From REST to Rich: Retargeting a DSL to Ajax

Master's Thesis, 21 September 2007

Jonathan Joubert

From REST to Rich: Retargeting a DSL to Ajax

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Jonathan Joubert

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Finalist IT Group b.v.
Groothandelsgebouw A4.191
Stationsplein 45
Rotterdam, The Netherlands
www.finalist.com

From REST to Rich: Retargeting a DSL to Ajax

Author: Jonathan Joubert
Student id: 1047477
Email: j.joubert@gmail.com

Abstract

A domain-specific language (DSL) should be specific enough to be useful and concise, yet general enough to cover its entire domain. In designing a DSL, there is a risk of designing it too close to the original target framework in mind.

In previous DSL engineering research, a DSL for modelling and generating web applications was created. This WebDSL generates an application with a traditional RESTful front-end, consisting of JavaServer Faces pages and Seam session beans. This thesis describes the process of creating a mapping from WebDSL to Echo2, a Rich Internet Application framework by NextApp.

In retargeting to another framework with another interaction style, WebDSL's capability of modelling web applications in general is judged. Also, modelling rich functionality, for which new methodologies are needed, is looked into.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Eelco Visser, Faculty EEMCS, TU Delft
Company supervisor:	Ing. Nico Klasens, Finalist IT Group b.v.
Committee Member:	Ir. Bernard Sodoyer, Faculty EEMCS, TU Delft

Preface

Nico and Eelco, thank you for all your help. You have been very open and approachable, giving advice and asking the right questions. You both know *too* much about your fields of expertise: web application development (Nico) and program transformation (Eelco). This made the project especially challenging to me – which is a good thing!

Rob, thanks for all your advice and ideas. Ma, pa, Joëlle and Geraldine, thanks for your support and maybe I will come visit you more often now! B'shizzle, B'wijs, B'liieve, Das Boot and Ariston, thanks for all the distraction in the form of serious meetings and serious fun. Finally, KM39 b'vo!

Jonathan Joubert
Delft, The Netherlands
21 September 2007

Contents

Chapter 1

Introduction

A domain-specific language (DSL) should be specific enough to be useful and concise, yet general enough to cover its entire domain. In designing a DSL, there is a risk of designing it too close to the original target framework in mind.

In previous DSL engineering research, Visser has created WebDSL: a DSL for modelling and generating web applications. WebDSL, described thoroughly in [36], generates an application with a traditional RESTful front-end, consisting of JavaServer Faces (JSF) pages and Seam session beans. This thesis describes the challenges faced in the process of creating a mapping from WebDSL to Echo2, a Rich Internet Application (RIA) framework by NextApp. In retargeting to another framework with another interaction style, WebDSL's capability of modelling web applications in general is judged.

To be clear, WebDSL creates a complete application, using Java Persistency API (JPA) Entities for database access. The new mapping to Echo2 will also use these Entities, so nothing is changed in that part of the WebDSL transformation engine.

Finally, a RIA framework, or Ajax framework, mixes web and desktop application functionality, creating a *rich* user interface. New methodologies are needed for modelling RIAs, as concluded by [28] after analyzing existing web and hypermedia modelling approaches. This work looks into such modelling.

1.1 Research questions

1. **Is it reasonably feasible to generate Echo2 code from WebDSL?**
Not being able to create a mapping to another framework than the original target, while staying in the domain the DSL claims to cover, means that the DSL is designed too close to the original target framework. And if it so happens, what needs to be added to or changed in the DSL to make it compatible?

Preferably, one should be able to write DSL code once and then choose which transformation to apply. If that is the case, the DSL is indeed independent of the implementation, as far as the two frameworks differ that is.

2. **Is it, at the same time, reasonably feasible to generate good RIA code from WebDSL?**
Even if generating Echo2 is possible, does the use of WebDSL limit rich functionality? Is it even possible to generate any rich features from WebDSL? And what is a RIA anyway?

During the retargeting process, care should be taken to note whether the mapping is really RIA-focused, or whether it is simply following the current DSL. In other words, is the model in WebDSL the right one for a RIA, or would one wish to model certain concepts differently? An example is whether to model complete pages, as is currently done in WebDSL, or whether one should model only changes to the current screen.

3. **Does WebDSL cover web application modelling?**
If generating rich Echo2 code is feasible, does WebDSL now cover its domain? What are the core elements of a web application and which are specific to the interaction pattern? What is the difference between a RESTful application (like JSF and Seam[1]) and a RIA?

4. **Can we learn anything new for DSL engineering from this retargeting?**
Can this work add insights to Visser's research into how to create a DSL? Which steps should be taken to make sure a DSL is not designed too close to a specific target framework?

1.2 Context of the thesis

This thesis is part of the Model-Driven Software Evolution (MoDSE) project[2] at the EEMCS faculty of the Delft University of Technology. MoDSE has the goal of developing a systematic approach to model-driven software evolution. Modelling and generating a web application with WebDSL, and especially the engineering of WebDSL itself, work towards this goal.

Preceding this thesis was a literature study, titled *Model-Driven Development, Deployment and Maintenance of Web Applications*; see [13]. A number of paragraphs from it, especially those with background information on frameworks, are also used here, appropriately cited.

The work was carried out at Finalist IT Group[3], a software company based in Rotterdam that calls itself specialist in open source and Java technologies. They have created and worked on a number of open source projects, among which JAG (see Similar Work) attracted me to them. Finalist is especially strong in web application development and content management. From this project they may come to understand web applications even better.

1.3 Similar work

Many initiatives exist that deal with modelling some form of user interaction and Entities, and generating a basic web application from these models. Here follows a non-exhaustive

[1] Technically, adding JavaScript could make JSF rich.
[2] `http://swerl.tudelft.nl/bin/view/MoDSE/WebHome`
[3] `http://www.finalist.com`

list of some of the more ambitious or interesting projects.

Finalist's Java Application Generator (JAG[4]) is an open source tool, mostly used for prototyping. It takes a database description or UML class diagram as input, generating code based on code templates. It is aimed at Java EE (Enterprise Edition), but it is retargetable to, e.g., .NET if someone would write the appropriate templates. JAG generates the complete structure of a Java EE application, easing its further development into a full-scale application. [13]

ArcStyler[5], by Interactive Objects[6], is one of the two proprietary, industry-leading tools for generating code from UML models, the other being OptimalJ[7]. A user can add 'marks' (annotations) to add necessary platform-specific information to the models. ArcStyler is very pluggable and extensible, with users being able to create cartridges or write scripts within the tool. For example, users can define OCL (Object Constraint Language) rules that the tool can use for model validation. It also has cartridges offering harvesting support, by either reverse engineering or by reading EJB (Enterprise JavaBeans) deployment descriptors. [13]

In [16], UML-based Web Engineering (UWE) is presented, showing that UML is powerful enough to cover web application modelling. A lightweight UML profile is used, enhanced with stereotypes and constraints. The UML diagrams used are use case diagrams for requirements specification, class diagrams for conceptual modelling, stereotyped class diagrams for navigation and presentation modelling, state chart and interaction diagrams for web scenarios modelling, activity diagrams for task modelling and deployment diagrams for documenting the distribution of web application components. [13]

These initiatives have in common that they are based on (one or more of) data, workflow and interaction models. They are not focused on the representation on the screen. One could say that their models push information to the front-end, instead of pulling information from the back-end, as WebDSL models do.

