A Smalltalk by the Seaside
Implementing a Website the OO-Way

Bernat Romagosa i Carrasquer

Consultant: Jordi Delgado Pin

Enginyeria Tècnica en Informàtica de Gestió
Universitat Oberta de Catalunya

1 Thanks

Thanks to:

Jordi Delgado Pin
For introducing me to Smalltalk, Squeak and Seaside, and for helping me along the whole process of completing this final project.

The Smalltalk Workgroup at Citilab
For helping me whenever I had a doubt and for letting me share my progress with them.

The people at *#seaside*, *#squeak*, *#smalltalk*, *#css* and *#latex* at *irc.freenode.com*
For kindly answering every single doubt and question.

The people at the Seaside mail lists
For replying to all of my e-mails and helping me solve every problem.

2 Abstract, Keywords and Project Area

A Smalltalk by the Seaside

by Bernat Romagosa i Carrasquer, Enginyeria Tècnica en Informàtica de Gestió (Technical Business Computer Science), Universitat Oberta de Catalunya (Open University of Catalonia), June 2009.

Abstract:

Modern languages and frameworks tend to depend too much on **HTML**, a technology born in the early nineties. Even most advanced frameworks keep making use of methodologies comparable to **GoTo** statements (**href**), which are known to potentially cause trouble due to arbitrary workflow. On the other hand, when developing web applications, we usually need to think about keeping state when needed, and to do so we map it into databases, files or whatever other support, which is a primitive way of dealing with such issues if we think of how desktop applications are developed. With this project we mean to introduce **Seaside**, a **Smalltalk** web framework that provides a solution for the problems aforementioned.

The project comes in two different parts, the first one consists in a real website developed in **Seaside**, while the second part means to help solve **Seaside**'s lack of documentation by writing a step-by-step manual on building websites.

Keywords:

Seaside, Squeak, Smalltalk, web development, web frameworks, object oriented programming.

Project Area:

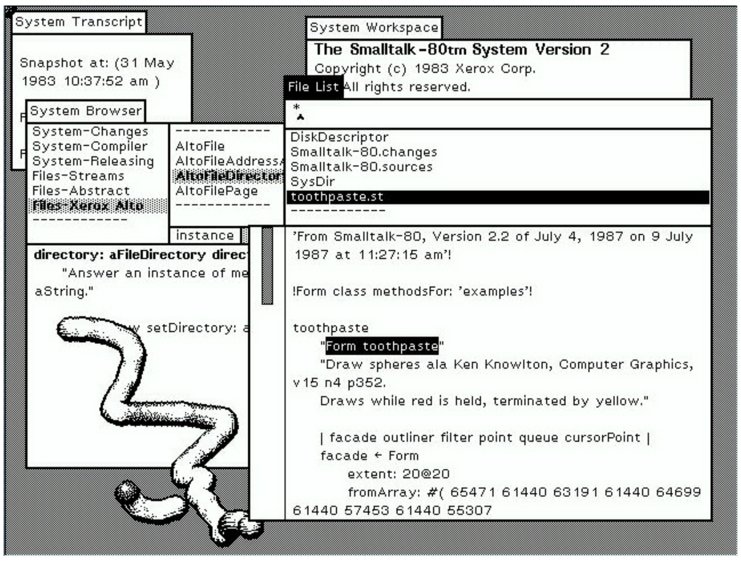
Web development by means of an object-oriented framework.

Contents

3 Introduction

3.1 Justification and Context of the Final Project

There are several **Smalltalk** implementations today, but when we look back in time we clearly see an inflection point on the popularity of the language with the birth of **Squeak** on 1996. **Squeak** is the newest, brightest opensource implementation of the **Smalltalk-80** environment/language developed by some of the same people who actually worked together creating the first **Smalltalk** in the late seventies.



```
Snapshot at: (31 May
1983 10:37:52 am )
```

```
The Smalltalk-80tm System Version 2
Copyright (c) 1983 Xerox Corp.
All rights reserved.
```

```
directory: aFileDirectory direc
    "Answer an instance of me
aString."
        setDirectory: a
```

```
'From Smalltalk-80, Version 2.2 of July 4, 1987 on 9 July
1987 at 11:27:15 am'!

!Form class methodsFor: 'examples'!

toothpaste
    "Form toothpaste"
    "Draw spheres ala Ken Knowlton, Computer Graphics,
v15 n4 p352.
    Draws while red is held, terminated by yellow."

    | facade outliner filter point queue cursorPoint |
    facade ← Form
        extent: 20@20
        fromArray: #( 65471 61440 63191 61440 64699
61440 57453 61440 55307
```

Figure 1: *A screenshot of the old Smalltalk-80 environment/language*

Squeak is not a mainstream language at all, in fact the amount of users belonging to the main mailing lists doesn't go over 2000, but its community is a very privileged one, counting with a few very active members who are really well-positioned into the software development world and computer science in general.

In the context of this new **Smalltalk** implementation, some programmers began to wonder what would it be like developing websites the object-oriented way, and this is where **Seaside** came out. **Seaside** is a powerful web framework for **Smalltalk**, originally implemented for **Squeak**, but nowadays ported to several other **Smalltalks**. The point in Seaside is to be able to code websites *as if* we were coding any other piece of software and, most important, do it the object-oriented way.

When programming desktop applications we don't usually deal with system calls when we want to print something or to draw windows and controls; we don't need -nor want- to know how the operating system rendering engine works. On the other hand, when we are developing websites we need to speak the same language the browser does, which slows us down and makes our main task more difficult and tedious. The idea behind **Seaside** is exactly this one, we will not see a single HTML statement, we won't even know how the website is going to look like -if we don't want to- because design and code are fully separated from each other.

3.2 Objectives

This work is meant to be a manual for newcomers to **Seaside**, its aim is to illustrate the whole process of building a website from scratch. The final piece of software will be a fully functional dynamic website, including sessions and users, as well as some simple dynamic components.

We expect this manual to be specific enough but still abstract enough, that is why we are going to name our website *"Whatever"* trying to stay away from the actual contents or particular purpose of a given site.

As a result of following this manual we have been able to develop two real websites, one of which was a direct pre-thought objective of this final project.

3.3 Approach and Methodology

The methodology followed to write this manual has been to document the learning process of developing a website as it was being built, focusing on those aspects that took the most to understand and master.

I began by developing a test application in **Squeak**, meant to teach myself the language and gain some experience before attempting to build something bigger. The result was a fully-functional educational software called **Scorer**[1][1], aimed at teaching children how to read musical scores and which has been being used in Mestral School in Sant Feliu de Llobregat (Barcelona) by its music teacher and his students.

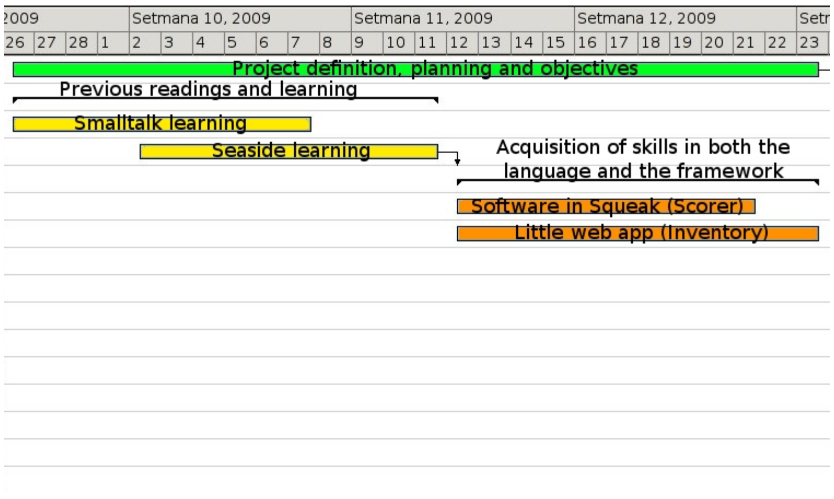
Meanwhile I followed the manual **An Introduction to Seaside**[2][2] in order to get started in Seaside itself and to learn how to develop a real project. Nevertheless, the mentioned manual does not focus much on building common websites, but on the specific strong points of the framework, which allow us to program web applications as if they were desktop ones. The result of the knowledge earned by reading **An Introduction to Seaside**[2] was the development of a simple web application used to manage the inventory of the different departments at the aforementioned Mestral School, which should begin to be used next September.

At this point I felt I had enough experience to begin coding the final product while documenting the whole process, and so I started writing this manual and developing **EduTech**'s website.

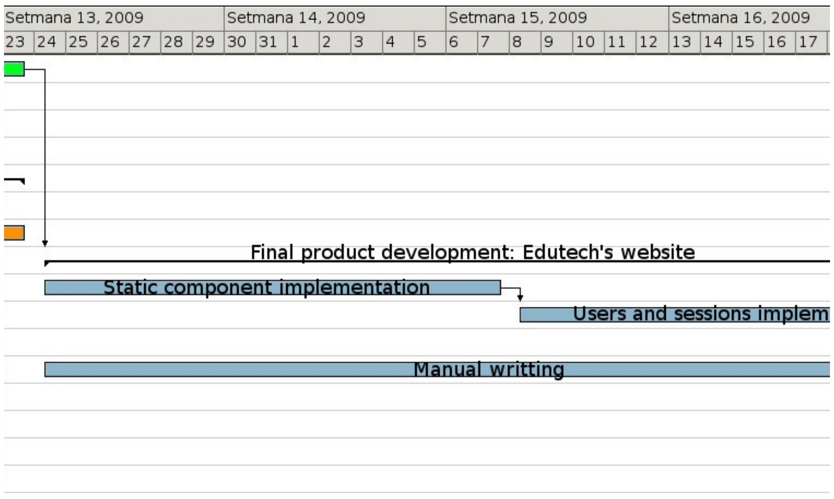
This whole project -including every image, video, document, piece of software and whatever other support- was developed by exclusively using free software under Debian GNU/Linux.

