to build a flow of components which embed several complex user interactions. Seaside allows one to call different components one after the other, using control-statements such as loops and conditional clauses, or other non web related code in-between these calls. An important point is that passing control to another component is done with normal message sending: A method returns and the execution continues from this point – even if this is at an undefined point in the future. This prevents the goto-like definition of control flow without the possibility to return.

5.3 Composing Components: Multiple Control Flow

The examples discussed so far, illustrate one flow of control at a time: StoreTask ≫ go (see Section 5.1) models the control flow of the application at the highest level. This control flow is defined at the top component, however with Seaside it is possible to let each sub-component define its own control flow. In the context of our web shop example, this enables the user to interact *at the same time* with the sushi list (*i.e.,* getting a detail description of the sushi, browsing the sushi list) *and* with the shopping cart.

Figure 4 shows this process in the web browser. It presents four states of the webshop user interface. First on the left we see the description of a sushi (Chuboro Magoro) which can be added to the cart displayed on the right. The card contains five California Rolls and three Chuboro Magoro. In the cart, the user can change his order by pressing the minus sign in front of the line. This is what the user did, and the right component is replaced by a dialog box that checks if the user really wants to remove some of the delicious sushi from the cart (1). The user is hesitating and quits the description of the Chuboro Magoro sushi to browse the sushi list (2). Finally he decides to remove all the Chuboro Magoro sushi from his order (3).

The following code describes how the available sushi list and the cart are plugged together. The two components are both stored in instance variables of the component StoreFillCart and are placed inside a HTML table to be rendered next to each other on the same page.

```
StoreFillCart ≫ renderContentOn: html
    html table: [
        html tableRow: [
```

