5. Seaside
Roadmap

> Introduction
  – Web applications / Overview
  – Installation
> Control Flow
> Components
> Composition

Original lecture notes by Lukas Renggli
Roadmap

> Introduction
  - Web applications / Overview
  - Installation
> Control Flow
> Components
> Composition
Introduction: Web Applications

Example: Adding two numbers

First Number

1

Second Number

2

The result is 3
What is going on?

```
<form action="second.html">
   <input type="text" name="value1">
   <input type="submit" name="OK">
</form>

<form action="result.html">
   <input type="hidden" name="value1" value="<% value1 %>">
</form>

<p>
   <% value1 + value2 %>
</p>
```
Control Flow: HTTP request-response

User: Web browser

GET first.html

first.html

<a href="second.html?number1=..."

GET second.html?number1=...

second.html

<a href="result.html?number1=...&number2=..."
Something is wrong…

> Control-flow quite arcane
  – Remember GOTO?
  – We do not care about HTTP!

> How to debug that?

> And what about
  – Back button?
  – Copy of URL (second browser)?
What we want

> Why not this?

go

| number1 number2 |

number1 := self request: ‘First Number’.
number2 := self request: ‘Second Number’.

self inform: ‘The result is ‘,
(number1 + number2) asString
Seaside: Features

- Sessions as continuous piece of code
- XHTML/CSS building
- Callback based event model
- Composition and reuse
- Debugging and Development tools
XHTML Building

```html
div id: 'title'; with: 'Title'
div id: 'list'; with: [
    span class: 'item'; with: 'Item 1'.
    span class: 'item'; with: 'Item 2'.
]
```

```html
<div id="title">Title</div>
<div if="list">
    <span class="item">Item 1</span>
    <span class="item">Item 2</span>
</div>
```
CSS Zengarden: http://csszengarden.com
Callback Event Model

Example3>>renderContentOn: html
html form: [
html submitButton
    callback: [ self inform: 'Hello' ];
text: 'Say Hello' ]

....
<form action="/seaside/example2" method="post">
<input type="hidden" name="_s" value="JBBTXBnPaTLOjcjI" class="hidden"/>
<input type="hidden" name="_k" value="FFQrpnBg" class="hidden" />
<input type="submit" name="1" value="Say Hello" class="submit" />
</form>
....

Composition + Reuse

Example: Multicounter

More later!
Example: SqueakSource
Example: DabbleDB

Dabbledb.com
Example: CMSBox

cmsbox.ch
Roadmap

> Introduction
  – Web applications / Overview
  – Installation
> Control Flow
> Components
> Composition
Installing Seaside

> Easy: Get from
  – http://www.iam.unibe.ch/~scg/Teaching/Smalltalk/Exercises/05Seaside/
  – For the config tool: user: admin, passwd: seaside

> Or install by hand
  – Install Seaside 2.8a via SqueakMap
  – Update using Monticello
  – WAKom startOn: 8080
  – Point browser to http://localhost:8080/seaside
http://localhost:8080/seaside
Roadmap

> Introduction
  – Web applications / Overview
  – Installation
> Control Flow
> Components
> Composition
2. Control Flow

> Defining control flow
> Convenience methods
> Call / Answer
> Transactions
Defining Flow

> Create a subclass of `WATask`
  > Implement the method `#go`
  > Split the method `#go` into smaller parts to ensure readability

> Tasks are a special kind of component
  > No visual representation
  > Define a logical flow (`#go`)
  > Call other components for output
Convenience Methods

- #inform: aString
- #confirm: aString
- #request: aString
- #request:label:default:
Call and Answer

> #call: aComponent
   — Transfer control to aComponent

> #answer: anObject
   — anObject will be returned from #call:
   — Receiving component will be removed
Call and Answer

Client

Server

A

B

A

A>>go
x := self call: B
x asString.

B>>go
...
self answer: 77.

A>>go
x := self call: B.
x asString.
-> 77
Transactions

> Sometimes it is required to prevent the user from going back within a flow
> Calling `#isolate`: treats the flow defined in the block as a transaction
> Users are able to move back and forth within the transaction, but once completed, they cannot go back anymore
Example for #isolate:

```ruby
self isolate: [
  self doShopping.
  self collectPaymentInfo].
Self showConfirmation.
```
Roadmap

- Introduction
  - Web applications / Overview
  - Installation
- Control Flow
- Components
- Composition
3. Components

> Rendering
  – XHTML
  – CSS

> Callbacks
  – Anchor
  – Form

> Customization
Components

> Components are the Views (and Controllers) of Seaside applications.
> Components keep their state (model and state of user-interface) in instance-variables.
> Components define the visual appearance and handle user interactions
Building Components

- Components are created by subclassing WACComponent
- Add instance-variables to hold your model and user-interface state
- Put view related methods in a category called rendering
- Put controller related methods into method categories named accessing, actions, private
Rendering

> XHTML is built programmatically.
> This process is called rendering.