[1]Its source and binaries can be found at *http://sourceforge.net/projects/scorer*.
[2]written by the Software Architecture Group at the Hasso Plattner Institute

3.4 Project Planning

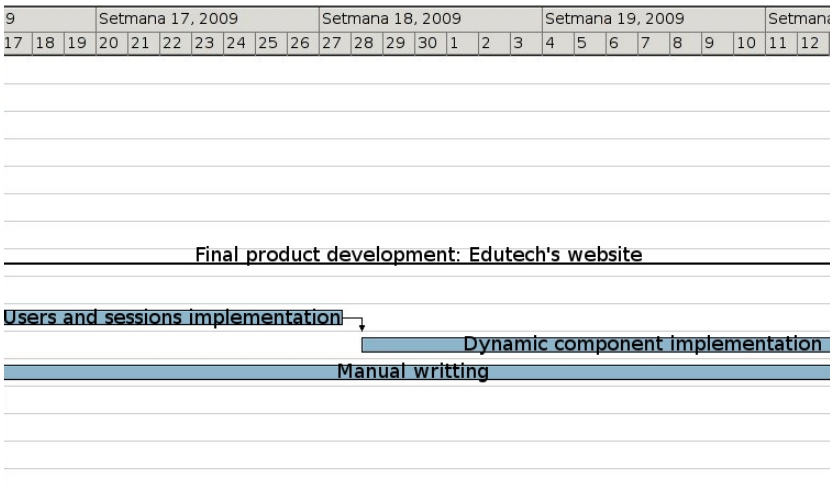


(a)

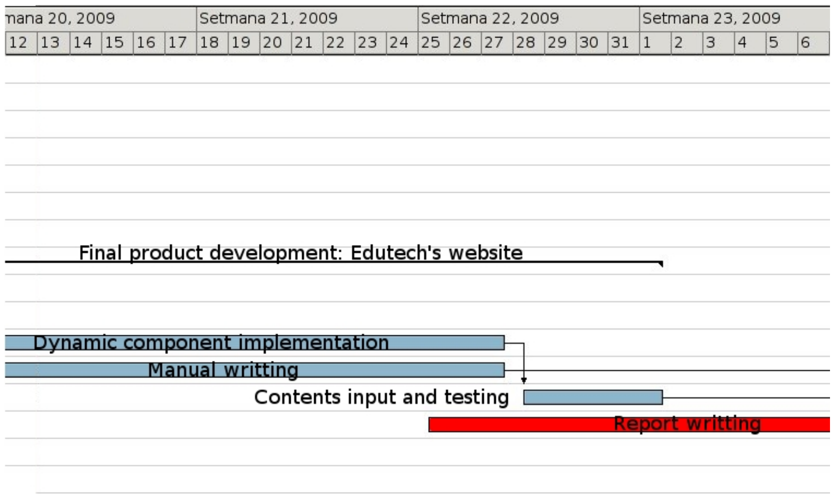


(b)

Figure 2: *Project planning: **(a)** February the 26th - March the 23rd, **(b)** March the 23rd - April the 17th*



(a)



(b)

Figure 3: *Project planning:* ***(a)*** *April the 17th - May the 12th,* ***(b)*** *May the 12th - June the 6th*

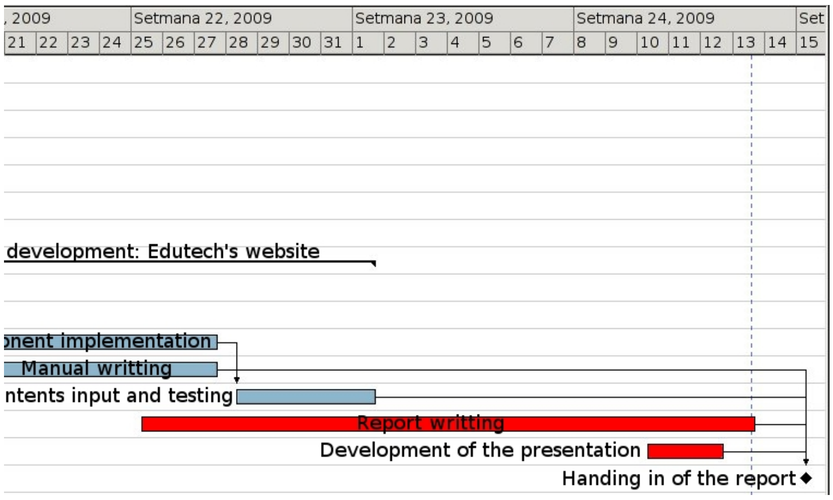
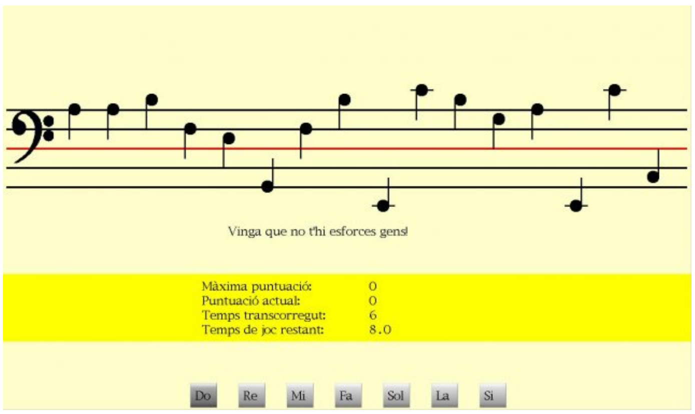


Figure 4: *Project planning: May the 21st - June the 15th*

3.5 Final Products

The final products obtained by this project are detailed next:

- **Scorer**:[1] An educational software meant to help music students get skillful at reading scores while playing. Since April 2009, it is being used at Mestral School in Sant Feliu de Llobregat (Barcelona) by its music teacher, with students ranging from 6 to 14 years old.

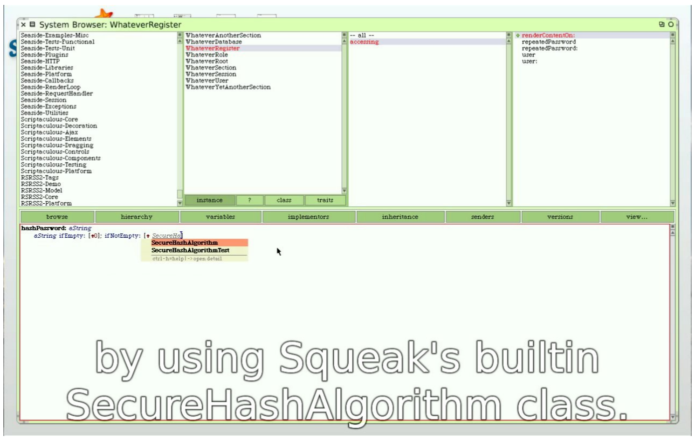


Vinga que no t'hi esforces gens!

Màxima puntuació:	0
Puntuació actual:	0
Temps transcorregut:	6
Temps de joc restant:	8.0

Figure 5: *Scorer: screenshot of the game*

- **A Smalltalk by the Seaside**: A manual teaching how to build a general-purpose website from the very beginning, featuring static contents, users and sessions, as well as dynamically editable content (a personal blog). This tutorial also features a 35 minutes subtitled screencast[3] illustrating the first two chapters in real-time.

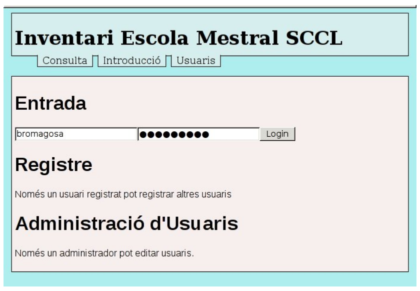


```
hashPassword: aString
    aString ifEmpty: [↑0]; ifNotEmpty: [↑ SecureHa
```

by using Squeak's builtin
SecureHashAlgorithm class.

Figure 6: *A Smalltalk by the Seaside: screenshot of the screencast [3]*

- **Inventari Escola Mestral**: A web application aimed at the management of Mestral School's inventory.



Inventari Escola Mestral SCCL

Consulta | Introducció | Usuaris

Entrada

bromagosa ●●●●●●●●● Login

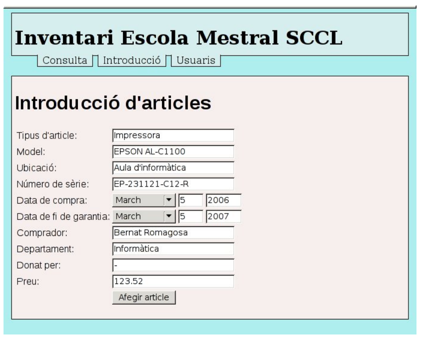
Registre

Només un usuari registrat pot registrar altres usuaris

Administració d'Usuaris

Només un administrador pot editar usuaris.