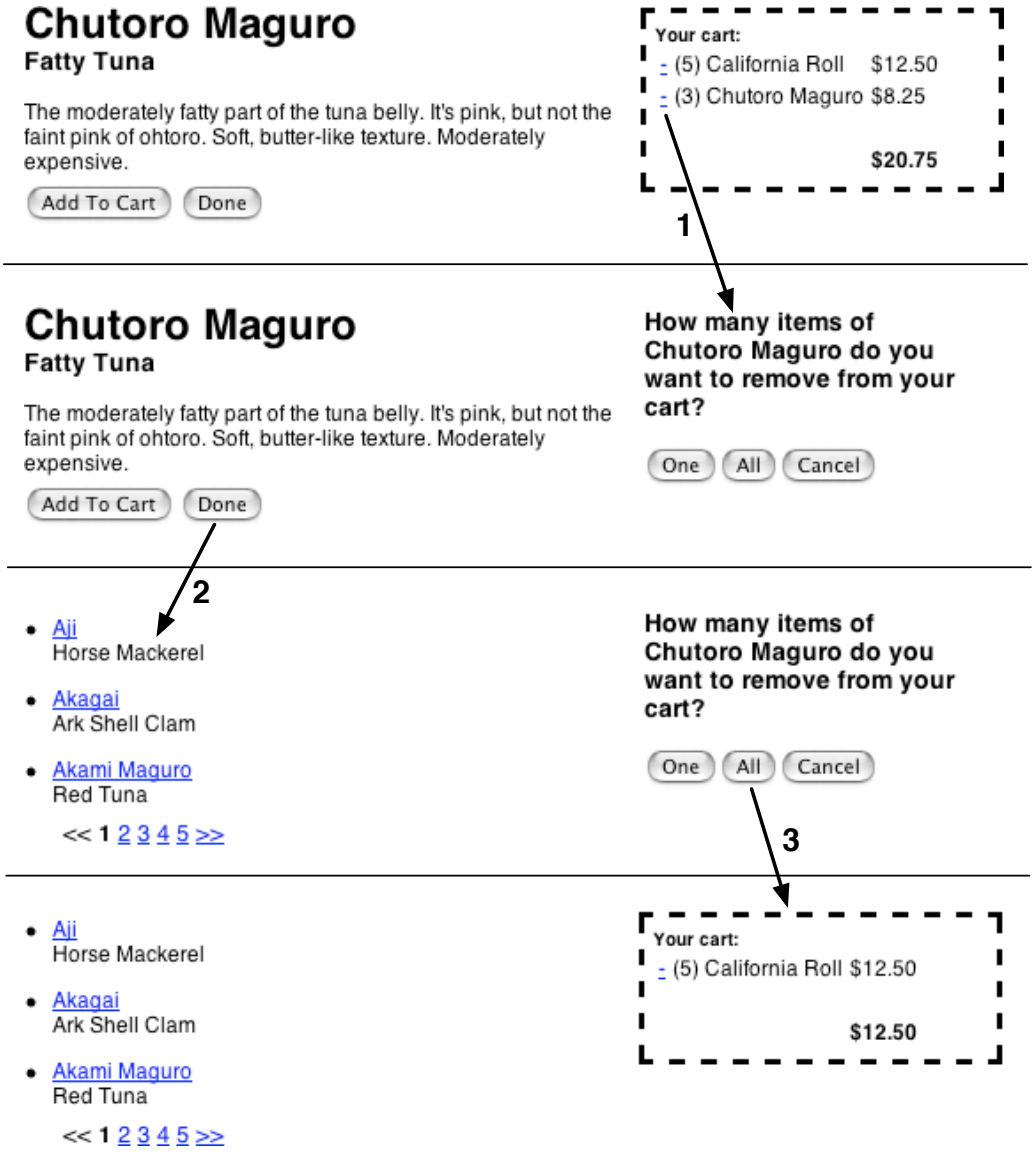

Fig. 4. Multiple flows on the same page in the web shop. The user can interact freely with different components and their flows of the sushi shop.

```
    html tableData: productList.
    html tableData: cartView ] ]
```

Each component can have its own control flow. Such a flow is either bound to the component itself (if our component is a subclass of Task) or defined in actions connected with user events (if our component is a subclass of Component). Whenever there is an interaction with a component and there is either a message send to call: or answer: a different component is shown instead of the old one and the flow of that particular component is able to advance. All the other components on the page however stay in the same state until there

is an user interaction allowing them either to return or to call another one.

6 Managing Non-Linear Control Flow

The control flow capabilities of Seaside provide a non-linear control flow implicitly – from the view of the developer the code is linear and does not need any additional logic to handle backtracking in the control flow. However there are still some problems related to the application state. Seaside handles backtracking in the control flow by resuming computation at the right place. But since the user interacting with the application modifies state, we also have to ensure that it is correctly handled. As discussed in the problems Section (see Section 3) we have to deal with user interface and domain model state. It is important to handle the latter correctly as it is crucial for security reasons.

In the case of our shop, when the user has checked out and already payed the products, it should not be possible for him to go back and modify the cart business object by adding additional sushi to it. Similarly, the user would not be happy if he were to accidently submit the credit billing page a second time, and be charged twice as a result.

This leads to two different kinds of situations: In the first situation we would like to support the user to backtrack. This is solved by backtracking user interface and domain model state so that when the user goes back, the old state is restored. The second situation is when the user is forbidden to backtrack. This is solved by specifying the conditions in which a request should not be processed anymore. We present these two situations now.

6.1 Backtracking State

Each component in Seaside has its own state which is stored in instance variables. For example, a batched list remembers its current page number or the search component stores its last search string and result etc. Since the user interactions share the same component instances (because the resumed computation is always the same) their instance variables may not represent what the user sees when he goes back, or takes several paths in parallel.

To solve this problem, Seaside offers a mechanism to register an object to be backtracked (Session ≫ registerObjectForBacktracking:). After each response sent to the client, Seaside snapshots the registered objects by creating a copy and putting them into a cache. The session stores the registry as a temporary variable in the computation which sends the response to the client and which

receives the next request. Since the continuation (which at this point composes the current request/response loop) restores the context when resuming, the registry is made persistent to the future point when requests, originated from this response, are handled. Before processing the request, the registry restores the registered objects.

