Programming with Seaside

Alexandre Bergel
bergel@iam.unibe.ch

SCG-IAM
University of Bern

Part I:
Seaside in a Nutshell

Alexandre Bergel

Outline

1. What is Seaside?
2. Starting Seaside
3. Create new Seaside Component
4. Creating GUI
5. Using CSS
6. Interaction Between Components

Alexandre Bergel

Introduction to Seaside

- Application server Framework
- Useful to generate dynamic web page

- Web server application for Squeak (used in this presentation) and VisualWorks.
- Works on the top of a webserver (Comanche, Swazoo).
- Provides high-level API to handle navigation between pages (links) and GUI.

Alexandre Bergel

Some of the Seaside Features

- Sessions as continuous piece of code
- XHTML/CSS building
- Callback based event-model
- Composition and Reuse
- Development tools
- Interactive debugging
- Multiple control flow

Alexandre Bergel

Starting Seaside

- Start the server with:
  WAKom startOn: 9090
- Go to to access the counter component:
  http://localhost:9090/seaside/counter
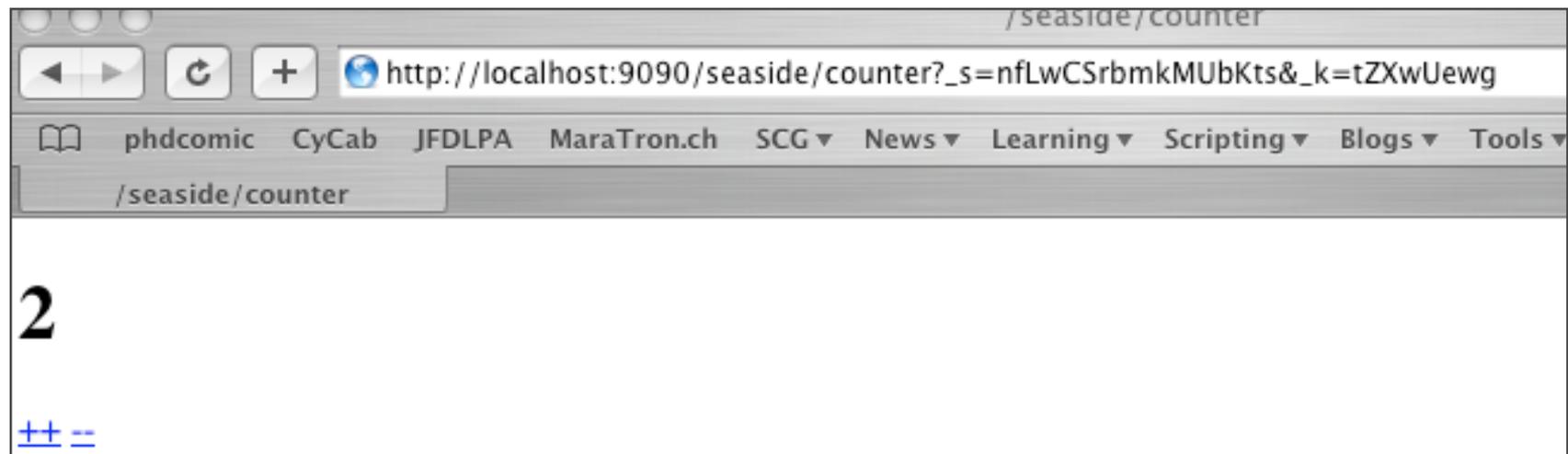
Component Responsibilities

- It is a subclass of WAComponent
- It contains a State modeled as instance variables
- The flow is defined by methods
- Rendering (high-level API that generate XHTML)
- Style (CSS)