Furthermore, they are more flexible, but less simple and domain-specific, leaving more work to the developer. The flexibility does have its use, as it probably makes full-scale application programming more attainable. That is not to say that a full-scale application is not feasible with WebDSL, but then development would have to continue after code generation.

The following projects do focus on screen output.

Web Relational Blocks (WebRB[8]), by IBM, is an online visual editor and run-time environment, but it is still under development. The available demonstration application is very simple to use. WebRB allows visual composition of web pages, with visual links to database tables for accessing data. The change-execute cycle is extremely short, which is great. But WebRB lacks many necessary features. An if-then-else construct is not allowed, a table can have buttons in only one of its columns and the screen gets cluttered quickly. In all,

[4] `http://jag.sourceforge.net`

[5] `http://www.interactive-objects.com/products/arcstyler`

[6] Interactive Objects is a participant in MoDSE

[7] `http://www.compuware.com/products/optimalj`

[8] `http://services.alphaworks.ibm.com/webrb`

it provided some inspiration to this thesis, but as long as it misses so much functionality, WebRB will not need to be reckoned with.

Muller *et al.* introduce a metamodel for dynamic web page composition and navigation. It encompasses a business, hypertext and presentation model. They also present the action language Xion, that is based on OCL and Java and enables code generation from the models. [25]

The Web Modeling Language (WebML[9]) is an initiative that seems promising. It started a number of years ago and was updated to target RIAs. Three models are used in it. First, a data model, which is an Entity-Relationship diagram or a UML class diagram. Second, a visual hypertext model, consisting of pages with a number of predefined content units (e.g. lists of Entities) and of links between pages. The content units are especially interesting, incorporating much used data viewing and editing patterns. Third, a presentation model, whose specifics are not prescribed by WebML, leveraging standard approaches familiar to graphic experts. [5]

Concerning retargeting web applications to RIAs, [21] describes a process and a tool for extracting a navigational model and identifying user interface components, for creating a single-page Ajax interface. Some of their ideas, most notably the concept of *diff*'s between pages (see section 5.2), are used in this work.

Finally, the closest initiative is by [38]. There, a process is described for developing a software factory for web applications from a number of separate DSLs. The Web Scenario DSL models user interaction and page flow, the Data Contract DSL models Entities, the Service DSL models services (like editing Entities) and the Business DSL models business rules and methods. In these DSLs, C# is used to write the code that cannot be modelled easily, like the business methods.

1.4 Cases

Three cases have been identified for trying out the functionality of WebDSL and its retargeting. The first, Research group SERG, is well elaborated by Visser (see [36]) and many examples in this thesis are based upon it. The other two are new ideas for testing WebDSL.

1.4.1 Research group SERG

The Software Engineering Research Group (SERG) is the research group MoDSE belongs to. The case is worked out mostly in terms of domain modelling, incorporating users, persons, publications, projects, articles, conferences, etc. Some extras are: all kinds of associations are used (like lists of associated Entities) and articles have a list of authors whose order is important.

1.4.2 User login

Being able to create an application with simple access control functionality, is – in my opinion – a good way to test a framework's abilities, as well as one's understanding of the

[9] `http://www.webml.org`

framework. It requires

- getting user input;
- accessing data from the database;
- executing a comparison;
- guarded control flow: either going to the "logged in" page or the "login failed" page;
- some form of access control, to make sure that a user who has not logged in cannot access the "logged in" page;
- and state management, for remembering the user during a session.

1.4.3 Webmail

Many examples of RIAs are in the form of a web-based e-mail application. It seems that this is a good way of testing and showing off the functionality of a framework. Webmail needs functions like searching, sorting and sending e-mails, but also richer ones like autocompleting e-mail addresses while typing, saving drafts at intervals, checking for new messages automatically and having multiple tabs in one page.

1.5 Structure of the thesis

The rest of this work is set up in the following way. DSLs and the transformation language and tool set Stratego/XT are explained in chapter 2. Chapter 3 describes the current situation: the mapping from WebDSL to Seam/JSF. Rich Internet Applications are explained in chapter 4. Then, two chapters describe the retargeting process to Echo2: chapter 5 deals with the design of the process, including issues that would come up in mapping to any RIA platform, while chapter 6 deals with the specifics of the mapping to Echo2. Finally, chapter 7 discusses the outcome of the thesis and gives suggestions for improvements to the Echo2 mapping in particular and to DSL engineering in general.

Note that bits of programming code are displayed throughout this work and that most of it is simplified. What is left out, is either generally uninteresting, or beside the point.

Chapter 2

Domain-specific languages and Stratego/XT

Domain-specific languages fall in the category of model-driven engineering. This chapter first introduces the basic concepts of model-driven engineering. It then discusses domain-specific languages, as opposed to general-purpose programming languages. These two sections are mostly from the literature study in [13].

The chapter ends with an introduction to program transformation with Stratego/XT, a technique used to transform WebDSL into other languages, that can be interpreted or for which a compiler exists already.

2.1 Model-driven engineering

Model-driven engineering (MDE) focuses on software development at a high level — preferably in a domain-specific modelling language — and generating code and other artifacts from this high-level model [29].

Programmers have always sought ways to automate as much of the process of programming as possible. Code generation achieves this goal partially. It has especially received emphasis in compilers, generating code from a higher level language to a lower level language, or from a machine independent description to a machine specific one.

Abstracting to a conceptual level and generating code automatically, has three main advantages. First, higher productivity can be achieved because of the smaller semantic gap between the domain and implementation space. Developers can shift focus from implementation and platform-specific details, to the problem domain and its characteristics. Ideally, they are able to define the solution in terms of the problem domain, which narrows the gap even further ([15] p. 9). Moreover, code and other artifacts that are generated automatically are again good for productivity.

Second, this is a way of designing for change. Designing and working at a higher level of abstraction, makes changes easier to incorporate. Specifically, MDE makes a system more understandable, portable and modifiable [30]. Understandability, as has been explained already, is directly correlated to the higher abstraction level. Still, it depends on

the quality of the models, besides the problem that not every aspect of the system can be modelled (easily). Theoretically, portability of an MDE system to new platforms is easy, but in practice it depends on the availability of tools. Modifiability, in turn, depends on the type of change to be made. Low-level changes, for example, may not even be reflected by changes to the model, needing just as much effort as in traditional software engineering approaches. Furthermore, software does not only go through an evolution of changing requirements, but also in the way it is built or modelled, perhaps defining the model in another modelling language. For example, a code generator could be changed to adapt to a new target platform, as is done in this thesis. The MoDSE project has the goal of addressing these types of evolution.

Third, generating code has the advantage of consistency in implementation, because the code generator uses the same programming style consistently. This helps understanding code and also makes it possible to automatically perform error checking, functional requirements checking, quality checking and further transformation [29]. Especially code quality is interesting in this context. In the early days of compilers, many programmers were convinced that they could write better (i.e., more efficient) code than a compiler. And in many cases they were right. Nowadays, we rely on the quality of the compiled (generated) code, trading the possible small loss of efficiency in code against the gain of efficiency in creating the system at all. After defining a transformation once and testing it thoroughly, a code generator can be trusted to deliver quality code [17].