This ensures that when processing a request, the values are the same as when the previous response was created. For example the batched list's current page number will be the same as the one shown to the user when a request from this page is processed.

6.2 Transactions

In complex applications it is often the case that we must ensure that the user is prevented from going back a sequence of pages to make modifications. This is applicable in the case of our checkout process, where the user should not be able to change anything after having paid. Controlling the control flow is implemented by the method Component ≫ isolate: which takes a block as argument. It treats the control flow defined in the block as a transaction. The transaction makes sure that the user can move forward and backward as he likes *within* the transaction. But as soon as he completed the transaction, he cannot backtrack anymore.

The following method shows the shop process enhanced with transactions. By surrounding the filling of the cart and its confirmation by an isolate: invocation, we allow backtracking freedom within that part of the application, while protecting a completed order from being changed. Similarly we are making sure that after the confirmation has been displayed, the user is unable to go back and change shipping and payment information.

```
StoreTask ≫ go
    | cart shipping billing creditCard |
    cart := StoreCart new.
    self isolate: [
        self fillCart: cart
        self confirmContentsOfCart: cart ] whileFalse ].
    self isolate: [
        shipping := self getShippingAddress.
        billing := (self useAsBillingAddress: shipping)
            ifFalse: [ self getBillingAddress ]
            ifTrue: [ shipping ].
        creditCard := self getPaymentInfo.
        self ship: cart to: shipping billTo: billing payWith: creditCard ].
    self displayConfirmation.
```

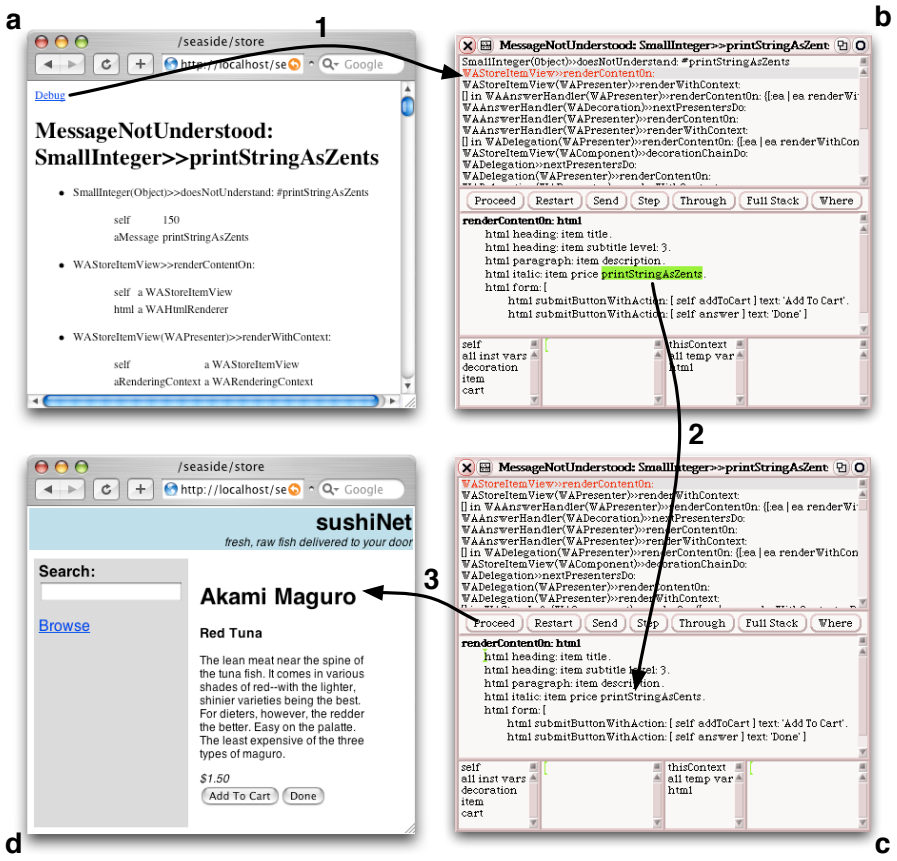

Fig. 5. Debugging with Seaside – a continuous loop fixing a bug without restarting the application.