Alexandre Bergel

Counter Example

WAComponent
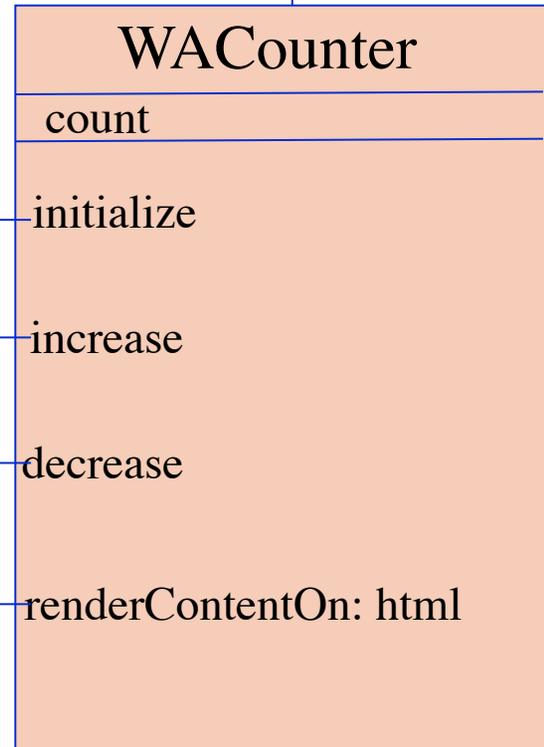
WACounter

count

self session registerObjectForBacktracking: self.
count := 0
— initialize

count := count + 1
— increase

count := count - 1
— decrease

html heading: count.
html anchorWithAction: [self increase] text: '++'.
html space.
html anchorWithAction: [self decrease] text: '--'.
— renderContentOn: html

WACounter class>>initialize

    self registerAsApplication: 'counter'

Alexandre Bergel

Creating new Component

- Designing a small application to memorize words in a foreign language.
- Display a score to show the progress.
- 2 ways of using:
  - Adding a new word in the database
  - Entering a translation

Alexandre Bergel

Creating new Component
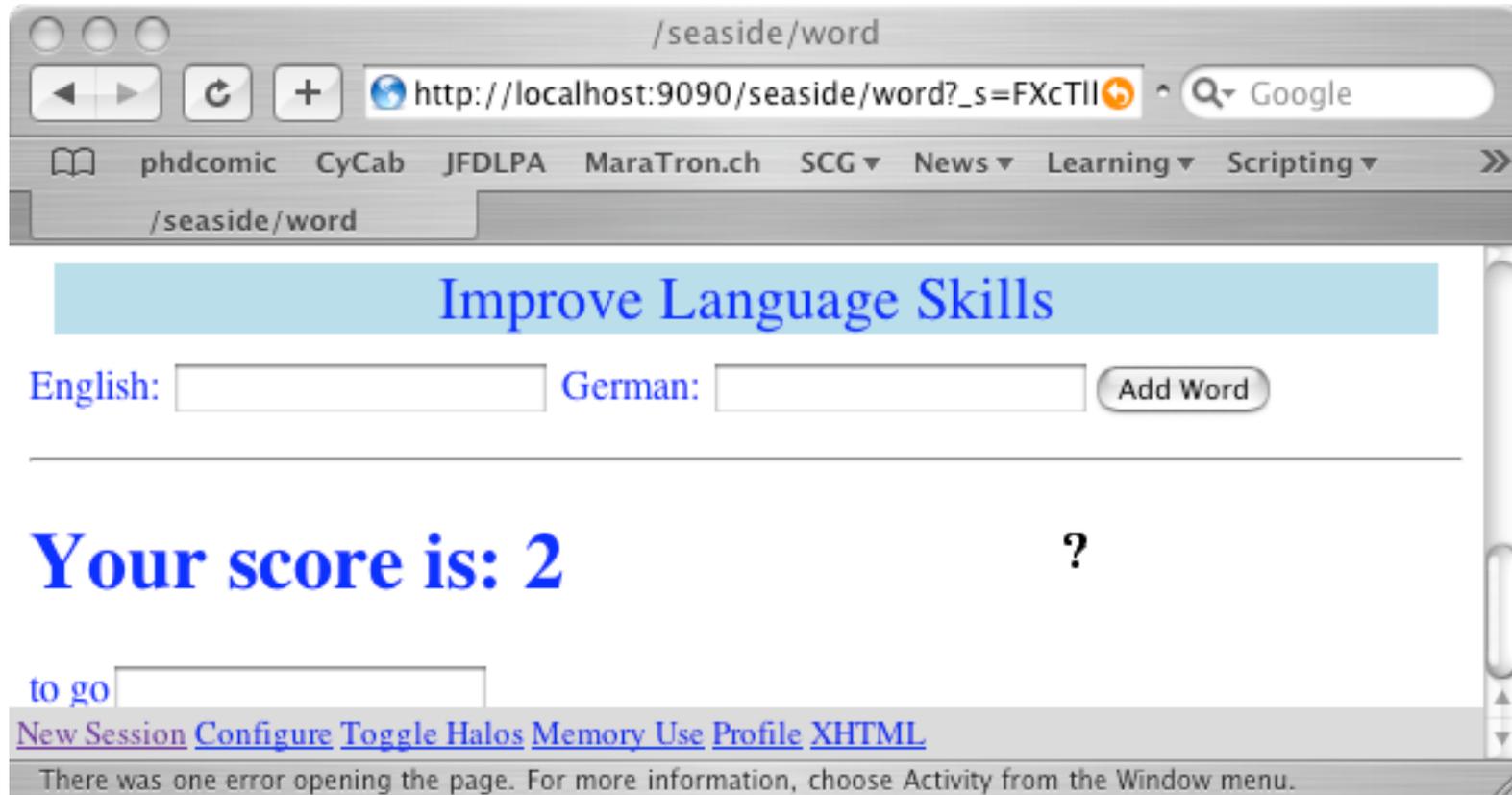
Component Definition