An interesting question in the domain of MDE is how far one can and should abstract from the details. Almeida *et al.* consider the concept of services of the desired system as a way of defining the most abstract useful model, in that a service describes *what* a system should do and does not prescribe *how* it should do it. [2]

Another problem is that although the way of modelling the static structure of a system is well-defined, for its dynamic behaviour it is not. Particularly, as far as code is not considered to be a model, it is difficult to model how a method should be implemented, without resorting to using programming code.

2.1.1 Round-trip engineering

Creating code from a design is a form of *forward engineering*. Going back from code to design, then, is called *reverse engineering*, or sometimes reverse architecting or software architecture reconstruction. The concept of going back and forth between design and code, finally, is known as *round-trip engineering*. The idea is that there is a neat one-to-one mapping between design and code and that changes to one are automatically reflected in the other.

Generally, for MDE, simply supporting repeated forward-engineering from model to code, instead of real round-trip engineering, is acceptable. The more code is generated and the less needs to be added or changed to the generated code, the better. Still, an MDE tool should provide support for these possibly conflicting generations. Ideally, no code needs to be adapted and so no real round-trip engineering is necessary.

2.1.2 Discussion

There are many articles on MDE, most of which sing praise of its methods and are hopeful that they will solve many of the problems in software engineering. And they have, to some extent. Creating a full Java EE application can be done a lot faster than before MDE, if only because of the productivity gains of generating the boilerplate code automatically. Using models has come into fashion a bit more, which eases and improves documentation of and communication about a project. As said before, MDE has three main advantages: productivity, ease of change and quality through consistency.

However, MDE has drawbacks too. In [22], five software engineering experts from, among other companies, Microsoft, Borland and Compuware, share their thoughts on MDE. Some of the negative sides they name are: that harvesting toward MDE is difficult and often inaccurate, while necessary in most projects; that MDE seems like nothing more than a marketing stunt for reviving the failed CASE tools of the 1980s; and that more domain-specific languages are needed for modelling the different views of a system, before MDE can be useful.

A lot of research and practice is still necessary for this technology to mature. And perhaps, without the right results, it will indeed meet the same fate as the CASE tools.

2.2 Domain-specific languages

Domain-specific languages (DSLs) differ from general-purpose programming languages (like Java or C) in that they use domain-specific terms and concepts, instead of offering a generic API. [8] proposes the following definition:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

What this means, is that a DSL focuses on representing and solving problems in a specific domain, instead of offering this support to any domain. The same point holds as for modelling at a higher level of abstraction, namely that working in the idiom of the problem domain improves understandability, maintainability, portability and communication. But more importantly, DSLs allow domain experts to better participate in the development process, because they can (or should be able to) understand the code. [8]

Furthermore, a DSL enables programs to be concise and, to a large extent, self-documenting, by using the appropriate terminology for describing or solving a problem [8]. Also, they enable reuse of software artifacts: a programmer can reuse domain concepts without having to analyse the domain thoroughly [19].

Examples of DSLs are: SQL for databases; HTML for web pages; YACC for parsers; and LaTeX, the document preparation system this document is written in. DSLs are usually declarative languages, nowadays often written in or converted to XML, although this does not have a human-friendly syntax at all.

[32] describes a number of DSL design patterns, one of them being the pipeline: using a family of DSLs sequentially. [7] describes MetaBorg, a method of embedding DSLs in a generic programming language and then having them assimilated, i.e., transformed to the generic programming language's API. Both the pipeline and language embedding were used in WebDSL, the embedding allowing the use of Hibernate Query Language (similar to SQL) inside WebDSL code.

DSLs have drawbacks too. The most prominent among them, are precisely because of the small applicability of DSLs. Developing one is relatively costly, because of the small amount of problems it can be applied to. Also, the developer will need both domain knowledge and software development knowledge, which is not common for one person to have [19]. Tool support is often more limited, needing *ad hoc* solutions to integrate DSL code with existing systems such as source browsers and debuggers [32].

Worse yet, there is the extra effort of training developers in a new language, especially considering that they realize that this language is not mainstream and so does not do much for their marketability. As for tool vendors, they are keen on convincing the market that general-purpose solutions are better, as this is better for *their* marketability. [3]

2.3 Program transformation

The domain of transformations deals with transforming a problem in one domain to one in another domain. Program transformation, then, is the transformation of a program, improving it relative to some cost function [35].

Program transformation is useful and necessary in two ways for DSLs. First, a DSL that is not interpreted directly, has to be transformed into runnable code to be useful. Generally this is called compiling, though compiling does not necessarily have executable code as its target; still, let us use the term here. An easier way of reaching runnable code is transforming the DSL into another language, for which a compiler exists already.

Another use of program transformation for DSLs is code generation. As described before, it is efficient to write little code and generate a lot. Code generation is used especially for creating boilerplate code.

2.3.1 Stratego/XT

Stratego/XT is a framework for creating program transformation systems. It consists of two main parts, namely the language Stratego and the tool set XT, and uses a number of other tools and languages. The general structure of a Stratego/XT transformation system will be explained first, followed by a more detailed explanation of its various parts. This is useful to understand the issues faced in this thesis and to be able to comprehend the implementation, or at least the bits of quoted implementation inside this work.

A transformation system follows a number of steps. These are shown in generic form in figure 2.1 (from the Stratego/XT manual[1]). A source file is parsed according to a syntax

[1] `http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-0.16/manual/tutorial-software-transformation-systems.html`

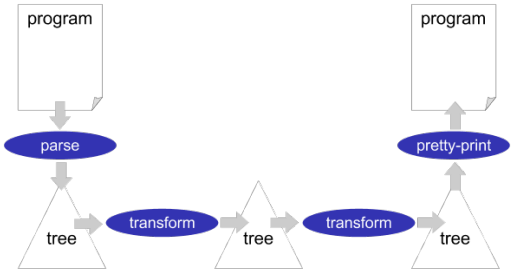


Figure 2.1: Transformation pipeline

definition, resulting in an abstract syntax tree that is then transformed a number of times and finally (pretty-)printed back to file.[2]

ATerm
The abstract syntax tree in a Stratego/XT system is made up of Annotated Terms (ATerms), instead of normal terms. This format allows internal term representation as well as providing support for persistence in files. It uses a binary exchange format that makes exchanging terms between applications fast and easy and it uses little memory. Generally, using ATerms allows an efficient and concise exchange of tree-like data structures. As such, the ATerm format allows a transformation system to be set up as a pipeline of many transformation tools that each perform transformations on the tree and pass on the result. [6]

SDF
The syntax files used in Stratego/XT are in SDF (Syntax Definition Formalism) format. SDF allows defining syntax concisely and purely declaratively. Furthermore, with SDF a grammar parser can be generated for any grammar, unlike Yacc, for example, that can only accomplish that for LALR input grammars. [37]

Most importantly, SDF is supported by *Scannerless Generalized LR Parsing*. Scannerless parsing is a technique that integrates a lexical scanner and a parser. It uses a combined definition of lexical and context-free syntax to be able to parse a grammar at once. This has a number of advantages, such as eliminating the complex interface between the scanner and parser, being able to define a language in one grammar and allowing better use of context information. It also creates a number of problems itself, such as needing a different form of disambiguity. This is dealt with by generalized parsing, a technique that allows ambiguity, essentially keeping record of all possible parse trees (a parse forest), at each step adding new possible trees and removing those trees that are not possible anymore. If at the end more than one parse tree is still possible, the parser should complain to its user. [33]

As an example of SDF code, the following is the simplified syntax of the Pane Location

[2]Readers common to compilers will recognize this pipeline and can keep the compiler paradigm in mind when reading about program transformation in this thesis.