7 Development Environment

As Seaside is written in Smalltalk it is based on a very powerful, fully object-oriented language and development environment. In addition to being able to use the tools provided by the environment, Seaside integrates them seamlessly with the web. This makes the platform a versatile and productive environment for web application development. We start by looking at the debugging facilities before presenting the Seaside specific tools.

Incremental Programming. 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 and there is neither the need to manually recompile the code nor to restart the server. In many cases this makes it possible to update a system in production on the fly without any outage and without the need to set-up a temporary backup server.

Debugging. Most of today's frameworks do not support debugging of web applications well. Most display the error and the line number in the web

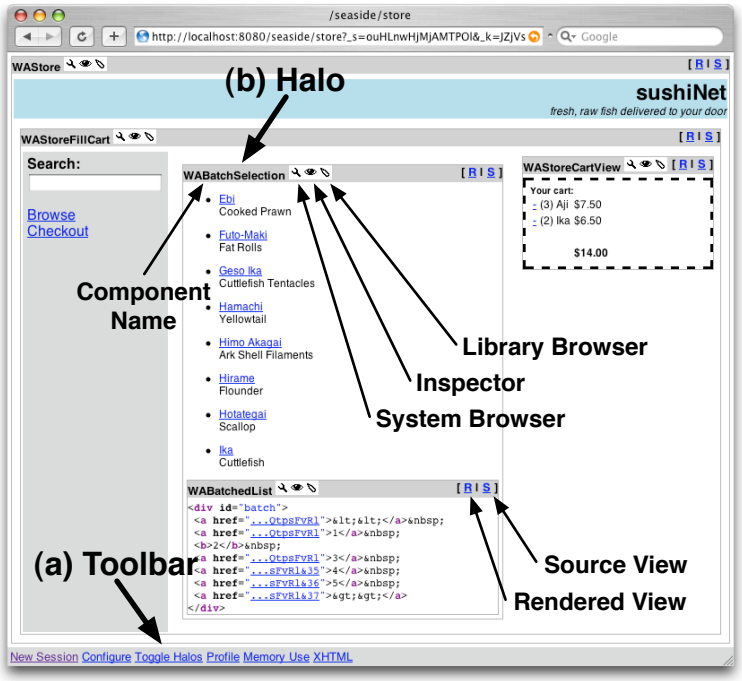

Fig. 6. The Sushi-Shop as seen in Figure 1 in development mode, with toolbar (a) and halos (b) enabled.

browser only, which makes it very inconvenient to find and fix bugs.

Seaside has unique debugging capabilities: When an unhandled exception occurs as seen in Figure 5, a stack trace is shown in the web browser (a) with a link called debug. Clicking this link (1), the developer activates a debugger (b) within the development environment which lets him inspect variables and even modify the code on the fly. In the given example the message printStringAsCents, that is automatically highlighted in the debugger (b), has been spelled wrongly and is fixed (2) by the developer. The debugger now displays the recompiled method (c). During this time, the web browser keeps waiting for the response of the server. When hitting proceed (3), the processing of the request which had caused the error is resumed and the resulting page is finally displayed in the web browser (d).

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 properly and is able to continue the testing session.

Toolbar. A toolbar that is shown at the bottom of the web-application during the development phase (Figure 6) enables the programmer to access additional tools from within the web. Of course, all these tools have been written in Seaside itself:

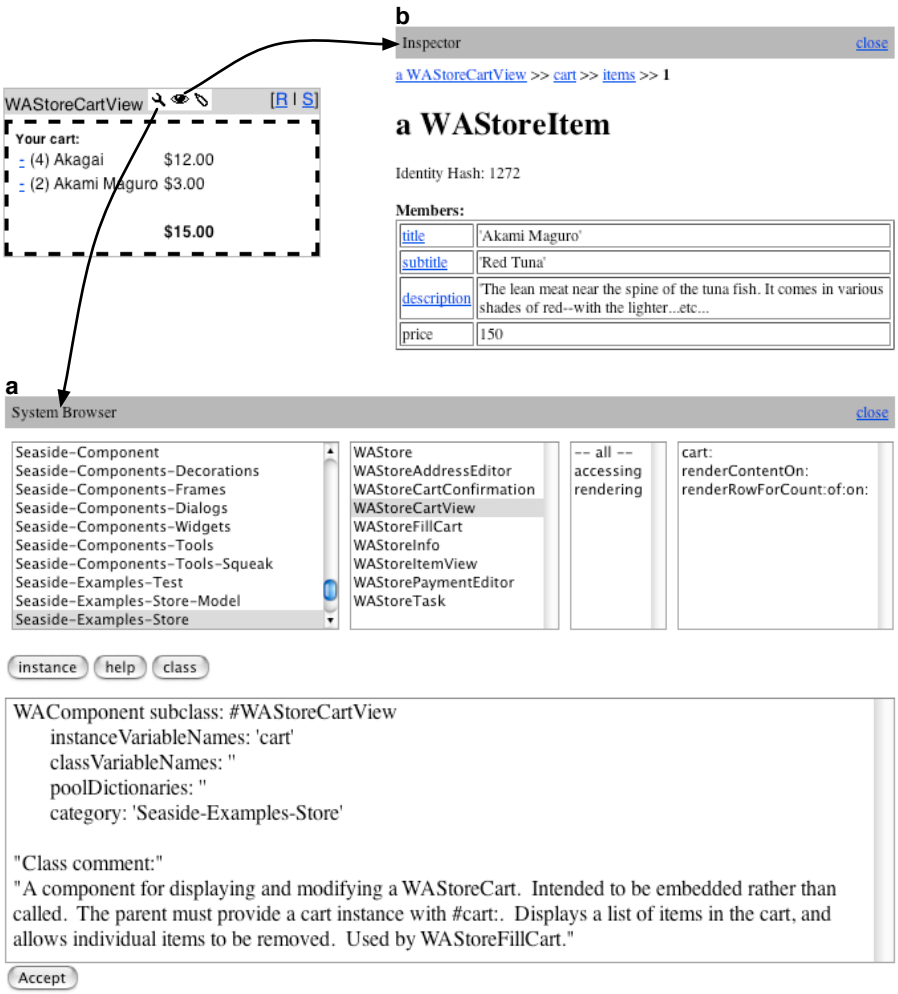

Fig. 7. Browsing (a) and inspecting (b) the CardView component of the running Sushi-Shop.

- New Session starts the application within a new session.
- Configure opens a dialog letting the user configure the settings of the application. This includes properties such as where to start a new session, what should happen in case of an exception or if the development toolbar is displayed or not.
- Toggle Halos shows or hides the halos, which are discussed in detail in the next paragraph.
- Profile shows a detailed report on the computation time that has been consumed while building this page.
- Memory Use displays a detailed report on the amount of memory consumed by this application.
- XHTML starts an external XML validator on this page.

Halos. When enabling the halos, every component gets surrounded by a thin grey line and a header giving the class name of the component and a set of buttons to run tools and to change the viewing mode (Figure 6).

- System Browser opens an editor on the current component and lets the developer modify its class and all the methods from within the web, while the application is still running in the background (Figure 7, a). When closing the browser-view the application immediately runs with the new code without having to restart the session.
- Inspector opens a view on the component, so that the developer can browse the internal structure of this object (Figure 7, b). It presents the names of the instance variables and the current values, whereas the user is able to dive into the referenced objects by clicking on the links. In Figure 7, b, the first item of the items of the cart instance variable of the WAStoreItem instance is displayed.
- Library Browser opens an editor that lets a UI designer tweak the associated CSS-Stylesheets. This makes it very convenient to try out different layouts directly in the web-browser without leaving the running application.
- Source View provides a pretty-printed and syntax-highlighted XHTML view onto the source code of the current component. Like this the developer is able to observe the generation of XHTML while still being able to interact with the application by clicking on its links.