- Definition of the main class:
  WAComponent subclass: #**Learner**
    instanceVariableNames: **'words germanWord englishWord chosenEntry score'**
    classVariableNames: ''
    poolDictionaries: ''
    category: '**WordLearning**'

Variables Initialization

- List of entered words:
  ```
  Learner>>words
      words ifNil: [words := OrderedCollection new].
      ^ words
  ```
- Score (increased when an entered word is correct):
  ```
  Learner>>score
      score ifNil: [score := 0].
      ^ score
  ```
- Choose a word:
  ```
  Learner>>chooseEntry
      chosenEntry := self words atRandom
  ```

Helper Methods

- Could we ask for a word?
  Learner>>readyToGuessWord

  ^ self words notEmpty
- Increasing the score:
  Learner>> increaseScore

  score := self score + 1

Managing the Back Button

- Need to keep the history of the objects, in case of pressing the back button on the web browser

```
Learner>>initialize
    super initialize.
    self session registerObjectForBacktracking: self.
```

- A trace of the lifetime is kept. When the back button is pressed, state previously recorded is restored.

Alexandre Bergel

Registration of the Application

· Application registration:
Learner class>>initialize
self registerAsApplication: 'word'
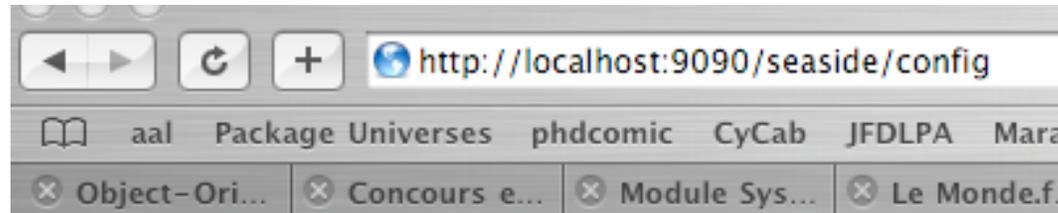
Rendering (1/2)

- Learner>>renderContentOn: html
  html heading: 'Improve your Language Skills'.
  html form: [

    html text: 'English: '.
    html textInputWithCallback: [:w| englishWord := w].
    html text: ' German: '.
    html textInputWithCallback: [:w| germanWord := w].
    html submitButtonWithAction:
        [self words add: (Array with: englishWord with: germanWord)]
            text: 'Add Word'.

  ].
  ...

Improve your Language Skills

English: [   ] German: [   ] [Add Word]

Alexandre Bergel

Rendering (2/2)

- ...

```
html horizontalRule.
self readyToChooseWord ifTrue: [
 html heading: 'Your score is: ', self score asString.
 html form: [ |chosenEntry|

   chosenEntry := self chooseEntry.

   html text: (chosenEntry first).
   html textInputWithCallback:
       [:w| (w = chosenEntry second) ifTrue: [self increaseScore]].
       ]]
```

Your score is: 4

house

Creating GUI (1/2)

- Displaying simple text:
  html text: 'My Text'
- Using different size:
  html heading: aBlockOrText level: level
  html heading: aBlockOrString
- Link with action:
  html anchorWithAction: aBlock text: aString
- TextField without any button:
  html form: [... html textInputWithCallback: aBlock ...]