DSL made during this work and discussed in section 6.1. It basically says that the input is a PaneLocation with one Container (a row or a column), which itself can contain more Containers recursively, having Switches as leaves. It furthermore disallows rows to directly contain rows and columns to directly contain columns. Finally, note that *context-free syntax* allows layout (as defined in the lexical syntax) to exist between any two (non-)terminals. For example, a line break may occur between "panelocation" and "{" in the input.

```
lexical syntax
    [a-zA-Z][a-zA-Z0-9\_]* -> Id
    [\ \t\n\r]             -> LAYOUT
    "//" ~[\n\r]* [\n\r]   -> LAYOUT

%% an Id may not be followed by more Id characters
lexical restrictions
    Id -/- [a-zA-Z0-9]

%% LAYOUT may not be followed by more LAYOUT
context-free restrictions
    LAYOUT? -/- [\ \t\n\r]
    LAYOUT? -/- [\/].[\/]

context-free syntax
    "panelocation" "{" Container "}" -> PaneLocation
    RowContainer                     -> Container
    ColumnContainer                  -> Container

    "row"    "{" RowContents+    "}" -> RowContainer
    "column" "{" ColumnContents+ "}" -> ColumnContainer
    ColumnContainer | Switch         -> RowContents
    RowContainer    | Switch         -> ColumnContents
    "switch" "(" Id ")"              -> Switch
```

Stratego

The program transformation language Stratego is based on the paradigm of *transformation rules* (rewriting rules) using *strategies* – hence, its name – for controlling these rules. A rule has the form `L : l -> r where s`, with label `L`, transforming the left-hand side `l` into the right-hand side `r` and succeeding when the computations (or conditions) in `s` all succeed (or hold). A strategy is a way of determining which rule to use. For example, a strategy could be to perform a certain rule on all terms in the tree, or to find any term for which the rewriting succeeds and then stop. [34]

As an example, take the following sentence in WebDSL:

```
row { "Hello " person.name }
```

This should print to the screen, next to each other, the Strings "Hello " and the name of the person, which should be a variable used in the page this sentence occurs in. The Seam/JSF

mapping uses the `elem-to-xhtml` rule to transform screen elements to JSF code. This rule is actually split up into many rules, each of which tries to match its left-hand side to the term-to-be-transformed. If a match is found, that rule is executed. If the `where` clause does not succeed, the rewriting is undone.

```
elem-to-xhtml : TemplateCall("row", [], elems) ->
    %>
      <tr><%= <elems-to-xhtml(table-cell-wrap)> elems ::*%></tr>
    <%
```

This rule changes the "row" term to an XHTML `<tr>` term; note that this rule does not have a `where` clause, so when a row is matched, it will always be transformed into a `<tr>` tag. It then performs the parameterized strategy `elems-to-xhtml(s)` on the row's contents:

```
elems-to-xhtml(s) = filter(elem-to-xhtml; s); concat
```

This strategy definition uses the rule `elem-to-xhtml` for transforming each element, followed by the application of strategy `s` on the transformed element and concatenating all transformed elements. In this case, the strategy parameter `s` is `table-cell-wrap`, which puts an element inside a `<td>` tag:

```
table-cell-wrap : elems -> %> <td><%= elems ::*%></td> <%
```

All in all, the original row is transformed into:

```
<tr><td>Hello </td><td><some_way_to_get_name_of_person></td></tr>
```

Syntax

Some of the syntax in the above example is explained here. For much more detail, see [34] or the Stratego/XT manual.

A strategy can be recognized by the equals (=) sign, while a rule uses the colon (:).

Two strategies separated by a semicolon (;) are executed in sequence, but if the first strategy fails, its result will be undone and the second strategy will not be executed.

The syntax `<s> t` applies strategy `s` to term `t`. Terms are often implicit however, as the above definition of `elems-to-xhtml(s)` shows: it is passed a list of terms to execute on, applying `filter` on this list and `concat` on the result of the `filter` application.

In transforming the row term to a textual term, Stratego needs to be able to recognize variables inside the text, to give them the correct value. The transformation from row to `<tr>` should not blindly print the text `<%= <elems-to-xhtml...etc`, but actually the resulting `<td>` elements. For that reason quotation and antiquotation marks are used. In this case, everything between `%>` and `<%` is considered XHTML text, except when the antiquotation mark `<%=` is encountered, which *escapes* back to Strategolevel (metalevel) and performs some Stratego function. Other antiquotation marks can also be used. To escape from a literal String, for example, the tilde (~) is used, as in `"Hello ~name"`[3].

[3] To get LaTeXto print the % symbol, a similar escaping mechanism is also needed: \%

Two other constructs, namely *metavariables* and *dynamic rules*, can be explained by the Echo2 transformation on the same row term:

```
add-components :
    TemplateCall("row", [], elems) ->
    bstm*
    |[
        Row x_row = new Row();
        x_parent.add(x_row);
        bstm2*
    ]|
    where x_row     := <newname> "row"
        ; x_parent  := <Parent>
        ; {| Parent : rules ( Parent := x_row )
           ; bstm2* := <filter-concat-warn(add-components|"")> elems
          |}
```

Here, the row is transformed into an Echo2 Row. In the SDF definition of the Java embedding used here, a metavariable is matched when `x underscore name` is encountered. `x_row` is such a metavariable, getting its value in the `where` clause, namely "row" followed by an integer, creating a unique variable name (in this program run). `bstm2*` is also a metavariable, for a list of block statements similar to the one used in this example: some Java statements between `|[` and `]|` symbols. In particular, `bstm2*` is replaced by the statements that are the result of the transformation of row's elements ("Hello " and person.name).

`x_parent` is another metavariable and it is set to the value of the `Parent` dynamic rule. A dynamic rule is created at a certain point and can be used until its scope is left. This can save context information for use lower in the tree. Here, `Parent` is set to the value of `x_row`, leading all elements inside the row to add themselves to `x_row`, their parent. When all elements inside the row have been transformed, the scope is left and the rule lost; in this case, a shadowed `Parent` rule will resurface, so that elements at the same level as the row will be added to the same parent.

Abstract and concrete syntax

In its transformation rules, Stratego supports both *abstract* syntax, e.g.:

```
For(name, sort, exp, elements}
```

and *concrete* syntax, e.g.:

```
for(name : sort in exp) { elements }
```

Writing concrete syntax makes some rules much more concise and/or readable. The row example above already uses concrete syntax, in that the term is transformed into concrete code, like `Row row23 = new Row();`, instead of into a much more complex Java term.