8 Evaluation

Based on continuations, Seaside transparently manages the request/response loop and the handling of the necessary URLs, query strings and hidden form fields. This prevents any name clashes and frees the developer from manually encoding information in the response and decode it later again. This is the foundation to model the control flow explicitly at a higher level of abstraction rather than having to hardcode the next step of flow in each component itself.

Seaside components are responsible for rendering themselves and handling input by action callbacks. Action callbacks are block closures that are bound to user interface elements such as input fields or submit buttons. When a request is processed, the applying action callbacks are evaluated. This enables the processing of form fields or the execution of embedded control flow directly in the component in a natural way.

In Seaside, each component can run its own control flow independently of the others. This makes possible to compose complex applications out of small and reusable components without having the problem of composing the individual control flows. This component composition makes possible to have multiple control flows within the same page naturally. In addition in Seaside, a transaction allows one to specify how to group a control flow part and to ensure that the user cannot go back into it after he left it. This is a powerful solution to define security independently form the involved components. Again, this is

crucial for reusability.

Seaside's control flow with its unique call/answer semantics offers passing around business objects between components. In contrast to other frameworks this avoids the need to pass state from one component/page to the next over the client. To synchronize state with the current page displayed in the user's browsers, Seaside offers backtracking of objects. This makes the back button to be a fully supported navigation facility of the application.

On the negative side, in the current version all the continuations have to be kept in memory which consumes resources. Continuations could be stored in database but this solution implies to be able to serialize continuations which is a non trivial task [15] [2].

9 Related Work

The idea of modeling sessions as a continuous piece of code has been popping up independently in multiple places in the past [14] [12] [15].

DrScheme. The *Scheme Web Servlets* library included with DrScheme [11] was one of the first frameworks to support continuation-based web development. However it does not provide a simple solution to provide multiple control flows feasible. Furthermore DrScheme does not provide a collection of components that can be composed easily and reused with different applications.

CocoonFlow. Most of todays mainstream programming languages unfortunately do not support continuations, therefore the authors of CocoonFlow [27] decided to enhance their JavaScript runtime to support this concept. However CocoonFlow does not provide such a high level abstraction over the HTTP protocol as Seaside does, the function sendPageAndWait has to be called to suspend the execution after the page has be generated and sent to the client. This mechanism does not allow to have multiple flows on the same page easily.

RIFE. Java Servlets/JSP [19] is lacking the possibility to model control flow in a clever way. RIFE [10] provides a declarative way to define application flow based on state machines. For simple Web applications, this model works. However, RIFE is facing a well-known problem with state machines: the number of states and transitions grow fast and it becomes hard to understand what is happening in the application. Multiple control flows are not supported.

Imposter. Python is lacking support for continuations as well. Imposter [28] provides an abstraction over the session handling by saving the whole internal state of the applications between two requests. However, as there is only one

snapshot stored in the memory, using the back button is not supported and the state of the application and the web browser window cannot be synchronized properly.

WebObjects. Apple Web Objects [5] provides a component-based framework to reuse and compose components which offers solution to the back button problems, however it is lacking the possibility to describe a flow of pages as a continuous piece of code and multiple control flow.

ASP.NET. Microsoft ASP.NET [6] is a web application framework running on the .NET platform. Its web controls are comparable to Seaside's components. However, web controls are not capable of modeling a continuous flow of user interaction in one piece of code. Moving to another page is still done in a goto-like manner by redirecting the user to the next page. Reusability and flexibility suffers and the problems of the back button and cloning of windows have to be addressed by the developer by implementing workarounds.

Struts. Struts [7] purpose is to bring the MVC pattern to the J2EE platform. The Struts architecture acts as a wrapper for Java applications and divides its code into a Model, View and Controller. Although Struts manages to add a layer of abstraction to model control flows, the costs are high. Compared to Seaside a form validation requires rewriting several parts of the application: the form class, an XML file and the JSP code to display validation error messages. Adding validation to an existing application even requires changing the inheritance trees. In Seaside a validation can be done by adding a decorator around the component.