(a)



Inventari Escola Mestral SCCL

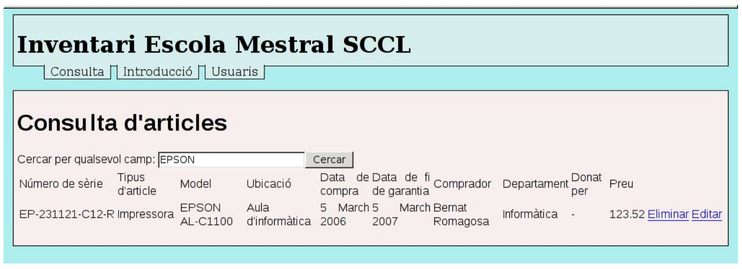
Consulta | Introducció | Usuaris

Introducció d'articles

Tipus d'article:	Impressora
Model:	EPSON AL-C1100
Ubicació:	Aula d'informàtica
Número de sèrie:	EP-231121-C12-R
Data de compra:	March 5 2006
Data de fi de garantia:	March 5 2007
Comprador:	Bernat Romagosa
Departament:	Informàtica
Donat per:	-
Preu:	123.52

Afegir article

(b)



Inventari Escola Mestral SCCL

Consulta | Introducció | Usuaris

Consulta d'articles

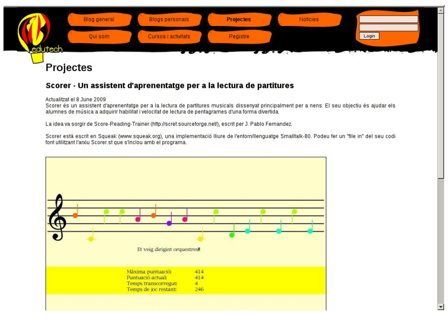
Cercar per qualsevol camp: EPSON Cercar

Número de sèrie	Tipus d'article	Model	Ubicació	Data de compra	Data de fi de garantia	Comprador	Departament	Donat per	Preu		
EP-231121-C12-R	Impressora	EPSON AL-C1100	Aula d'informàtica	5 March 2006	5 March 2007	Bernat Romagosa	Informàtica	-	123.52	Eliminar	Editar

(c)

Figure 7: *Inventory management web application:* ***(a)*** *user administration,* ***(b)*** *item input and* ***(c)*** *item grid and management*

- **Edutech's Website**:[4] The official website of **EduTech**, a teaching / learning / development project hosted by **Citilab**, in Cornellà de Llobregat (Barcelona). The website counts with a personal blog for each user, a general blog (built by aggregating all user posts), different user roles, dynamically-editable content of all sections and a user administration panel, being these two last features only available for administrators.



Projectes

Scorer - Un assistent d'aprenentatge per a la lectura de partitures

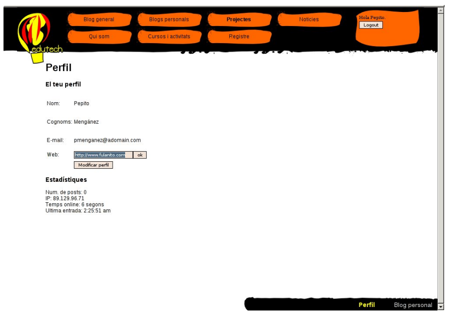
Actualitzat el 8 June 2009
Scorer és un assistent d'aprenentatge per a la lectura de partitures musicals dissenyat principalment per a nens. El seu objectiu és ajudar els alumnes de música a adquirir habilitat i velocitat de lectura de pentagrames d'una forma divertida.

La idea va sorgir de Score-Reading-Trainer (http://scret.sourceforge.net/), escrit per J. Pablo Fernandez.

Scorer està escrit en Squeak (www.squeak.org), una implementació lliure de l'entorn/llenguatge Smalltalk-80. Podeu fer un "file in" del seu codi font utilitzant l'arxiu Scorer.st que s'inclou amb el programa.

Màxima puntuació:	414
Puntuació actual:	414
Temps transcorregut:	4
Temps de joc restant:	246

(a)



Perfil

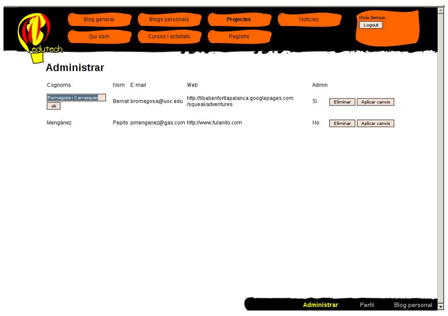
El teu perfil

Nom:	Pepito
Cognoms:	Mengánez
E-mail:	pmenganez@adomain.com
Web:	http://www.fulanito.com

Estadístiques

Num. de posts: 0
IP: 89.129.96.71
Temps online: 6 segons
Ultima entrada: 2:25:51 am

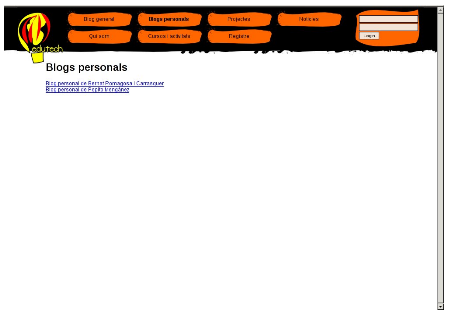
(b)



Administrar

Cognoms	Nom	E-mail	Web	Admin	
Romagosa i Carrasquer ok	Bernat	bromagosa@uoc.edu	http://tibabenfortlapalanca.googlepages.com /squeakadventures	Si	Eliminar Aplicar canvis
Mengánez	Pepito	pmenganez@gas.com	http://www.fulanito.com	No	Eliminar Aplicar canvis

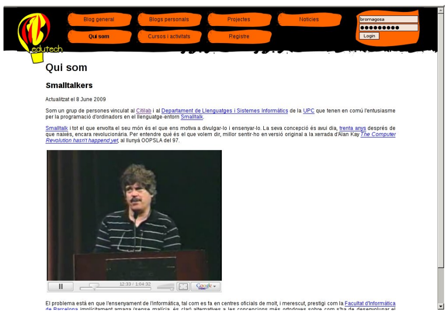
(c)



Blogs personals

Blog personal de Bernat Romagosa i Carrasquer
Blog personal de Pepito Mengánez

(d)



Qui som

Smalltalkers

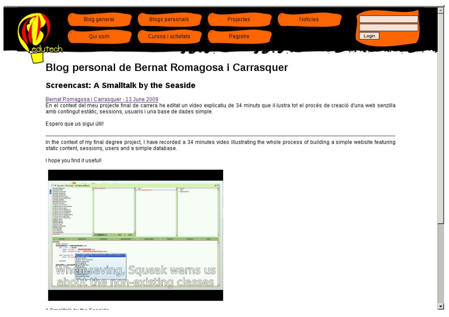
Actualitzat el 8 June 2009

Som un grup de persones vinculat al Citilab i al Departament de Llenguatges i Sistemes Informàtics de la UPC que tenen en comú l'entusiasme per la programació d'ordinadors en el llenguatge-entorn Smalltalk.

Smalltalk i tot el que envolta el seu món és el que ens motiva a divulgar-lo i ensenyar-lo. La seva concepció és avui dia, trenta anys després de que naixés, encara revolucionària. Per entendre què és el que volem dir, millor sentir-ho en versió original a la xerrada d'Alan Kay *The Computer Revolution hasn't happend yet*, al llunyà OOPSLA del 97.

El problema està en que l'ensenyament de l'informàtica, tal com es fa en centres oficials de molt, i merescut, prestigi com la Facultat d'Informàtica de Barcelona implícitament amaga (sense malícia, és clar) alternatives a les concepcions més ortodoxes sobre com s'ha de desenvolupar el

(e)



Blog personal de Bernat Romagosa i Carrasquer

Screencast: A Smalltalk by the Seaside

Bernat Romagosa i Carrasquer - 13 June 2009
En el context del meu projecte final de carrera he editat un vídeo explicatiu de 34 minuts que il·lustra tot el procés de creació d'una web senzilla amb contingut estàtic, sessions, usuaris i una base de dades simple.

Espero que us sigui útil!

In the context of my final degree project, I have recorded a 34 minutes video illustrating the whole process of building a simple website featuring static content, sessions, users and a simple database.

I hope you find it useful!

A Smalltalk by the Seaside

(f)

Figure 8: *Edutech's official website:* ***(a)*** *dynamic content editable by administrators,* ***(b)*** *live editing of a user profile,* ***(c)*** *live user administration,* ***(d)*** *personal blog list,* ***(e)*** *embedded multimedia content in an editable section and* ***(f)*** *a personal blog.*

3.6 Description of All Chapters

4 **1st CHAPTER - Creating a Static Website**
In this chapter we are going to introduce some basic **Seaside** concepts and we will end up with a functional website featuring a couple of tabs and some static sections.

4.1 **Previous Steps**
Some recommendations and concepts before beginning to build the website.

4.2 **A Root Component And Some Empty Tabs**
Building the main basic structure of our example website.

4.3 **Rendering Our Root Component**
Coding the way the root component is shown.

4.4 **Adding a CSS Stylesheet**
A little bit of design to make our example website look better and to show what can be achieved by using **CSS**.

5 **2nd CHAPTER - Sessions and Users**
In this chapter we will be learning how to add a simple database capable of holding users, along with understanding how to deal with sessions in **Seaside**.

5.1 **What Is a Session, What Is a User**
Some basic concepts related to users and sessions.

5.2 **Creating a User Object**
Building a class representing users with all their attributes.

5.3 **A Simple Database**
A simple image-based database useful for keeping all of our users.

5.4 **Subclassing The WASession Class**
Creating our own session class to be able to specify which user class and which database we want our website to be using.

5.5 **Adding a Sign In Component**
Building a component which shows a form capable of registering new users.

5.6 **Embedding Components**
How to embed a complex component -such as last subsection's sign in component- into another component, in our case the root one.

5.7 **Adding a Login Component**
Creating a component to let already registered users log into our website.

6 **3rd CHAPTER - A (Very) Brief Introduction to Magritte**
In this chapter we will superficially introduce **Magritte**, a very powerful framework capable, among many other features, of building forms automatically.