Alexandre Bergel

Creating GUI (2/2)

- Using a form:

```
html form: [
    html textInputWithCallback: aBlock.

    ...

    html submitButtonWithAction: aBlock text: aString]
```

- Look at the class **WAHtmlRenderer** and **WAAbstractHtmlBuilder**

CSS: to give a better look

- Use divNamed: aString with: aBlockOrObject
    html divNamed: 'title' with: [
        html text: 'Improve Language Skills'
    ].
- Or
    html divNamed: 'title' with: 'Improve Language Skills'

CSS: defining the style

- Define a method named **style** on the seaside component:

```
WordLearningComponent>>style
 ^ '#title {
 background-color: lightblue;
 margin: 10px;
 text-align: center;
 color: blue;
 font-size: 18pt;
 margin-top: 400px}
body {
 background-image: url("http://www.iam.unibe.ch/~bergel/
 catsEye_hst_full.jpg");
 background-repeat: no-repeat;
 background-position: top center;
 color: blue;}'
```

CSS: more info

- Supported by many web browsers

- Where to get more information:
  http://www.w3schools.com/css

- ZenGarden:
  http://www.csszengarden.com/

Alexandre Bergel

call: / answer:

```
A>>m1
   x := self call: B
   x printString
```

```
B>>m2
   self answer: 69
```

```
A>>m1
   x := self call: B
   x printString
   -> 69
```

code   components in browser

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.

Alexandre Bergel

call: / answer:

- To transfer control to another component, WAComponent provides the special method #call:. This method takes a component as a parameter, and will immediately begin that component's response loop, displaying it to the user.

- If a called component provides an argument to #answer:, that argument will be returned from #call:. In other words, calling a component can yield a result.

Example: Sushi Shop Online

search component

list component

cart view component

batch component

sushiNet

http://localhost:8080/seaside/store?_s=RakTLY Google

sushiNet
fresh, raw fish delivered to your door

Search:

Browse
Checkout

Your cart:
- (3) Aji $7.50
- (2) Ika $6.50

$14.00

- Ebi
  Cooked Prawn

- Futo-Maki
  Fat Rolls

- Geso Ika
  Cuttlefish Tentacles

- Hamachi
  Yellowtail

- Himo Akagai
  Ark Shell Filaments

- Hirame
  Flounder

- Hotategai
  Scallop

- Ika
  Cuttlefish

<< 1 2 3 4 5 >>

Alexandre Bergel

Logical Flow

A flow chart diagram with the following boxes and transitions:

- **Fill cart** —buy→ **Confirm contents. Checkout?**
- **Confirm contents. Checkout?** —no→ back to **Fill cart**
- **Confirm contents. Checkout?** —yes→ **Shipping address**
- **Shipping address** —ok→ **Use shipping as billing address?**
- **Use shipping as billing address?** —no→ **Billing address**
- **Use shipping as billing address?** —yes→ **Payment infos**
- **Billing address** —ok→ **Payment infos**
- **Payment infos** —ok→ **Confirmation**
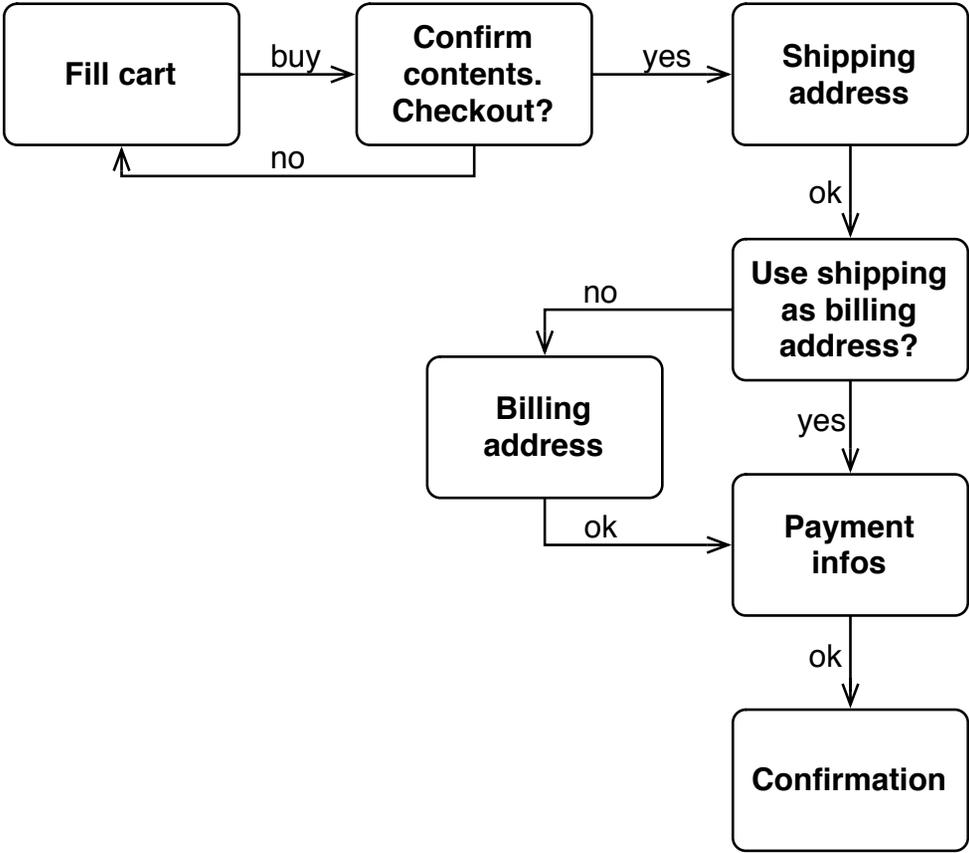
XHTML generation