XT tool set
Finally, some of the tools in the XT tool set that can be of great assistance, are tools for parsing, pretty-printing, coupling Stratego transformation components to SDF parsers and compiling Stratego files to C code.

Chapter 3

WebDSL to Seam/JSF mapping

This chapter explains the mapping from WebDSL to Seam/JSF. First, Seam and JSF are described (from the literature study in [13]).

Then, the syntax and semantics of the DSL are explained, followed by the structure of the program transformation engine and a description of the generated code. In other words, these sections describe the source, transformation and target codes, languages and structures. In fact, the source code is hardly touched in the remapping, this being one of the design criteria (see section 5.1), so most of the newly programmed code is in the transformation rules, generating new target code (for Echo2). The description of the DSL given in this chapter is an overview; it should be enough to generally understand the implementation of the transformation system and to understand the retargeting to Echo2. Much more detail can be found in [36].

The chapter ends with a discussion of WebDSL's value.

3.1 Seam and JSF

Seam[1], by JBoss, is an open source web development framework. It is the 'seam' between the EJB3 state management layer and the JSF presentation layer, the two Java EE standards for their respective layers. It provides a unified component model, offering the use of the same components, *Plain Old Java Objects*, everywhere in the application; Seam calls this the 'one kind of stuff' principle.

Before going into more detail on Seam, it is useful to have some understanding of JSF first. JSF[2] is a framework for developing web application user interfaces. By default, it uses JavaServer Pages for its web pages. It offers an extensible component model for widgets and pages use data components directly to present data. Using data components directly, makes JSF very useful in combination with EJB, in that components can travel along the entire stack; this will become clearer in the explanation of Seam.

Seam applications are inherently stateful. To achieve this, Seam adds two scopes to the default Java EE page, request, session and application scopes. First, the *conversation*,

[1] `http://www.jboss.com/products/seam`

[2] `http://java.sun.com/javaee/javaserverfaces`

which is the default scope in Seam and is shorter than a session. It holds state during a conversation, which is a sequence of requests and responses to accomplish a user task. A user will typically have more than one conversation during a session. Seam even makes concurrent conversations possible, calling this *workspace management*. The explicit state handling of conversations eliminates many web application state management difficulties, like when a user clicks on the 'back' or 'refresh' button or submits a form twice. It also eases having multiple frames or windows open, interfacing with the same application.

Second, the *business process*, which is a long term scope for dealing with *business process management*. A business process can encompass multiple users, may take years to complete and can survive the application scope. It is defined in an XML configuration file and takes on the form of a state chart. This scope is not used in the context of this thesis, so we will not go into more detail on it.

In accordance to the EJB specification, Seam uses Entity beans for database Entities and Session beans for performing actions. Both these types of beans can be used directly from within a JSF page. An Entity that has been read from the database, can be acted upon by a Session and given to a JSF page, that can edit the Entity and give it back to some EntityManager to persist it (update it) in the database. This is what was meant before by components that can travel along the entire stack.

Seam uses dependency *bijection*, an extension of dependency injection, to configure the components' dependencies. The annotation `@In` signifies that Seam should *inject* the following attribute. Conversely, the annotation `@Out` signifies that the following attribute is *outjected* for possible use by other components. In the following example, a User attribute is first injected into the class from a scope, used and possibly edited and then outjected back to the scope, for use in other components. This is a much more concise and programmer-friendly solution than explicitly finding the User in the HttpSession and then storing it back there when finished:

```
@In  @Out User user;
```

Figure 3.1 (from the Seam manual[3]) shows the architecture of a Seam web application.

3.2 Source – WebDSL

WebDSL consists of two main parts, namely Entity and page definitions. These will both be explained here, including a short example of page elements definable in the DSL. Furthermore, the templating mechanism for pages is described, followed by syntactic sugar, the use of Hibernate Query Language for database querying and the imperative code that can be used in actions.

3.2.1 Entity definitions

The following is an example of three Entity definitions in WebDSL:

[3] http://docs.jboss.com/seam/1.1.6.GA/reference/en/pdf/seam_reference.pdf

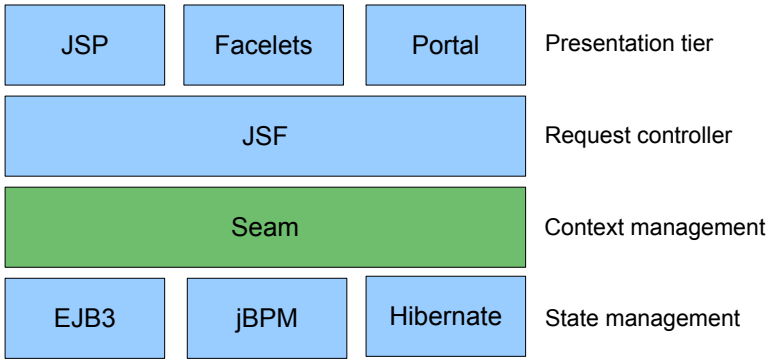


Figure 3.1: Web application architecture with Seam

```
Address
{
    street   :: String
    city     :: String
}

Person
{
    fullname :: String
    homepage :: URL
    address  <> Address
}

Publication
{
    title    :: String
    subtitle :: String
    year     :: Int
    authors  -> List<Person>
    abstract :: Text
}
```

So, an Entity definition starts with the Entity name, followed by a list of properties. These each have a name, a propertykind and a type. The propertykind can be either `::`, meaning "simple", `<>`, meaning "composite" (i.e. a Person has an Address and the Address is only useful when it belongs to a Person), or `->` meaning a "reference" to another Entity. The type is either another Entity or one of a few built-in types, like the Java type String or a String representation of an Url.

3.2.2 Page definitions

An example of a page using some of the Entities above is:

```
define page viewPerson(person : Person)
{
    title{text(person.fullname)}

    section
    {
        header{"Coordinates"}
        table
        {
            row{"homepage" navigate(url(person.homepage))}
            row{"address"  text(person.address.street)
                           text(person.address.city)}
        }

        header{"Publications"}
        publicationTitlesBy(person)
    }
}
```

This should render a page similar to the one in figure 3.2. Going over the elements from top to bottom, the first line gives the page its name, `viewPerson`, and it declares that the page needs a Person parameter. The next line puts the person's name in the title bar of the browser[4]. Then, a section is started, having an effect on the size of the headers inside: the deeper the header is located inside nested sections, the smaller it will be rendered. Next, a table with two rows is added. The first row starts with a literal text, followed by a link to the person's homepage.

The table and row elements are actually rather similar to the HTML they create. The only real effect the table element has, is that the elements in its rows are neatly lined up.

Finally, `publicationTitlesBy(person : Person)` is a method defined elsewhere, putting the titles of the person's publications in a list on the screen. It is described in section 3.2.5.

3.2.3 Templates

A page template can be defined, with hooks, so that a page can use the template and redefine some of the hooks. This is done extensively in the current use of WebDSL, as it saves many lines of code. For example, it can look like figure 3.3, where `logo()`, `sidebar()`, etc. are hooks that make redefinitions possible. Figure 3.4 shows the viewPerson page again, this time using the `main()` template and redefining its `body()` hook.