6.1 **What Is Magritte**
Defining what Magritte is and what it is used for.

6.2 **Descriptions**
Explanation of what a Description is in **Magritte** and how to build our own ones.

7 **4th CHAPTER - Adding Dynamic Components**
In this chapter we will be building a dynamic component, meaning that it can be modified by users in real-time without the need of programming.

7.1 **A Simple Personal Blog**
How to build a simple personal blog component.

7.1.2 **Posts And Comments: The Objects**
Creating classes for posts and comments, along with all of their attributes.

4 1st CHAPTER - Creating a Static Website

4.1 Previous Steps

Some links are provided below for a fast and concise introduction to **Smalltalk** and **Squeak**:

- **Squeak by Example** *http://www.squeakbyexample.org*
- **A Terse Guide to Squeak** *http://squeak.joyful.com/LanguageNotes*

Prior to following this tutorial, it is strongly recommended to read the official introductory tutorial, which can be found at:

- **A Walk on the Seaside** *http://www.seaside.st/documentation/tutorials*

The **Squeak** image used in the present tutorial can be found at **Seaside**'s official website, squeak download subsection (*http://seaside.st/download/squeak*) filed under **Developer Image**.

4.2 A Root Component And Some Empty Tabs

Our first approach to the final website is just going to have a couple of tabs, a header and some random contents.

Since we are dealing with static contents, we will not need anything else than web components. Remember a component is an object with a visual web representation, that is, an object that can be rendered in HTML. In **Seaside**, websites are, in a way, tree-structured. To put it simple, the main component is the one that contains the whole website, the one that is at the beginning of this tree, hence: the **root** component.

Let us begin by creating our root component by opening the **Class Browser**, creating a new category and naming it *"Whatever"*. This is the category under which our whole website is going to be filed. Next step is creating a new subclass of **WAComponent**, which is **Seaside**'s web component class:

```
WAComponent subclass: #WhateverRoot
    instanceVariableNames: 'aCoupleOfTabs selectedTab'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

The two instance variables will be used as explained below:

aCoupleOfTabs Will contain an **OrderedCollection** with the whole tree structure for the tab items.

selectedTab Will contain the selected item from **aCoupleOfTabs**. That is, the component pointed by the tab that was last clicked.

For our root component to be classified as such by **Seaside**, we will have to override a couple of methods in the class side of our **WhateverRoot** class, namely:

```
WhateverRoot class >> canBeRoot
    ^true


WhateverRoot class >> initialize
    (self registerAsApplication: 'whatever')


WhateverRoot class >> description
    ^ 'A website about whatever'
```

The first method is for **Seaside** to know this is a component that is eligible as a root component. The second method [3] registers the component as a web application entry point programmatically, giving it the name *"whatever"*. In the real world, this means our website can now be accessed at:

*http://localhost:8080/Seaside/**whatever***

We are now done with the side class methods, let us get back to instance and add the following methods:

```
WhateverRoot >> title
    ^ 'Whatever Website'

WhateverRoot >> updateRoot: anHtmlRoot
    super updateRoot: anHtmlRoot.
    anHtmlRoot title: self title
```

Again, the first method is obvious. The role of the second one is -in a nutshell- to allow us to use **CSS styles** and **JavaScript** in each one of our components. Specifically, it allows components to add items to the html head.

```
WhateverRoot >> states
    ^ Array with: self
```

The method **states** is the one that allows us to use the famous back-button feature. **Seaside**, unlike other web frameworks, keeps the state of the website all the time, meaning that view and data are both saved as a unit. This makes the browser's back-button work as it would in any other non-web-based application.

Implementing it is optional, but we might need it later on.

The next method calls some selectors we have not defined yet, that is why **Squeak** is going to ask us what to do with them. Since they will end up being

[3]This method is deprecated in **Seaside** 2.9, instead we would use:

```
WhateverRoot class >> initialize
   WAAdmin register: self asApplicationAt: 'whatever'
```

classes anyway, we can save some work by telling **Squeak** to create new classes for each one of these selectors.

```
WhateverRoot >> initialize
    super initialize.
    aCoupleOfTabs := OrderedCollection new
    add: 'A Tab' -> (Array
        with: 'A section' -> WhateverSection new
        with: 'Another section' -> WhateverAnotherSection new);
    add: 'Another Tab' -> (Array
        with: 'Yet another section' -> WhateverYetAnotherSection new);
    yourself.
    selectedTab := aCoupleOfTabs first value
```

As the code shows, when the website is first called (created) we initialize **aCoupleOfTabs** as follows:

- **aCoupleOfTabs** is an **OrderedCollection** containing the names of the tabs we will need in our website.
- Each of those tabs is an **Association** between a **String** and an **Array** of **Associations**.
- Each of those **Associations** link a **String** (which will be displayed later) to a component (which we will create later on).

4.3 Rendering Our Root Component

Up to now we have already built the inner structure of our website by using web components and referencing them from the root component. To render a component, **Seaside** uses a method called **renderContentOn**, which we have to define for each one of the components we want to have a view for (which is usually all of them).

Let us first override the root's **renderContentOn** method:

```
WhateverRoot >> renderContentOn: html
    self renderHeaderOn: html.
    html div id: 'tabs'; with: [self renderTabsOn: html].
    self renderChildrenOn: html
```

Essentially, **html** is a rendering object -instance of **WARenderCanvas**- which has a very wide range of methods for the different HTML tags and combination of tags. For instance, in this method we have used the message **div** with the arguments **id: 'tabs'** and **with:** [**content**], which essentially corresponds to HTML's <**DIV id="tabs"**> **content** < /**DIV**>.

As for the rest, we decided to split the rendering of the three separate parts of our website into three rendering methods.

Header will be rendered by **renderHeaderOn**, it will contain the website title as well as a login/logout component.

Tabs will be rendered by **renderTabsOn**, it will hold the different tabs that are going to let us navigate through the contents.

Children will be rendered by **renderChildrenOn**. Remember Seaside's workflow resembles a tree structure; a child could be defined as a branch of the root component, which is enough of a definition for the moment. We will take up this subject again later in more depth.

Now that we have delegated the rendering to these three methods, let us write them:

```
WhateverRoot >> renderHeaderOn: html
    html div id: 'header';
        with: [html heading: self title]
```

This message will build an HTML **div** identified as 'header' and containing an HTML heading with the title of the website (remember we have aleready defined **title** previously).

```
WhateverRoot >> renderTabsOn: html
    html unorderedList id: 'tabs'; with: [
        aCoupleOfTabs do: [ :each |
            html listItem: [
                html anchor
                    class: (selectedTab = each value
                        ifTrue: [ 'active' ]);
                    callback: [ selectedTab := each value ];
                    with: each key ] ] ]
```

In this method we are creating an **unorderedList** (an HTML unorderedList, not a Smalltalk one) of anchors, each of which corresponds to an item of our **aCoupleOfTabs** collection, and setting it to link the selected tab with the instance variable we named after that purpose.

```
WhateverRoot >> renderChildrenOn: html
    html div id: 'content'; with: [
        selectedTab do: [ :each |
            html heading: each key.
            html paragraph; render: each value.
            ] ]
```

Our website has a main structure defined in our root component, and as we go navigating through anchors, some parts of it keep changing. Those changing parts are called children. So, when a tab is selected, **selectedTab** will become a reference to that tab, and **renderChildrenOn** will print out a heading with that tab's name and its contents in a paragraph.

The next step is to fill up the **renderContentOn** methods for the rest of our components. For the moment, what we write in them is totally irrelevant.

When we wrote the **initialize** method for our root component, we asked **Seaside** to automatically build some missing classes for us, now we need to modify them to turn them into **WAComponent** subclasses, as shown next:

```
WAComponent subclass: #WhateverSection
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'

WAComponent subclass: #WhateverAnotherSection
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'

WAComponent subclass: #WhateverYetAnotherSection
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

Now let us fill their respective **renderContentOn** methods with some random contents:

```
WhateverSection >> renderContentOn: html
    html paragraph: 'This is a section, it contains this text.'


WhateverAnotherSection >> renderContentOn: html
    html paragraph: 'This is another section containing this paragraph...'.
    html paragraph: '...and this paragraph too.'.


WhateverYetAnotherSection >> renderContentOn: html
    html paragraph: 'And this is yet another section, containing:'.
    html orderedList with: [html listItem: 'The previous paragraph';
        listItem: 'This list';
        listItem: 'The three items of this list'].
```

4.4 Adding a CSS Stylesheet

Design of websites coded in **Seaside** is always done by using CSS, that is why we should always be careful when programming to ensure every distinguishable element has its own class or identifier set in order to help the designer's task of arranging them along the page.

CSS is read by **Seaside** from a method called **style**, which has to be coded in the instance side of the component the CSS will be applied to. Since the whole page is rendered from the root component, we can save ourselves some trouble by coding it all there in a single method.