> Create a method called `renderContentOn`:

```plaintext
SomeComponent>>renderContentOn: html
.html text: ‘Hello World’
```
Text Rendering

> Render a string:

```
html text: ‘My Text’
```

> Render an un-escaped string:

```
html html: ‘<foo>Zork</foo>’
```

> Render any object (using double dispatch):

```
html render: 1
```
Canvas and Brushes

> html parameter is instance of WARenderingCanvas
  - Basic html output
  - Render logic

> Canvas provides brushes
  - For rendering html tags
Basic Brushes

> Render a new line `<br />`:  
```html
html break.
```

> Render a horizontal Rule `<hr />`:  
```html
html horizontalRule.
```

> Render a non-breaking space `&nbsp;`:  
```html
html space.
```
Using Brushes

1. Ask the canvas for a div brush

```html
html div.
```

2. Configure the brush, e.g. set attributes

```html
html div class: 'beautiful'.
```

3. Render the contents of the tag-brush:

```html
html div
class: 'beautiful';
with: 'Hello World'.
```
Painting with Brushes

Seaside

```html
<html><div class="beautiful">Hello World</div></html>
```

XHTML

```xml
<div class="beautiful">Hello World</div>
```

Lukas Renggli and Marcus Denker
Nesting Brushes

> Render a text in **bold**:

```html
html strong with: ‘My Text’.
```

> Render a text in *italic*

```html
html emphasis with: ‘My Text’.
```

> Render a text in **bold** and *italic*:

```html
html strong with: [
    html emphasis with: ‘My Text’].
```
Nesting Brushes

> To nest brushes use the message #with:.
> Always send #with: as the last message in the configuration cascade.
> The argument of #with: is rendered using double-dispatch, therefore any object can be passed as an argument.
> To nest tags, pass a block that renders the elements to nest.
Nesting Brushes

> Render nested divs:

```html
div id: 'frame'; with: [
    div id: 'contents'; with: ... 
    div id: 'sidebar'; with: ... ].
```

> Render a list:

```html
orderedList with: [ 
    listItem with: ... 
    listItem with: ... ].
```
Don’t change the state of the application while rendering, unless you have a really good reason to do so.

> Rendering is a *read-only* phase.

> Don’t put all your rendering code into a single method. Split it into small parts and choose a method name following the pattern `#render*On:`
Rendering Pitfalls II

> Rendering is a read-only phase.
  > Don’t send `#renderContentOn:` from your own code, use `#render:` instead.
  > Don’t send `#call:` and `#answer:` while rendering

> Always use `#with:` as the last message in the configuration cascade of your brush
Anchor Callback

> Ask the rendering canvas for an anchor and configure it with a callback-block:

```html
html anchor
  callback: [self someAction]
  with: ‘Some Action’.
```

> The callback-block is cached and will be executed later.
Anchor Example

WACounter>>renderContentOn: html

    html heading
    level: 1;
    with: self count.

    html anchor
    callback: [self increase];
    with: ‘++’.

    html space.

    html anchor
    callback: [self decrease];
    with: ‘--’.
Forms

> Render a form around your form elements:

```html
html form: [ ... ]
```

> Put the Form elements inside the form:

```html
html form: [
  html textInput
  value: text;
  callback: [:value | text := value].
  html submitButton ].
```
More Brushes with Callbacks..

- Text Input / Text Area
- Submit Button
- Check-Box
- Radio Group
- Select List
- File-Upload

Have a look at the tests!
Register new Component / Task

> Create method `canBeRoot` returning true on class side

> Register using Seaside configuration interface.

> Or call `registerAsApplication:` in the class-side `#initialize`
Roadmap

> Introduction
  – Web applications / Overview
  – Installation
> Control Flow
> Components
> Composition
4. Compositon

> Backtracking

> Subcomponents

> Widgets
Backtracking State

> Seaside does not backtrack state by default.
> Often it is not obvious whether an object should be backtracked or not. Mostly this has to be decided by the developer on a per-object basis.
> Any object can be declared to be backtracked.
Shopping Cart Problem

> Online Bookstore (without backtracking)
  — When using the back-button, usually the items should not be removed from the cart; just resume browsing from the old location.

> Flight Reservation System (with backtracking)
  — When using the back-button, usually you want to check other flights, this means the selected flight should be removed.
Register Object

> Implement method `#states` that returns an Array that contains your object.

> This will backtrack the `instance-variables` of the objects, not the objects themselves.

```
SomeComponent>>#states
  ^ Array with: model.
```
Subcomponents

> It is common for a component to display instances of other components.

> Components can be nested into each other using the composite pattern.

> A subcomponent is displayed using the method `#render:` on the canvas.
Subcomponents are usually stored within instance variables of the parent component.

Subcomponents are commonly created lazily or as part of the components `initialize` method.

```ruby
SomeComponent>>initialize
    super initialize.
    counter := WACounter new.
```
Enable Children

> Parent Components *must* implement a `#children` method returning a collection of subcomponents that they *might* display.

> If you fail to specify `#children` correctly, Seaside will raise an exception.

```ruby
SomeComponent>>children
  ^ Array with: counter
```
Children are rendered by sending the message `#render:` to the rendering canvas.

> Never directly send `#renderContentOn:` to the subcomponent.

``` Smalltalk
SomeComponent>>renderContentOn: html
  html heading level: 1; with: ‘My Counter’.
  html render: counter.
```
Components can be reused in different contexts within different applications.

Seaside is shipped with a small collection of widgets ready to use.

Load and use widgets that have been developed by the Seaside community.

Write your own widgets that exactly fit your needs.
Widgets: Examples

> Batched List

> Tab Panel

> Calendar

Have a look at the classes in Seaside-Components-Widgets
Custom Widgets

> Create a new component.
> Add methods to specify domain-model, subcomponents, properties…
> Assign CSS names/classes to make it skinnable with css style-sheet.
  — Implement method `#style` to return CSS for component
> Write tests and small example applications.
There is more..

> Development Tools
  — Demo in the Exercise Session (Halo, Configuration…)
  — Debugging: Next Lecture

> AJAX and script.aculo.us

> Persistency (Databases)
Literature

> HPI Seaside Tutorial:
  — http://www.swa.hpi.uni-potsdam.de/seaside/tutorial


> Ducasse, Lienhard, Renggli: Seaside, a Multiple Control Flow Web Application Framework (Proceedings ISC04)

> …. More at Seaside.st
License

http://creativecommons.org/licenses/by-sa/3.0/

Attribution-ShareAlike 3.0 Unported

You are free:
- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author’s moral rights.