- XHTML code is generated programmatically:

```
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: task
```

Control Flow

```
WAStoreTask>>go
 | shipping billing creditCard |
 cart := WAStoreCart new.
 self isolate:
    [[self fillCart. self confirmContentsOfCart] whileFalse].
 self isolate:
    [shipping := self getShippingAddress.
     billing := (self useAsBillingAddress: shipping)
                            ifFalse: [self getBillingAddress]
                            ifTrue: [shipping].
         creditCard := self getPaymentInfo.
         self shipTo: shipping billTo: billing payWith:
 creditCard].
  self displayConfirmation.
```

Alexandre Bergel

Control Flow

- To fill in the cart:
  WAStore>>fillCart

    self call: (WAStoreFillCart new cart: cart)
- To confirm contents of cart:
  WAStoreTask>>confirmContentsOfCart

      ^ self call:

          ((WAStoreCartConfirmation new cart: cart)

              addMessage: 'Please verify your order:')
- Payment:
  WAStore>>getPaymentInfo

    ^ self call:

      ((WAStorePaymentEditor new

          validateWith: [:p | p validate])

      addMessage: 'Please enter your payment information:')

Control Flow

- answer returns the component itself
   WAStoreFillCart>>checkout
      self answer

Some Guidelines

- Tasks are used to embed the logical flow of an application within the go method, whereas
- The rendering is in charge of components.
- Hence, the entry point of an application should be a task's go method

Seaside

- Used in industries
- More info on:
  http://www.beta4.com/seaside2
- Seaside's fathers: Avi Bryant and Julian Fitzell
- Mailing list:
  http://lists.squeakfoundation.org/listinfo/seaside

Part II: Developing Web-based Applications

Alexandre Bergel

Outline

1. What is a Web-based Application?
2. Issues when Directly Dealing with HTML
3. Example: Sushi Shop Online
4. Seaside Approach
5. Manipulating Non-Linear Control Flow
6. Development Tools

Alexandre Bergel

What is a Web-based Application?

- A collection of functions that take HTTP requests as input and produce HTTP responses as output.
- Logical part centralized

Directly Manipulating HTML

- Stateless connection toward the server. State has to be passed around for each connection.
- ASP, PHP

What is a Web-based Application?

flight.html

address.html

GET
flight.html

<a href=
"address.html**?cart=...**"

GET
address.html**?cart=...**

<a href=
"payment.html**?
cart=...&address=...**"

...