Being that CSS[5] is not the topic of this manual, we will just provide an example one:

```
WhateverRoot >> style

    ^ '
    body{
        background-color: #aeeeee;
    }

    #header{
        border:1px solid;
        border-color: #2c89a0;
        height: 70px;
        margin-left: 5px;
        margin-right: 5px;
        padding-left: 5px;
        padding-right: 5px;
        padding-top: 5px;
        padding-bottom: 5px;
        background-color: #d7eeee;
    }

    #content{
        font-family: arial;
        text-align: justify;
        margin-left: 5px;
        margin-right: 5px;
        padding-left: 5px;
        padding-right: 5px;
        padding-top: 5px;
        padding-bottom: 5px;
        background-color: #f7eeee;
        border:1px solid;
        border-color: #2c89a0;
    }

    #tabs{
        list-style-type: none;
        margin-top: 2px;
        margin-bottom: 10px;
    }

    #tabs li{
        display: inline;
    }

    #tabs li a{
        text-align: center;
        width: 100%;
        height: 25px;
        color: #000;
        background-color: #d7eeee;
        margin: 0.5em;
        padding: 0.5em;
        font-size: 12px;
        text-decoration: none;
        border-bottom:1px solid;
        border-left:1px solid;
        border-right:1px solid;
        border-color: #2c89a0;
    }

    #tabs li a:hover{
        font-weight: bold;
    }
'
```

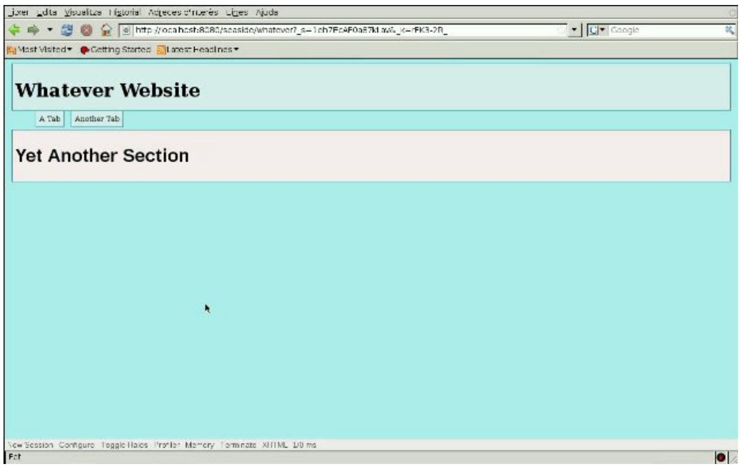


Figure 9: *Our website with a simple CSS applied*

Usually we don't want our graphic designer to be forced to deal with our code, which could be troublesome, to solve this we can ask our **style** method to read an external stylesheet file instead of hardcoding it, which can be done as follows:

```
WhateverRoot >> style
    |aCSSFile theCSSdata|
    aCSSFile := FileStream fileNamed: 'style.css'.
    theCSSdata := aCSSFile contentsOfEntireFile.
    aCSSFile close.
    ^theCSSdata
```

Now the file **style.css** must be placed in the same directory as the running image and can be substituted or modified live anytime by the usual ways (FTP for instance) a designer is used to.

5 2nd CHAPTER - Sessions and Users

5.1 What Is a Session, What Is a User

A session is an object which is automatically created when a user accesses a web application (in our case, the Whatever Website). Its aim is to handle and hold all data relative to that particular usage and navigation through the application, and to do so until it expires. The lifetime of a session is controlled by a method named **defaultTimeoutSeconds**, time after which it is going to be garbage collected by the system.

In other words, if we considered a web application (or website) in its entirety as a class, a session would be just an instance.

Hence, a user is an entity which interacts with a particular session. But more than that, a user needs some of the changes and actions performed during that session to persist, so that they can be found again in a later session, ensuring that the work done will be kept to be taken from a certain point on without having to manually save it or repeat it all over.

5.2 Creating a User Object

Just like everything else in **Squeak** -and consequently, in **Seaside**-, a user is also an object with its attributes and actions. In our case, we are going to need a user to have the following instance variables, explained only when needed:

- firstName
- surname
- userName
- password *(properly encrypted)*
- email
- website
- role *(used to define groups of users with different roles and permissions within the website)*

Let us define them:

```
Object subclass: #WhateverUser
    instanceVariableNames: 'userName password firstName
                            surname email website role'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

The only methods a user needs are the accessors (setters/getters), which **Squeak** can automatically generate for us. To do so, right-click on the class name within the **System Browser** and choose **Refactor class → Accessors**.

As for the role, we should create a very simple class to define it, which we can extend at our own will later on:

```
Object subclass: #WhateverRole
    instanceVariableNames: 'roleName isSuperAdmin'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

The only method we are going to implement (besides the accessors, which we automatically created by following the same procedure as before) is the initialization:

```
WhateverRole >> initialize
    super initialize.
    self roleName: 'user'.
    self isSuperAdmin: false.
```

For the moment we are not going to make use of user roles, but we leave it prepared in case we need to assign different permissions to different groups of users.

5.3 A Simple Database

Like we stated at the beginning of the present chapter, a user needs some data to be persistant. Due to the complexity and extension an introduction to object-oriented databases would cost in comparison to the actual topic we are dealing with (sessions and users), we prefer to build a very simple image-based database, which we could later on map to a real one.

For the moment we are going to use our simple database to store users, which is the only data our website handles right now, but later on we will see how to store anything else.

Let us create a new class named **WhateverDatabase** with the class variables **Users** and **WriteMutex**. Notice that they are class variables because all data must be shared by all instances.

```
Object subclass: #WhateverDatabase
    instanceVariableNames: ''
    classVariableNames: 'Users WriteMutex'
    poolDictionaries: ''
    category: 'Whatever'
```

The first thing we are going to do is to lazy-initialize **Users**, remember that we are in the class side:

```
WhateverDatabase class >> users
    ^ Users ifNil: [Users := OrderedCollection new]
```

As the code shows, we will store all of our users in an **OrderedCollection**, the preferred data structure in **Smalltalk**. Let us define **WriteMutex** next:

```
WhateverDatabase class >> writeMutex
    ^ WriteMutex ifNil: [WriteMutex := Monitor new]
```

Again, we lazy-initialize the class variable. In this case, **WriteMutex** becomes a **Monitor**, which is an object that provides process synchronization; we will use it to be able to save the image in background, as shown in the implementation of the next two instance methods:

```
WhateverDatabase >> saveImage
    self class writeMutex critical: [self saveImageWithoutMonitor]


WhateverDatabase >> saveImageWithoutMonitor
    SmalltalkImage current saveSession.
```

Notice how we ask our **writeMutex** object (which is a **Monitor**) to background the process of saving the current image. From now on, anytime we modify the database we should call **saveImage** in order to have the new data kept.

Now we just need some methods to find, write and erase users from the database:

```
WhateverDatabase >> addUser: aUser
    self class users add: aUser.
    self saveImage.
    ^aUser.

WhateverDatabase >> removeUser: aUser
    self class users remove: aUser ifAbsent: [^aUser].
    self saveImage.
    ^nil
```

Since **users** is an **OrderedCollection**, adding and removing items from it is easy and fast. As for searches:

```
WhateverDatabase >> findUserByUsername: aUsername
    ^ self class users
        detect: [:each | each userName = aUsername]
        ifNone: [nil]
```

For each user in the database, we check if its username matches the requested one; if it does we return it, otherwise we return **nil**. Notice again how an

OrderedCollection was the proper choice as a user container, saving us time and trouble when dealing with its items.

We now have everything we need to finish the development of our web application without having to worry about persistence, we could easily map this database to a real non-image-based one, but this should not be necessary as long as our website is not accessed by too many users at a time, since the process taking the most time and resources is saving the image and we have already backgrounded it, allowing the rest of the processes to run unalteredly.

5.4 Subclassing The WASession Class

Since we need to deal with particular sessions, as we hold users who in turn hold their data, we are going to have to extend **Seaside**'s session class, so let us begin by subclassing it:

```
WASession subclass: #WhateverSession
    instanceVariableNames: 'user database'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

A session, as we pointed out before, deals with the current user (the **user** instance variable) and has a database to which it writes and accesses (the **database** instance variable). We need to create accessors for both variables, which **Squeak** does automatically for us -as shown before- when we choose **accessors** in **refactor class** from the class context menu.

So what else should our particular session class handle? To begin with, we should map **database** to our **Database** when the session is initialized:

```
WhateverSession >> initialize
    super initialize.
    self database: WhateverDatabase new.
```

Next step is to define what logging a user in means, which is nothing but telling our session that the current user is the given one:

```
WhateverSession >> login: aUser
    self user: aUser.
```

In the same way, when a user logs out:

```
WhateverSession >> logout
    self user: nil.
```

Since we are often going to need to know whether a user is logged in, we implement an additional method which tells us:

```
WhateverSession >> isLoggedIn
    ^self user isNil not
```

The next and last method we are going to implement is just a map to a method we previously implemented for our **WhateverDatabase**:

```
WhateverSession >> findUserByUsername: aUsername
^ self database findUserByUsername: aUsername.
```

Our session class is now complete and usable, but we must inform **Seaside** that we want to use **WhateverSession** instead of **WASession** for our website. To do so, we have to override the default session class through the web administration panel.

5.5 Adding a Sign In Component

There should be a way for users to register to our website, which is exactly what this component is meant to allow. We are going to name it **WhateverRegister** and it is going to have two instance variables to hold the user and the "repeated password", which we will explain later on:

```
WAComponent subclass: #WhateverRegister
    instanceVariableNames: 'user repeatedPassword'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

As always, we will ask **Squeak** to create instance variable accessors automatically for us. One easy mistake to make at this point is to assign a new **WhateverUser** to our instance user variable when initializing the object, which might seem logical but would lead to trouble given that the component is initialized only once per session.

To prevent this from happening, we will assign a new **WhateverUser** to our user instance variable every time the component is rendered.