[4] Actually, the Seam/JSF mapping does not render the title, at least not yet.

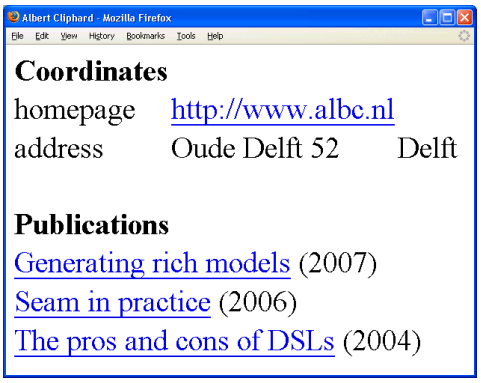


Figure 3.2: Page viewPerson

```
define main()
{
    div("sidebar")
    {
        logo()
        sidebar()
    }
    div("mainpane")
    {
        menubar()
        body()
        footer()
    }
}
```

Figure 3.3: Example template

```
define page viewPerson(p : Person)
{
    main()

    title{text(p.fullname)}
    define body()
    {
        section
        {
            //coordinates
            //publications
        }
    }
}
```

Figure 3.4: Page using template

3.2.4 Sugar

The DSL has core functionality and syntactic *sugar*, meaning elements with a higher level of abstraction. A relatively small core has the advantage of needing fewer transformation rules, making the transformation system more maintainable. More flexible constructs can then be defined at syntactic level and *desugared* to the core language, before being transformed.

For example, a link to another page is created with `navigate`, as seen before. A link is defined as follows:

```
navigate(e1) {text(e2)}
```

where `e1` is the page to go to and `e2` is the text to write to the screen as the link text. Adding a link to an URL and printing the same URL as the link text, would therefore be done with

```
navigate{url(e)} {text(e)}
```

For this construct, a sugared syntactic definition is added, namely that

```
navigate{url(e)}
```

is enough to create the link. It is desugared to the previous `navigate` construct, for which a transformation rule exists already.

Similarly, creating a link to view an Entity, can be done with:

```
output(person)
```

which is desugared to

```
navigate(viewPerson(person)) {text(person.name)}
```

3.2.5 HQL

An important feature of web applications is their ability to lookup data and present it to the user in different ways, for example sorted. SQL is a good DSL that already exists for this purpose. WebDSL uses an SQL derivative, namely Hibernate Query Language (HQL). This HQL code can be embedded inside "normal" WebDSL code, making possible the following definition of `publicationTitlesBy` (used in the page definition example above):

```
define publicationTitlesBy(person : Person)
{
    var orderedPublications : List<Publication> :=
      select pub from Publication as pub, Person as pers
       where (pers.id = ~person.id)
       and (pers member of pub.authors)
       order by pub.year desc;

    list
    {
```

```
        for(pub : Publication in orderedPublications)
        { listitem { output(pub) " (" text(pub.year) ")" } }
    }
}
```

3.2.6 Action language

Finally, some code is much easier to write imperatively. This is especially so for code that has to be executed when clicking a button. For this purpose an `action` method can be created, that can be called from an `action` button. Examples of code inside an `action` method are saving form input and going to another page. This code is written like Java code, e.g. `person.save();`, but only a few statements are as yet defined in WebDSL, keeping the DSL simple.

Adding the possibility of embedding general-purpose language code (like Java code) inside WebDSL would surely extend its capabilities, but at the cost of less control and portability. It is almost as easy and a lot more controllable to make it possible for WebDSL to call external libraries. But this functionality has not been added to WebDSL yet. [36]

3.3 Transformation – the program transformation engine

The transformation system consists of a parser, a pipeline of term transformations or checks, and a pretty-printer. Here, an overview is given of the strategies that perform transformations or checks. Note that not all strategies are named and that some are used more than once (for instance, desugar is called a number of times).

add-view-edit-pages-to-app
All Entity definitions are collected and for each, a number of pages is generated. For example, for Entity Person, the following pages are added to the application (*CRUD*[5] functionality, of which *delete* is part of the "View all" page):

- Create, for creating a new Person;
- View, for viewing a selected Person;
- Edit, for editing a selected Person;
- and View all, for viewing all Persons in the application's database.

typecheck
`typecheck` goes over every term in the source code, checking for semantic correctness (syntax checking is done by the parser). For example, for a link to another page, `typecheck` makes sure the other page exists in the application. If it doesn't, this will stop the transformation and give back an error message to the user.

[5] Create, Read, Update and Delete are the basic database functions

expand-page-templates
When a page calls a template, the template is expanded or inlined into the page. This effectively creates a complete page definition, with no dependencies on other pages.

desugar
`desugar`, as explained before, reduces high-level syntactic sugar to core syntax, to which other transformation rules may apply.

entity-to-java-Entity
This creates a JPA Entity for each Entity definition in the source code. It is pretty much a one-to-one conversion, but it does add the Entity's superclass' properties, a property `id` and some annotations for the database mapping.

page-to-xml
A page is transformed to three files, namely a JSF file for the presentation layer, a Seam Session bean and the bean's local interface. `page-to-xml` creates the JSF page, rendering DSL page elements to JSF elements.

page-to-java
`page-to-java`, in turn, creates the Seam Session bean and its local interface. The Session bean has the data and actions (methods) needed by the JSF. For example, a list of Entities can be held in the Session bean, to be accessed by the JSF.

Note that the retargeted code uses the Entities, but creates a different kind of front-end; it does not have the distinction between Sessions and pages, incorporating both into a single ContentPane class.

3.4 Target – the generated code

The Stratego code is compiled to C by the Stratego compiler. It can then be run on WebDSL code, generating code consisting of three parts: JPA Entities, Seam Session beans (and their local interfaces) and JSF pages. The Java files can then be compiled and deployed to a Java Servlet container or application server, together with the JSF pages and some configuration files, like database settings and Seam properties.

3.5 Discussion

According to [5], a web application consists of content, links and operations. In WebDSL, this distinction is not made explicitly, though the three elements do exist in it: page elements, navigation and actions respectively. [25] uses three models for web applications: the business model, the hypertext model and the presentation model. Again, WebDSL does not use these models explicitly, but Entities, navigation, and page elements and layout are similar concepts. Finally, of the Model View Controller (MVC) pattern, the model (Entities)

and view (page elements) are used in WebDSL, as is a decentralized controller (the Session beans).

All in all, it seems that WebDSL incorporates the basic building blocks of a web application. Perhaps that retargeting it to another platform and user interaction style encloses necessary constructs not defined in it. Or it may turn out that some constructs follow the Seam/JSF paradigm too closely.

Early results, from [36], show that WebDSL can reduce the amount of lines of code (LOC) to about 25%, or even to a tenth of that when the extra generated code (like in the View Entity pages) is counted. Note that this is for a relatively simple application, missing, for example, access control. Still, this counts as a good improvement over general-purpose languages[6].

No real qualitative comparison has been done, so what follows is my opinion. I find the DSL easy, flexible and expressive. However, the more "business logic" is needed, like access control, workflow management or business rules (e.g. constraints), the harder I believe it will be to express this in the DSL. Library calls should thus become supported.