User: Web browser

Alexandre Bergel
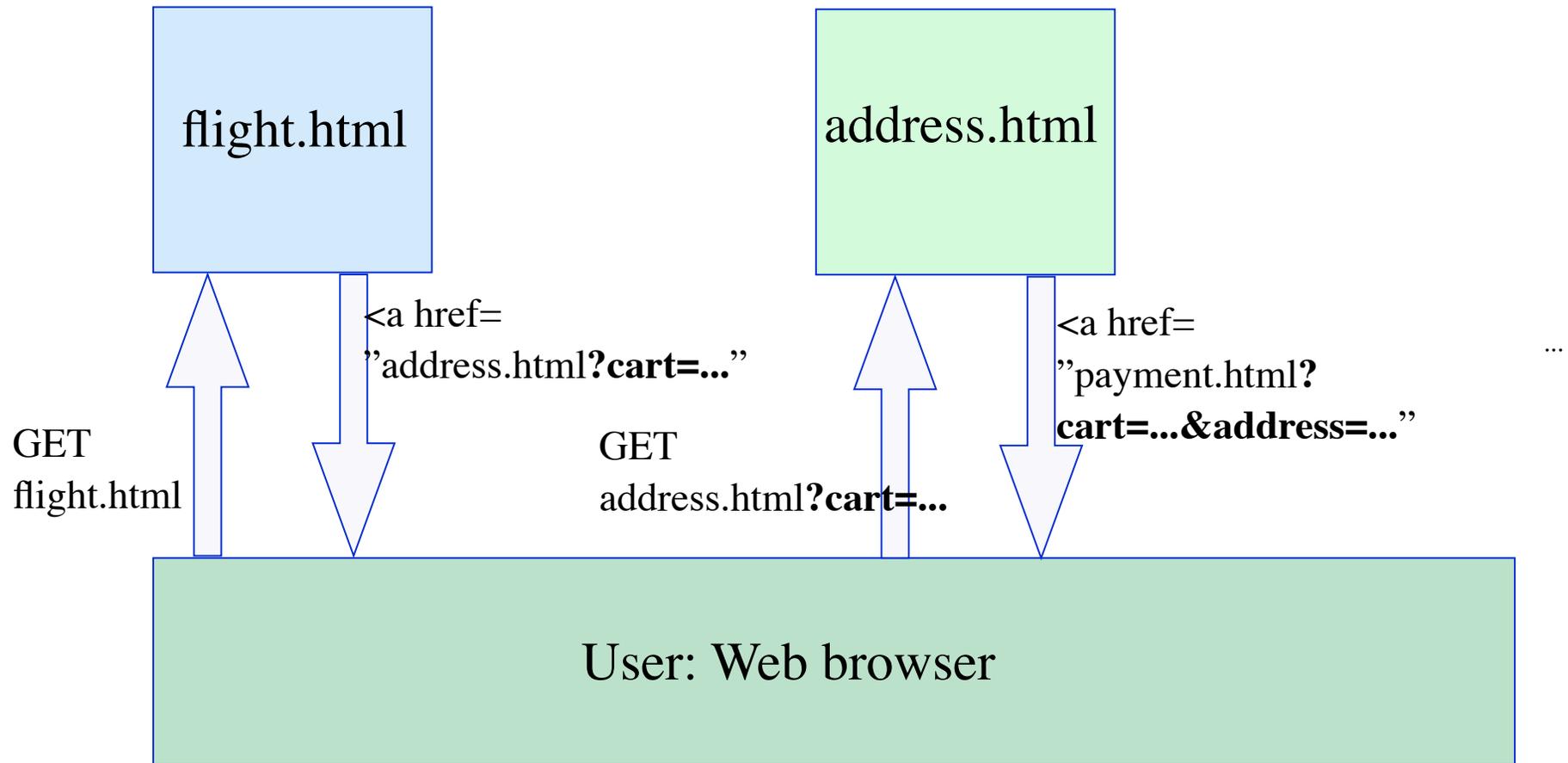
Directly Manipulating HTML

- Applications are difficult to maintain:
    - Adding, renaming, removing some state is difficult
    - Flow execution scattered in several files
    - Poor management of the bandwidth: state has to be passed for each action!

Common Issues with Classical Framework

- Applications are often tedious to use:
  - Do not use the back button!
  - Do not duplicate the windows!
  - "Press OK only once!!!"
  - "Do you want to resend the form?"
  - Cookies manipulations

Seaside Approach

- Each session has one unique ID kept over its life time:
  - Users (web browsers windows) are distinguished
- Each action has one ID unique over the session:
  - In the lifetime of a session, an action is unique ("press OK only once")

Non-Linear Control Flow

· The control flow of an application can always be modified by the user when pressing the back button or by opening a new browser on the same url.

Backtracking State

- With seaside, an object can be backtracked using the method:
  WASession>>registerObjectForBacktracking: anObject

- After each response sent to the client, Seaside snapshots the registered objects by creating a copy and putting them into a cache.

- Pressing the back button on the browser restores the state of the object which is in sync of the display.

Alexandre Bergel

Transaction

- In complex applications it is often the case that we must ensure that the user is prevented from going back over a sequence of pages to make modifications.

- Controlling the control flow is implemented by the method:
  Component>>isolate: aBlock

- It treats the control flow defined in the block as a transaction. It makes sure that the user can move forward and backward *within* the transaction. Once completed, the user cannot go back anymore.