10 Conclusion

Building web applications raised specific problems due to the disconnected flow between clients and servers. Application control flow and state management made web application development difficult and cumbersome. Programmers are often forced to program using goto-style. The solutions can be roughly classified into two categories: the ones that use objects to model pages and applications, and the ones that use the power of continuations to provide an advanced control flow. The first ones to not address well the problem related to control flow. While more advanced the second approaches do not support well the composition of independent control flows.

Seaside, the framework presented in this article, combines both approaches: a web application is composed of components, each having its own control flow and been able to pass the control to other component. Applications are then a composition of components whose control flow is combined and can run

independently. This enables Seaside to offer a better reuse and composition of predefined components. With Seaside, the programmer is able to write business application logic at a high-level of abstractions.

Acknowledgment. We thank Avi Bryant for his feedback and encouragements, Shriram Krishnamurthi for the discussions on continuations and Orla Greevy and Alexandre Bergel for their feedback.

References

[1] CGI, The Common Gateway Interface, http://hoohoo.ncsa.uiuc.edu/cgi/.

[2] C. Queinnec, Continuations and web servers, Higher-Order and Symbolic Computation: an International Journal (2004) 1–16.

[3] E. W. Dijkstra, Go To statement considered harmful, Comm. ACM 11 (3) (1968) 147–148, letter to the Editor.

[4] O.-J. Dahl, E. W. Dijkstra, C. Hoare, Structured Programming, Academic Press, 1972.

[5] WebObjects, http://www.apple.com/webobjects/.

[6] ASP.NET, http://www.microsoft.com/net/.

[7] The Apache Struts web application framework, http://jakarta.apache.org/struts/.

[8] JWIG, Java Extensions for High-Level Web Service Development, http://www.brics.dk/JWIG/.

[9] A. S. Christensen, A. Moller, M. I. Schwartzbach, Extending java for highlevel web service construction, ACM Transaction on Programming Languages and Systems 25 (6) (2003) 814–875.

[10] RIFE, https://rife.dev.java.net.

[11] DrScheme, http://www.drscheme.org.

[12] J. Hughes, Generalising monads to arrows, Science of Computer Programming 37 (2000) 67–111.

[13] P. Graham, Beating the averages, http://www.paulgraham.com/avg.html.

[14] C. Queinnec, The influence of browsers on evaluators or, continuations to program web servers, in: ACM SIGPLAN International Conference on Functional Programming, 2000, pp. 23–33.

[15] P. Graunke, S. Krishnamurthi, S. Van Der Hoeven, M. Felleisen, Programming the Web with high-level programming languages, in: Proceedings of ESOP 2001, Vol. 2028 of Lecture Notes in Computer Science, 2001, pp. 122–136.

[16] Seaside: Squeak enterprise aubergines server, http://www.beta4.com/seaside2/.

[17] Squeak home page, http://www.squeak.org/.

[18] D. Coward, Java servlet specification version 2.3, http://java.sun.com/products/servlet/ (2000).

[19] Java Server Pages, http://java.sun.com/products/jsp/.

[20] PHP: Hypertext Preprocessor, http://www.php.net/.

[21] ASP, Microsoft Active Server Pages, http://msdn.microsoft.com/nhp/?contentid=28000522.

[22] Zope, http://www.zope.org.

[23] P. Graunke, R. B. Findler, S. K. and Matthias Felleisen, Automatically restructuring programs for the web, in: International Conference on Automated Software Engineering, 2001.

[24] J. Matthews, R. B. Findler, P. Graunke, S. Krishnamurthi, M. Felleisen, Automatically restructuring programs for the web, Automated Software Engineering: An International Journal .

[25] G. E. Krasner, S. T. Pope, A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, Journal of Object-Oriented Programming 1 (3) (1988) 26–49.

[26] N. Kurt, Using lisp as a markup language the LAML approach, european Lisp User Group Meeting (1999).

[27] Apache Cocoon, The Apache Cocoon Project, http://cocoon.apache.org/.

[28] Imposter, http://csoki.ki.iif.hu/∼vitezg/impostor/.