[6]Even though JSF is not considered to be a general-purpose language, its LOC are counted as part of the target application's LOC.

Chapter 4

Rich Internet Applications and Echo2

This chapter describes Rich Internet Applications and focuses on the difference with RESTful applications. It also discusses Ajax, the approach to RIA development used by, among many others, Echo2. The chapter ends with a description of a couple of RIA frameworks, including Echo2.

4.1 What is a Rich Internet Application?

The term *Rich* Internet Application distinguishes it from a traditional, *RESTful* web application. It is useful to discuss this dichotomy between REST and RIA first.

In reality, the distinction between web applications types is not quite so sharp. It is a matter of definition and more types of web applications are indeed distinguished in literature. [31] speaks of five major schools of web application frameworks:

- Request-based (e.g. Struts), which has stateless requests, possible server-side sessions and is very close to the original CGI specification;
- Component-based (e.g. JSF), which abstracts the internals of request handling, uses components directly in the page to encapsulate logic and often has an event-handling paradigm;
- Hybrid (e.g. RIFE), essentially a combination of the previous two;
- Meta (e.g. Seam), a framework for integrating other frameworks;
- and RIA-based (e.g. Echo2), which are stateful and have user interface features commonly found in *fat* clients, like drag-and-drop.

REST, for its part, does not cover all traditional web applications. The term is introduced in [9], being an acronym for Representational State Transfer. It is an architectural style prescribing how a web application should work. It consists of a set of constraints for minimiz-

ing network communication and latency, while maximizing independence and scalability of its components.

Still, since its introduction in 2000, the term REST has become a way of describing a general web communication style of resources requested over HTTP. This thesis uses this broad definition of the term REST to mean traditional web applications, as opposed to RIAs, as is also done in, e.g., [20, 28].

Allaire, who coined the term *Rich Internet Application* in [1], complains about the limitations of traditional web applications in the form of a limited user interface and media content. They are adapted to the challenges imposed by the web's architecture, not being able to use the rich user interaction models that are common in desktop application programming. Allaire demands RIA models, that

> combine the media-rich power of the traditional desktop with the deployment and content-rich nature of web applications. [1]

Many authors have a view on the properties a RIA should have:

- Only transmit the data that needs to be transmitted (the changes to the screen), for faster communication. [1, 20]
- Integrate content and communication into a common environment. A host of technologies are needed to build a rich web application; this could use integration and simplification. Also, an integrated form of media handling should be used, instead of the multiple media players for audio and video. [1]
- A powerful and extensible object model, with event-handling, for better user interactivity. [1]
- Both online and offline application usage. Likewise, two-way notification-based communication should be possible over persistent connections. [1]
- Both server-side and client-side processing, to reduce communication and server load. [5]
- Both server-side and client-side caching and persistence. [5]
- A more seamless user experience[1], with immediate responses, smooth screen transitions and no unnecessary interruptions. A user should especially not be confronted with a blank screen, waiting for the application to respond. [26]
- Sophisticated interface behaviour, like drag-and-drop functionality and animations. [5]

Finally, most authors agree that the major difference between RESTful applications and RIAs is that REST deals with requests for resources (like pages), whereas a RIA deals with actions and reactions (events and changes) on the current state. [20]

[1] Has nothing to do with Seam!

4.1.1 Rich modelling in WebDSL

Even though the interaction model of RESTful applications is quite different from that of RIAs, the elements or components that are shown on the screen at any given time are more or less the same. These are the elements that should be defined in the DSL and most of them are indeed defined in it.

Some more complex concepts, like drag-and-drop functionality, data push, mnemonics and movies, are not defined yet however, although a `mnemonic` element is proposed in section 7.2.3. These concepts surely add richness to an application, so in future one should be able to model them in WebDSL too.

In retargeting the DSL, an interesting question is whether the current modelling of pages is adequate. In RIAs, one would typically model changes to the screen, instead of modelling complete pages. This issue is dealt with in section 5.2.1.

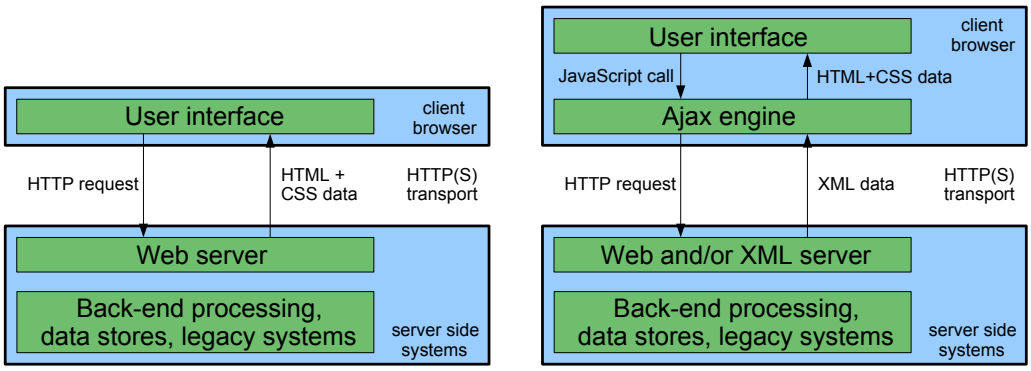
4.2 Web 2.0 and Ajax

Two hyped names for the RIA method are Web 2.0 and Ajax. Web 2.0 is generally used for some type of functionalities of an application (e.g. user-created content) instead of some kind of technical implementation *per se*, so we will not deal with it here. Ajax, however, is tightly coupled with RIAs. The following explanation of Ajax, and the rest of this chapter, comes from [13].

Ajax stands for Asynchronous JavaScript and XML. It was coined by Garrett in 2005, as a way of describing a new phenomenon in web user interfaces, namely rich user interfaces with low perceived latency. Ajax is not a technology in itself, but a mix of existing technologies and new ideas of web interface design. Besides JavaScript and XML, these technologies are XHTML, CSS, DOM, XSLT and the XMLHttpRequest. The last is a browser native object (like Window), first designed by Microsoft, that creates a communication channel between client and server. Figures 4.1a and 4.1b show the classic and Ajax web application communication models, respectively. Ajax uses asynchronous communication and only requests the needed information, providing the user with a more responsive interface. [10]

Using Ajax is really a design decision. Customers used to ask not to use JavaScript in web applications, because of security and accessibility issues; now, they ask for Ajax, for the interactive interface and because of the good momentum it has. Developers, for their part, have largely ignored JavaScript, because of portability and learning issues and because of limited tool support; these problems are mostly solved now. Finally, the architects decide whether to use Ajax or not, based on the application's requirements trade-offs. E.g., responsiveness as opposed to the guarantee of being found by search engines; learning client side scripting as opposed to programming on the server side in a programming language one knows; and the complexity of web programming at all as opposed to only using, e.g., a content management system. [14]

To be sure, other approaches to RIAs exist. Macromedia's Flash, e.g., creates vector-graphics-based animation programs and supports audio and video, which Ajax does not. On the other hand, it requires a plug-in to function and for many applications it is simply



(a) Classic (b) Ajax

Figure 4.1: Web application communication models

too powerful; the prototypical application of Ajax in Google Suggest[2], is an example where using Flash would be overdoing it. An advantage of Flash is that it is simpler to create than Ajax and it is closer to other tools used by graphic designers. [27]

4.3 RIA frameworks

4.3.1 Echo2

Echo2, by NextApp[3], is a Java framework for Swing-like programming on the web, creating Ajax applications that can run in any Java Servlet container. It is open source, but NextApp does sell a very useful Echo2 IDE.