Alexandre Bergel

Debugging with Seaside

- When debugged, an application does not need to be restarted or manually recompiled

Debugging

Alexandre Be

Toolbar

(b) Halo

Component Name

Library Browser

Inspector

System Browser

Source View

Rendered View
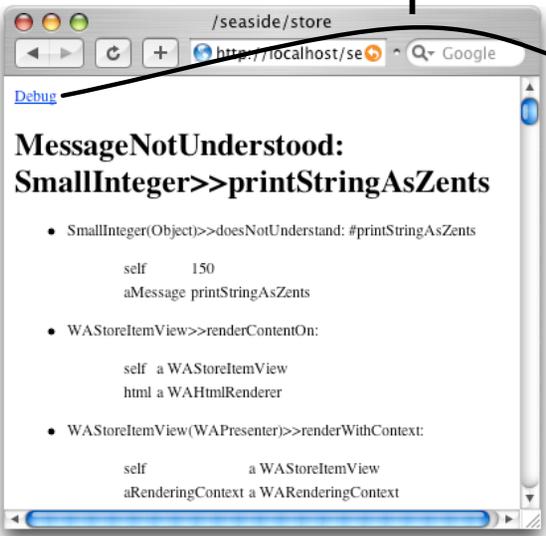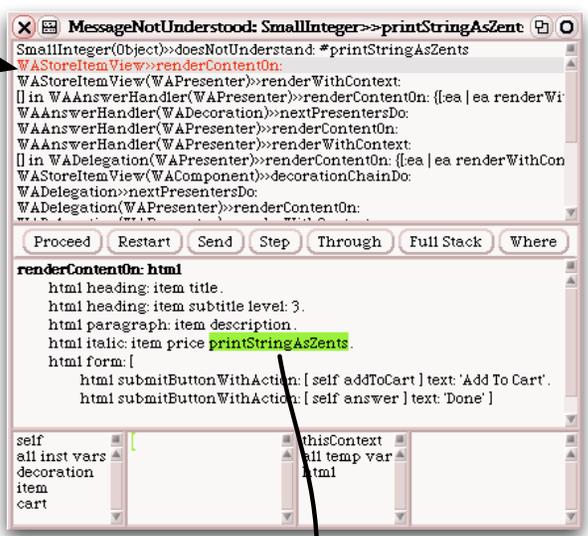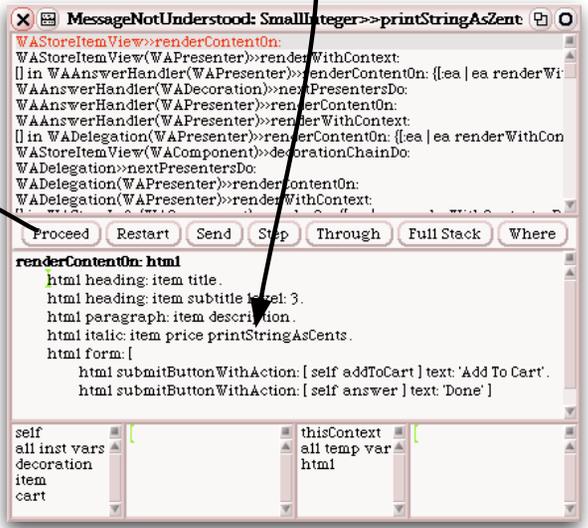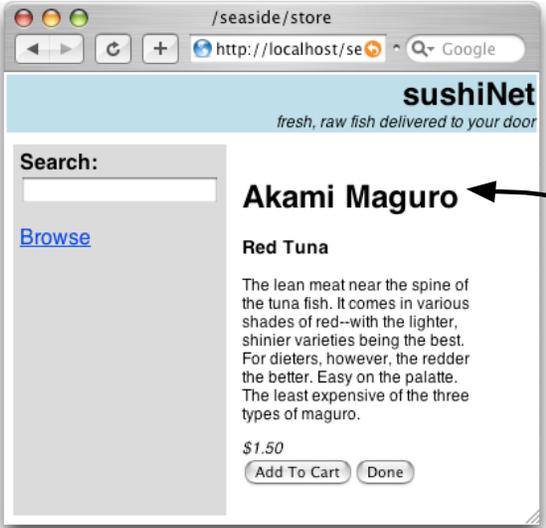
(a) Toolbar