For the first time, the rendering method for this component is going to be quite long, consisting in a form containing a table with a row for each one of the fields we want the user to input.

```
WhateverRegister >> renderContentOn: html
   self user: WhateverUser new.
   html div id: 'register';
       with: [html form: [html table id: 'registerTable'; with: [html
           tableRow: [html tableData:[ html text: 'First Name: '].
           html tableData:[html textInput
              callback: [:value | self user firstName: value].
              html break]].
           html
              tableRow: [html tableData: [html text: 'Surname: '].
           html tableData:[html textInput
              callback: [:value | self user surname: value].
              html break]].
           html
              tableRow: [html tableData:[html text: 'E-mail: '].
           html tableData:[html textInput
              callback: [:value | self user email: value].
              html break]].
           html
              tableRow: [html tableData:[html text: 'Website: '].
           html tableData:[html textInput
              callback: [:value | self user website: value].
              html break]].
           html break.
           html
              tableRow: [html tableData:[html text: 'Username: '].
           html tableData:[html textInput
              callback: [:value | self user userName: value].
              html break]].
           html
              tableRow: [html tableData:[html text: 'Password: '].
           html tableData: [html passwordInput
              callback: [:value | self user
                 password: (self hashPassword: value)].
              html break]].
           html
              tableRow: [html tableData:[html text: 'Password (repeat): '].
           html tableData:[html passwordInput
              callback: [:value | self
                 repeatedPassword: (self hashPassword: value)].
              html break]].
              html tableRow: [
           html tableData: []. html tableData:[html submitButton
              callback: [self registerUser];
              text: 'Sign in']]]]]
```

Only a few lines from the previous piece of code demand an explanation; we used a method called **hashPassword** which we haven't implemented yet, and another method called **registerUser** which we also need to implement.

hashPassword will use a SHA algorithm (already built-in in **Squeak**) to encrypt the string and secure it properly:

```
WhateverRegister >> hashPassword: aString
    aString ifEmpty: [^ 0];
    ifNotEmpty: [^ SecureHashAlgorithm new hashMessage: aString].
```

registerUser is the message that will add a user to the database with all the data given when submitting the form:

```
WhateverRegister >> registerUser
    self user password = repeatedPassword ifTrue:[
        self user role: WhateverRole new.
        self session database addUser: self user.
        self answer: self user.
        self inform: 'User ',self user userName,' registered
            successfully.']
    ifFalse: [
        self inform: 'Passwords do not match!']
```

The code above is quite self-explanatory, we first check if both passwords match, and if they do we add the user to the database and answer it (remember this method is called from within a callback, so we should provide an answer); otherwise we inform the user that passwords didn't match. Notice how we assign a new default role to the user, which we could later on change either programmatically or by setting up a superuser who has permissions to switch other users roles.

One thing we could now do is to add the component we just implemented to the collection of tabs, so that we can access it, which will require to rewrite our root component initialization, but before doing so we need to introduce component embedding.

5.6 Embedding Components

One of the most astonishing features of Seaside is its component reusability, clear examples of it can be found in many projects developed on top of Seaside, like **Pier**, a CMS with reusable pluggable components, or **ShoreComponents**, a collection of Seaside components totally independent and embeddable.

Embedding components is an easy task but, because of **Seaside**'s way of dealing with them, it might look a bit tricky at first glance. Let us clarify a little how it is done.

First of all, we saw in the first chapter of this manual how to embed static components (like **WhateverSection** or **WhateverAnotherSection**), but when the components we need to embed have to deal with callbacks and have their own state the process gets fairly complicated. In the first chapter we also talked about **children** and we briefly explained how the root component deals with them.

What we need to do when a child component has its own independent behavior is to make it persistent to the root component, that is, to add it as an instance variable which we can refer to by its name at all times, so let us rewrite the definition of our **WhateverRoot** class to include an instance of **WhateverRegister**:

```
WAComponent subclass: #WhateverRoot
    instanceVariableNames: 'aCoupleOfTabs selectedTab
                            registerComponent'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

We will also have to tell the root component that this component is one of its children, which we can do by overriding **WAComponent**'s **children** method:

```
WhateverRoot >> children
    ^Array with: registerComponent.
```

We now will have to initialize **registerComponent**, assigning it to a new instance of **WhateverRegister** when our root component is created, as well as add it to the tabs OrderedCollection, so let us modify our root component initialization method:

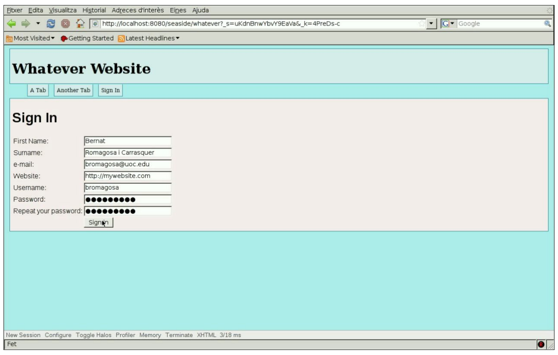
```
WhateverRoot >> initialize
    super initialize.
    registerComponent := WhateverRegister new.
    aCoupleOfTabs := OrderedCollection new
    add: 'A Tab' -> (Array
        with: 'A section' -> WhateverSection new
        with: 'Another section' -> WhateverAnotherSection new);
    add: 'Another Tab' -> (Array
        with: 'Yet another section' -> WhateverYetAnotherSection new);
    add: 'Sign in' -> (Array
        with: 'Sign in' -> registerComponent);
    yourself.
    selectedTab := aCoupleOfTabs first value
```

Our registering component is now embedded to our root component, so we can now test it on the web and try to add a new user.

5.7 Adding a Login Component

Since the login component is going to be pretty similar to the previously explained registering component, we will show here how to embed a component into a different place of our website, in this case we chose to include our login component at the top right corner of the page, inside its header.

As always, we begin by subclassing *WAComponent*:


Fitxer Edita Visualitza Historial Adreces d'interès Eines Ajuda
Google
Most Visited
Getting Started
Latest Headlines
A Tab
Another Tab
Sign In
Sign In
New Session Configure Toggle Halos Profiler Memory Terminate XHTML 3/18 ms
Fet

http://localhost:8080/seaside/whatever?_s=uKdnBnwYbvY9EaVa&_k=4PreDs-c

Whatever Website

First Name:	Bernat
Surname:	Romagosa i Carrasquer
e-mail:	bromagosa@uoc.edu
Website:	http://mywebsite.com
Username:	bromagosa
Password:	●●●●●●●●●●
Repeat your password:	●●●●●●●●●●

Figure 10: *Adding a new user through the new sign in component*

```
WAComponent subclass: #WhateverLogin
    instanceVariableNames: 'userName password'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

Next step is to automatically build accessors for both instance variables, as shown in previous chapters. Let us write our rendering method:

```
WhateverLogin >> renderContentOn: html
    self session isLoggedIn
        ifTrue: [html text: 'Hi ' , self session user firstName , '.'.
        html form: [html cancelButton class: 'btn';
             callback: [self logout]; text: 'Logout']]
    ifFalse: [html
            form: [html textInput on: #userName of: self;
                    value: ''.
                html passwordInput
                    callback: [:value | self
                            password: (self hashPassword: value)];
                    value: ''.
                html submitButton class: 'btn';
                     callback: [self validateLogin]; text: 'Login']]
```

First thing we do when rendering this component is check whether the user is logged in. When a user is logged in, the component just greets him and shows a logout button, which calls the not-yet-implemented **logout** method. If there is no user logged in yet, the component shows a form with two inputs, one for the username, another one for the password, along with a login button which calls another not-yet-implemented method called **validateLogin**.

The **btn** class here is set for design issues, so that our graphic designer can refer to the login component buttons in a different way when writing the CSS stylesheet.

Notice that we use **hashPassword** again, so we will have to write it here too. In order to follow **Smalltalk**'s best practices, we should find a way to reuse it instead of having two methods doing exactly the same in two different classes. The way to do that would be to subclass both components (that is, **WhateverRegister** and **WhateverLogin**) from a component implementing this method. We leave this improvement up to the reader in order to not complicate the main subject here explained.

```
WhateverLogin >> hashPassword: aString
    aString ifEmpty: [^ 0];
    ifNotEmpty: [^ SecureHashAlgorithm new hashMessage: aString].
```

As for the two methods we haven't yet implemented:

```
WhateverLogin >> logout
    self session logout.
    self inform: 'Goodbye!'.


WhateverLogin >> validateLogin
    |user|
    user := self session findUserByUsername: self userName.

    (user notNil and: [user password = self password ])
        ifTrue: [self session login: user. self answer: user]
        ifFalse: [self loginFailed]
```

The first method is self-explanatory, as for the second one, we try to find the user by its username in the database, if we are successful we login that user and answer it, while if we are unable to find it we call a method which we will next implement in order to deal with failed login attempts:

```
WhateverLogin >> loginFailed
    self inform: 'Login failed!'
```

By following the same procedure shown in previous chapter, we are going to embed this component, but this time into the header of the website:

```
WAComponent subclass: #WhateverRoot
    instanceVariableNames: 'aCoupleOfTabs selectedTab registerComponent
                            loginComponent'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'


WhateverRoot >> children
    ^Array with: registerComponent with: loginComponent.
```

```
 WhateverRoot >> initialize
     super initialize.
     loginComponent := WhateverLogin new.
     registerComponent := WhateverRegister new.
     aCoupleOfTabs := OrderedCollection new
     add: 'A Tab' -> (Array
         with: 'A section' -> WhateverSection new
         with: 'Another section' -> WhateverAnotherSection new);
     add: 'Another Tab' -> (Array
         with: 'Yet another section' -> WhateverYetAnotherSection new);
     add: 'Sign in' -> (Array
         with: 'Sign in' -> registerComponent);
     yourself.
     selectedTab := aCoupleOfTabs first value
```

So, in the header rendering method:

```
 WhateverRoot >> renderHeaderOn: html
     html div id: 'header';
         with: [
             html heading: self title.
             html div id: 'loginBox';
                 with: [html render: loginComponent]
             ]
```

Now it is our graphic designer's task to give the component the right size and style, for the moment, we will add the next CSS code to our style[4]:

```
#loginBox{
  position:absolute;
  right:15px;
  top:10px;
  margin: 5px;
  padding:3px 5px 5px 5px;
  background-color: #d7eef4;
  border:1px solid;
  border-color: #2c89a0;
  width: 132px;
  font-size: 10px;
}
```

[4]Located at the instance side of **WhateverRoot** or in an external file, depending on the chosen implementation method

```
input.btn{
  margin-top: 2px;
  color: #2c89a0;
  background-color: #d7eef4;
  border:1px solid;
  border-color: #2c89a0;
  float: right;
}

input {
  font-size: 10px;
  margin-top: 2px;
  color: #2c89a0;
  background-color: #d7eef4;
  border:1px solid;
}
```

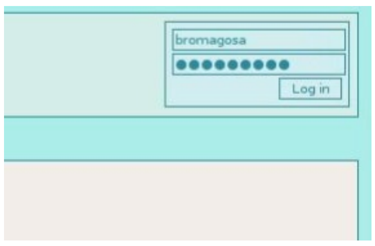


Figure 11: *The new login component embedded into the website's header*

6 3rd CHAPTER - A (Very) Brief Introduction to Magritte

6.1 What Is Magritte

Magritte is described in a wide variety of terms, but for the use we are going to make of it here, *meta-component* might be the one that fits best. **Magritte** is actually a very complex framework, as his author points:

> Most applications consist of a big number of model- or so called domain-objects. Building different views, editors, and reports; querying, validating and storing those objects is very repetitive and error-prone, if an object changes its shape frequently.
>
> Magritte is a fully dynamic meta-description framework that helps to solve those problems, while keeping the full power to the programmer in all aspects. Moreover since Magritte is described in itself, you can let your users modify the meta-world and add their own fields and forms without writing a single line of code.[5]

For us, **Magritte** will be a way of dealing with web forms without having to manually build them like we did previously on chapter **Sessions And Users**, but instead describing the objects we want to be filled with information and letting **Magritte** create the web component for us. This chapter should serve as a quick introduction to its usage and basic features, for a detailed documentation it is advised to read Lukas Renggli's tutorial. [6]

We will be using **Magritte** to build some simple forms from now on, which is why it is important to briefly introduce it now. Being **Magritte** abstract in itself, we are going to use an example of a small bookstore when showing code, since abstracting an abstraction would be too far from illustrative.

This section is based on a post by Ramon Leon [7]

6.2 Descriptions

A description is *a statement that represents something in words*, and that is exactly what a description does in Magritte.

The first class we need to familiarize with is **MADescription**, we will always describe our objects by using subclasses of **MADescription**, but not in the usual **Smalltalk**-like way of overriding them, but by creating instances of them in our domain objects class side (in general). By doing so, **Magritte** will have a way to build forms out of these descriptions and link each one of their fields to the corresponding methods of the described objects. Note that, being this a web development tutorial, we are only going to talk about **Magritte**'s web form building features, but it should be pointed out that this framework can

[5] Lukas Renggli, *http://www.lukas-renggli.ch/smalltalk/magritte*

[6] Lukas Renggli, http://www.lukas-renggli.ch/smalltalk/magritte/tutorial.pdf

[7] Using Magritte with Seaside, *http://onsmalltalk.com/using-magritte-with-seaside*

also deal with morphs by default and was built as open as can be in order to make it suitable for whatever other purposes.

For instance, if we had a class named **Book** with the methods **addTitle**, **addBarcode**, and **addPurchaseDate** we could create the following descriptions:

```
Book class >> descriptionAddTitle
    ^ (MAStringDescription new)
        selectorAccessor: #addTitle;
        label: 'Title';
        priority: 1000;
        yourself


Book class >> descriptionAddBarcode
    ^ (MANumberDescription new)
        selectorAccessor: #addBarcode;
        label: 'Bar Code';
        priority: 900;
        yourself


Book class >> descriptionAddPurchaseDate
    ^ (MADateDescription new)
        selectorAccessor: #addPurchaseDate;
        label: 'Purchase Date';
        priority: 800;
        yourself
```

The description methods must return an instance of a **MADescription** subclass (which one, as shown above, depends on the data types). We should also send some messages to that instance in order to set up the description, namely:

selectorAccessor is, as its name points out, the selector of the accessor method we want to link with this description.

label is the label that will be shown in the form, but abstractly speaking would be the name of the described attribute.

priority indicates the order in which the attribute will be requested in the resulting form. The abstraction here would be immediate: the priority of a descriptor represents the order in which we would preferably enumerate it as an element of the object it belongs to.

Once we have our descriptions, we can ask **Magritte** to build a form in two different ways depending on our needs. If we want to build a form requesting every field described, we can just do:

```
Book >> buildComponentFor: aBook
    ^aBook asComponent addValidatedForm
```

We notice one particularly odd thing about the code above, which is the **addValidatedForm** message. This message wraps the form into a **decoration**; this leads us to introduce two important concepts in Magritte and Seaside in general:

Memento: A **memento** can be described as a temporary copy of an object, which will not be applied to the actual object until we explicitly ask for it.

Decoration: A **decoration** is a way to add functionality by assembly. In **Magritte**, form decorations act as intermediaries between the form, the memento and the actual object.

In the code above, **addValidatedForm** is a particular decoration that will validate the information before sending it to the object via its accessors. This decoration deals with a memento of the object the form will be applied to. When the user fills in the information, the form decoration validates it against the memento, and if everything works out, changes are applied to the actual object.

Since most of the time we won't need every one of our described fields to be shown in the form, we will more often be using a composite representation:

```
buildComponentFor: aBook
    ^((Book descriptionAddBarcode,
        Book descriptionAddPurchaseDate) asComponent on: aBook)
            addValidatedForm;
            yourself
```

Magritte, as pointed out before, is a very complex framework which takes long time and practice to master, which is why we have only introduced the aspects we are going to use in this manual. Every single form used in this manual could be defined and rendered by using **Magritte**, though it would complicate it way too much for the objectives we intend to accomplish, which is why we will keep on coding some forms manually when it comes to specific ways of asking user input or different validating procedures than the ones coming with **Magritte** by default.

7 4th CHAPTER - Adding Dynamic Components

Dynamic components are those holding contents that can be modified by interacting with a website through a browser. These modifications must be persistent, which is why we will be both building a dynamic component and showing how to map it to our already existing database.

The dynamic component we are going to introduce is a simple blog based on a screencast by Ramon Leon [8].

7.1 A Simple Personal Blog

A blog is essentially a collection of posts ordered by date of entry. Each of those posts belong to a user and should ideally have a date and a collection of comments also ordered by date of entry.

From this point of view, it seems pretty reasonable to understand a blog as an **OrderedCollection** of posts, being a post an object with its title, text content, user, date and an **OrderedCollection** of comments. In its turn, a comment will be an object with some attributes, such as a title, the name of the person who wrote it and, of course, its text content.

Note that, by following a very similar procedure to the one we introduce here, we could very easily build a fully-editable website taking in account user roles to decide which sections can be edited by each group of users.

Let us begin by creating the two objects we have just described.

7.1.1 Posts And Comments: The Objects

As we stated before, a post is an object holding a title, a content, a user, a date and some comments:

```
Object subclass: #WhateverPost
    instanceVariableNames: 'title content user date comments'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

And the only methods it needs, as it usually happens with model objects, are the accessors.

In this case, though, we are going to lazy-initialize comments to be an **OrderedCollection**:

```
WhateverPost >> comments
    ^comments ifNil: [comments := OrderedCollection new]
```

Since we already introduced **Magritte**, we can now save lots of time by implementing descriptors for **WhateverPost** in its class side:

```
WhateverPost class >> descriptionTitle
    ^(MAStringDescription selector: #title
    label: 'Title: ' priority: 10)
        beRequired;
        yourself.

WhateverPost class >> descriptionContent
    ^(MAMemoDescription selector: #content
    label: 'Content: ' priority: 20)
        beRequired;
        yourself.
```

user and **date** are going to be set automatically by us when the form is submitted, so we can forget about describing them.

As for comments, the procedure is pretty much the same:

```
Object subclass: #WhateverComment
    instanceVariableNames: 'commenterName title comment'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

Just like before, we automatically create accessors for all instance variables and turn to the class side to define our descriptors:

```
WhateverComment >> descriptionCommenterName
    ^(MAStringDescription selector: #commenterName
    label: 'Name: ' priority: 10)
        beRequired;
        yourself.


WhateverComment >> descriptionTitle
    ^(MAStringDescription selector: #title
    label: 'Title: ' priority: 20)
        beRequired;
        yourself.


WhateverComment >> descriptionComment
    ^(MAMemoDescription selector: #comment
    label: 'Comment: ' priority: 30)
        beRequired;
        yourself.
```

And that should be everything we need to begin building the blog view.

7.1.2 The Blog View

The blog view will have to show a list of posts with their respective comments, and that list should be reversed so that newer posts come first.

Let us begin by creating a new component:

```
WAComponent subclass: #WhateverBlog
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

Next step is to write **WhateverBlog**'s rendering method:

```
WhateverBlog >> renderContentOn: html
  self session database class blogRepository
     reverseDo: [:eachPost | html
       div: [html heading: eachPost title level: 2;
           paragraph: (eachPost user firstName asString, ' ',
             eachPost user surname asString);
           paragraph: eachPost date;
           text: eachPost content.
       html
           div: [html strong: 'Comments'.
             eachPost comments
                do: [:eachComment | html heading: eachComment
                      title level: 4;
                    paragraph: eachComment commenterName asString;
                    text: eachComment comment]].
             html anchor callback: [self addCommentTo: eachPost];
                with: 'Add Comment'
    ]].
  html anchor on: #newPost of: self
```

Notice we sent a message that our database doesn't understand yet (**blogRepository**); this will be covered in the next subsection. For the moment let us focus in the methods lacking to the current class, **WhateverBlog**.