Alexandre Bergel
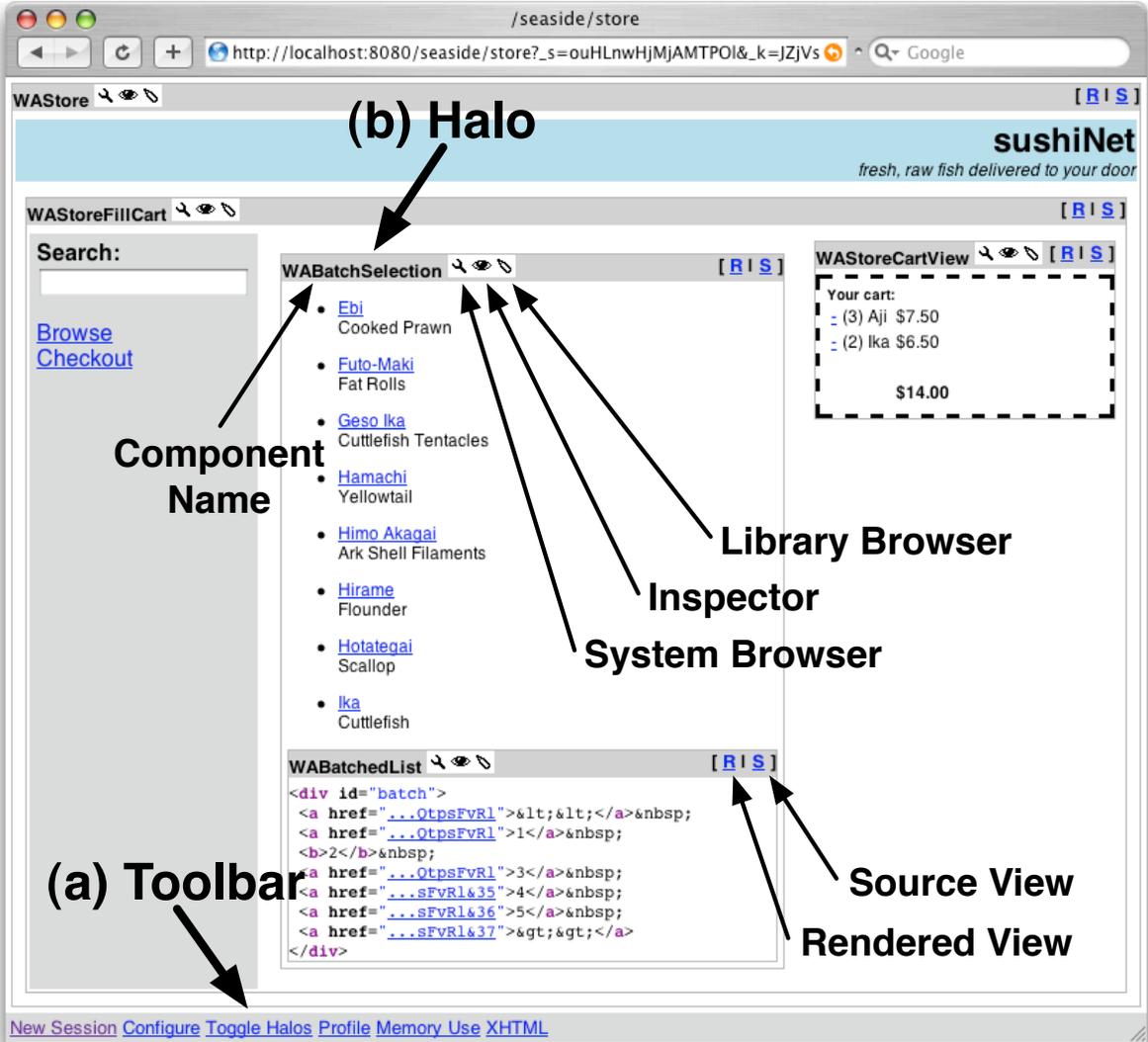
Toolbar

- A toolbar is shown at the bottom of the web-application during the development phase.
- It allows one to access some tools:
  - **New Session** restart the application
  - **Configure** opens a dialog letting the user configure some settings
  - **Toggle Halos** shows or hides the halos (explained later)
  - **Profile** shows a detailed report on the computation time used to render the page
  - **Memory Use** display a detailed report on the memory consumption
  - **XHTML** start an external XML validator on this page

Halos

- When enabling the halos, every component gets surronded 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.
    - **System Browser** opens an editor on the current component.
    - **Inspector** opens a view on the current component.
    - **Library Browser** opens an editor that lets a UI designer tweak the associated CSS-Stylesheets.
    - **Source View** provides a pretty-printed and syntax-highlighted XHTML view onto the source code .

Benefits with Seaside

- With PHP: Control flow scattered into files (flight.html, address.html, ...)
- With Seaside: Control flow = method calls (getFlight, getAddress, ...)
- Bandwidth saved: session state is only stored on the server side.
- It facilitates reusability!