```
WhateverBlog >> newPost
    | aPost |
    aPost := self call: ((WhateverPost new asComponent)
        addValidatedForm; yourself).
    aPost ifNotNil:
    [
        aPost
            date: Date today;
            user: self session user.
        self session database class blogRepository add: aPost.
        self session database saveImage]
```

This method makes use of **Magritte**'s advantadges, notice how we automatically build a form that will request the user's input, validate it and store it in the corresponding object (remember how tedious was to manually build input forms for components as **WhateverRegister** or **WhateverLogin**).

As for the last lacking method:

```
WhateverBlog >> addCommentTo: aPost
    | aComment |
    aComment := self call: (WhateverComment new asComponent
        addValidatedForm; yourself).
    aComment
        ifNotNil:
            [aPost comments add: aComment.
            self session database saveImage]
```

Both methods look very short, similar and easy to understand thanks to using **Magritte**, however, if we wanted to do something more complicated, like embedding a rich text editor (like the one **ShoreComponents** provides) into our blog post editor, **Magritte**'s way to do it would become fairly difficult.

7.1.3 Database Mapping

First thing we need to do is add a new class variable to our database, which will hold the whole collection of posts in the same way it already does with users:

```
Object subclass: #WhateverDatabase
    instanceVariableNames: ''
    classVariableNames: 'BlogRepository Users WriteMutex'
    poolDictionaries: ''
    category: 'Whatever'
```

Actually, the way to deal with blog entries will be pretty much the same we used to deal with users, the next method in the class side looks exactly the same as the one we wrote for the **Users** class variable:

```
WhateverDatabase class >> blogRepository
    ^ BlogRepository ifNil:
        [BlogRepository := OrderedCollection new].
```

While, usually, mapping data to a database can become a pretty complex task, in our case this is all we are going to need. Now we only need to embed our new blog component to our root.

7.1.4 Embedding the Blog to our Root Component

To embed this component we are going to follow exactly the same procedure as before, when we embedded both register and login components.

First of all, we add an instance variable to our root:

```
WAComponent subclass: #WhateverRoot
    instanceVariableNames: 'aCoupleOfTabs selectedTab
        registerComponent loginComponent blogComponent'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Whatever'
```

We add it to its children:

```
WhateverRoot >> children
    ^Array with: registerComponent with: loginComponent
        with: blogComponent.
```

And in the initialization method, we assign it to a new **#WhateverBlog** and add a tab for it:

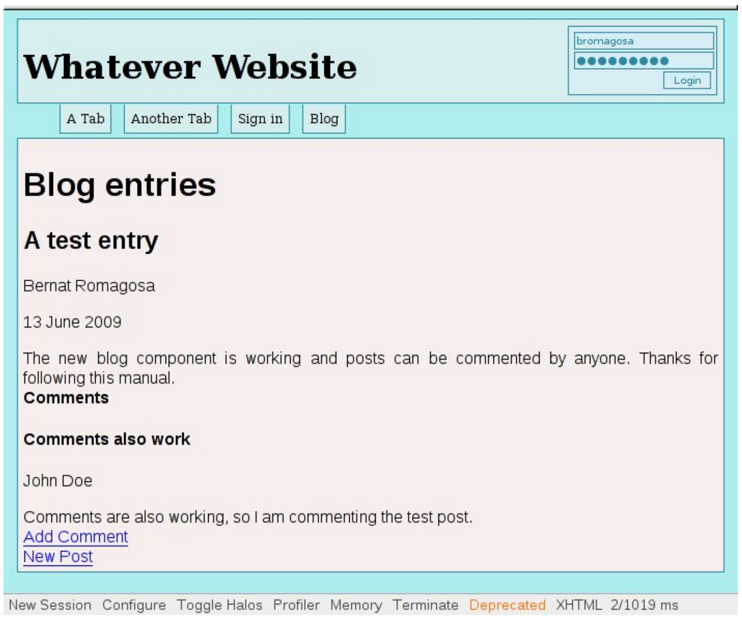
```
WhateverRoot >> initialize
    super initialize.

    loginComponent := WhateverLogin new.
    registerComponent := WhateverRegister new.
    blogComponent := WhateverBlog new.

    aCoupleOfTabs := OrderedCollection new
    add: 'A Tab' -> (Array
        with: 'A section' -> WhateverSection new
        with: 'Another section' -> WhateverAnotherSection new);
    add: 'Another Tab' -> (Array
        with: 'Yet another section' -> WhateverYetAnotherSection new);
    add: 'Sign in' -> (Array
        with: 'Sign in' -> registerComponent);
    add: 'Blog' -> (Array
        with: 'Blog entries' -> blogComponent);
    yourself.
    selectedTab := aCoupleOfTabs first value
```

Our blog is now ready and working.

We introduced the main concepts and procedures needed to get started in **Seaside**, from this point on we can already build anything we want from scratch, and since reutilization of components is -as we have seen- a very simple task, we can profit from the work done by others to grow our website or web application as big and complex as we want.



bromagosa
●●●●●●●●●
Login
A Tab
Another Tab
Sign in
Blog

Whatever Website

Blog entries

A test entry

Bernat Romagosa

13 June 2009

The new blog component is working and posts can be commented by anyone. Thanks for following this manual.

Comments

Comments also work

John Doe

Comments are also working, so I am commenting the test post.

Add Comment

New Post

New Session Configure Toggle Halos Profiler Memory Terminate Deprecated XHTML 2/1019 ms

Figure 12: *A screenshot of the simple blog working*

8 Conclusions

Developing in **Seaside**, even when just starting to learn the language and the framework, has proved to be highly productive and consistent with the object-oriented programming paradigm. Once basics are learnt, scaling becomes fast and easy and can be substantially boosted up by reusing ready-made components and libraries.

Seaside proves that web development can also be pure object oriented, and -as opposed to what a first glance to the framework and its methodology could infer- performance, speed and scalability are not sacrificed for the sake of programming comfort.

Seaside is usually introduced as a way to build web applications, and this is what most manuals and tutorials teach. The present project intended to check whether **Seaside** was also a valid approach for building common websites -such as **Edutech**'s-, which showed up true.

However, the learning curve for this framework can be fairly rough for newcomers. Even though during the last years the lack of documentation is being gradually solved, there is still a big gap to cover. One of the objectives of this project was to help making up for this problem, and I think and hope that the resulting manual will certainly be of great use to beginners and people interested in a different way of developing websites.

Glossary

accessor	Creating accessors automatically, 23
aCoupleOfTabs	WhateverRoot instance variable assignation and explanation, 17
aCoupleOfTabs	WhateverRoot instance variable, 15
addCommentTo	WhateverBlog method, 41
addUser	WhateverDatabase method, 25
background	Backgrounding the procedure of saving a Squeak image, 25
blog	Term definition, 38
blogComponent	WhateverRoot instance variable, 41
BlogRepository	WhateverDatabase class variable, 41
blogRepository	WhateverDatabase class method, 41
buildComponentFor	Magritte method, 36, 37
canBeRoot	WhateverRoot method, 15
children	Embedding a component into another component, 29
children	Rendering our root component's children, 18
children	WhateverRoot method, 29, 32, 42
comment	WhateverComment instance variable, 39
commenterName	WhateverComment instance variable, 39
comments	WhateverPost instance variable, 38
comments	WhateverPost method, 38
content	WhateverPost instance variable, 38
CSS	An example of a simple Cascading Style Sheet for our website, 20
CSS	How to read an external CSS file, 21
CSS, login box	An example CSS for our login box, 33
database	Creating an image-based database, 24
database	Methods for saving data, 25
database	WhateverSession instance variable, 26
date	WhateverPost instance variable, 38
decoration	Magritte concept, 36
description	Magritte concept, 35
description	WhateverRoot method, 15
descriptionComment	WhateverComment class method, 39
descriptionCommenterName	WhateverComment class method, 39
descriptionContent	WhateverPost class method, 38
descriptionTitle	WhateverComment class method, 39
descriptionTitle	WhateverPost class method, 38
dynamic component	Term definition, 38
Edutech	A teaching/learning/development project hosted by Citilab, 12, 15
email	WhateverUser instance variable, 23

References

[1] **Scorer**
Romagosa Carrasquer, Bernat
http://sourceforge.net/projects/scorer

[2] **An introduction to Seaside**
Michael Perscheid, David Tibbe, Martin Beck, Stefan Berger, Peter Osburg, Jeff Eastman, Michael Haupt and Robert Hirschfeld
ISBN 978-3-00-023645-7
2008
http://www.hpi.uni-potsdam.de/hirschfeld/seaside/tutorial

[3] **A Smalltalk by the Seaside - Screencast**
Romagosa Carrasquer, Bernat
http://www.vimeo.com/5130619

[4] **Edutech**
Romagosa Carrasquer, Bernat
http://mestralserver.no-ip.org:8080/seaside/edutech (temporary URL)

[5] **CSS by Example**
Steve Callihan
ISBN 0-7897-2617-3
2004

[6] **Personal Blog**
Renggli, Lukas
http://www.lukas-renggli.ch/smalltalk/magritte

[7] **Using Magritte With Seaside**
Leon, Ramon
September the 10th, 2007
http://onsmalltalk.com/using-magritte-with-seaside

[8] **How to Build a Blog in 15 Minutes with Seaside**
Leon, Ramon
November the 20th, 2006
http://onsmalltalk.com/screencast-how-to-build-a-blog-in-15-minutes-with-seaside