It offers an object-oriented, event-driven, component-based user interface. The client/server interaction is abstracted from, so that a developer can actually think in conventional desktop application terms. Internally, the server keeps the application state and all functionality. The client can be thought of as a user interface for the server-side application. The client sends XML messages to the server, indicating user actions and events that have fired, so that the server state can be updated. The server, in turn, sends XML messages to the client, indicating the changes it should perform on its representation. [20]

An Echo2 Window is the main component on the screen and it can contain one ContentPane. A ContentPane can hold other ContentPanes and each ContentPane can contain components like Buttons and TextFields, but also ActionListeners or other attributes. The following code would render a "`hello <person.name>`" text:

```
public class MyScreen extends ContentPane
{
    public MyScreen(Person person)
```

[2] `http://labs.google.com/suggest` Check it out!
[3] `http://www.nextapp.com/platform/echo2/echo`

```
    {
        Row row = new Row();
        add(row);

        Label hello = new Label("hello");
        row.add(hello);

        Label name = new Label(person.getName());
        row.add(name);
    }
}
```

The framework does have a couple of drawbacks. First, documentation is very limited, though the developer is encouraged to look at Swing's documentation. Class names are usually the same as or similar to the ones in Swing. Still, as with Swing, one really needs to follow some tutorials, as the JavaDoc of any single class is almost useless: classes have dependencies without which they will not function. Adding, for example, a Table to a ContentPane will not render anything, before a TableCellRenderer is added as well. Second, Echo2 renders a complete page, so it is not possible to only use it for one half of the screen. And third, it is really meant for stateful applications, which in itself is a good thing, but the workflow is anchored in the code, so a page halfway through cannot be bookmarked. [12]

In all, programming in Echo2 is a great improvement over traditional web programming.

4.3.2 Google Web Toolkit

The Google Web Toolkit (GWT[4]) is a competing open source framework. It also allows writing Java similar to Swing and also offers a library of components that can be reused. The main difference with Echo2, is that GWT compiles the Java files to JavaScript code, which runs on the client, instead of keeping the user interface components on the server and communicating state changes. This is closer to the conventional Ajax style, using (X)HTML combined with JavaScript to create a rich interface. [20]

This approach has a number of advantages over the Echo2 approach. First of all, GWT needs less client-server interaction, as the client is able to act on some events itself. Second, and closely linked, it allows client-side processing, lessening the burden to the server. It also claims to render more quickly, leading to even better user interaction. [11]

It has some disadvantages too, however. Because the (JavaScript) code is run on the client, not the complete Java API is allowed to be used, for security reasons. That does not only effect the application in question, but also its links with other existing applications. Furthermore, debugging and testing, though mostly done in the original Java code, is more difficult in the running environment.

[4] `http://code.google.com/webtoolkit`

4.3.3 EchoPointNG

EchoPointNG is not a competing framework, but a library of components that extend and improve Echo2's. It is used in the Echo2 mapping in this thesis, though very little. For example, it offers a KeyStrokeListener that is more advanced that Echo2's and useful for WebDSL. Documentation, as with Echo2, is very limited. Some information, and the libs, can be found on the original EchoPoint web site[5] and its API can be found on another web site[6].

[5] http://echopoint.sourceforge.net
[6] http://docs.rakeshv.org/java/echopointng

Chapter 5

WebDSL to Echo2 mapping - Design

5.1 Approach

The general approach in mapping the DSL to Echo2 is based on the DSL development approach advocated in [36]. The approach used in this thesis is as follows:

1. Write a simple web application for Echo2 in Java.
2. Try to refactor this application so that it seems generic enough to be able to generate.
3. Write transformations in Stratego for generating basic page elements, like literal text output, the page title and links between pages, having the Java implementation as the goal of the transformation.
4. Write transformations for more difficult elements, like variable declarations and actions.
5. Discover new requirements for the mapping. Find a solution in Java, refactor the solution as in point 2 and write transformations towards it.
6. In case of a large issue, generate a large solution and then go on generating smaller and smaller parts of the solution.

As in the Seam/JSF mapping, an Entity definition will create CRUD code automatically. That is, the modeller will not need to explicitly model this CRUD code. However, we decided that one of the design criteria for this thesis should be that code that is thus generated automatically, should be generated first in DSL code. This DSL code is then transformed to target code. In other words, this extra code should not be generated in target code directly. Because of this criterion, the final code is always in terms of what the modeller is able to model. Also, when the mapping between source and target is changed, this extra code will also use the new mapping definition.

As an example hereof, a neat solution in Java for viewing and editing Entities was first created. Abstracting out almost all of the code, it became possible to generate a page for viewing and editing an Entity with very little Java code. However, the modeller has no

influence on this solution at all and changing the Stratego implementation has no effect either. Therefore, this solution is not correct and the existing solution of DSL-generated view and edit pages should be accommodated.

For the generation of Echo2 code, certain generic constructs are handy or even necessary. To this end, a number of Java utility classes are created, especially extensions of Echo2 components. Some of these will be explained where appropriate in this text and all are described in appendix A.

Finally, a major design criterion is to try to keep the DSL intact, without giving up rich functionality. It is precisely this intention that makes this an interesting research project. In designing a DSL it is important that it is generic to its domain of application. This is not easy to accomplish, especially when one has a specific target language in mind already, like Seam/JSF in the case of WebDSL. At the same time, as said before, care has to be taken not to adapt a solution to the DSL too much and to keep checking that the model is also good for this new target language.

Still, it should be stressed that the constraining factor to rich functionality is not so much the model (WebDSL), as it is the fact that the code should be generic enough to be able to generate it. In other words, a DSL specifically suited to RIA still might not be able to model and generate rich features easily.

When implementing a RIA, a couple of issues will always need to be addressed. These issues will now be described, including the general design of their solutions. The specific solutions and detailed problems in the Echo2 implementation are addressed in the following chapter.

5.2 Page deltas

One of the defining mechanisms in RIAs, is the ability to update only part of a page on a user's request. HTTP traditionally replaces a page completely on each request. Often (a large) part of the page the viewer sees is the same as the previous page, so the RIA method is faster and more responsive, which is precisely the *rich* experience.

Delta (difference) encoding and delta updating – only sending the delta between the old and the new values – is a concept often used in Computer Science. MPEG 1 (1989) had such a scheme already, but it was not until the mid 1990s that the idea was conceived to use it in making HTTP more efficient. At first, this was mostly used to update web browser caches. Pages do not change often – especially the static information pages most traditional web sites use – and when they do change, the new page is often only partly different from the old one. Sending only the delta over the network is more efficient than sending the complete new page. [23]

This research lead to the Request for Comments 3229, "Delta encoding in HTTP" [24], suggesting to make delta encoding a standard option in HTTP (which is still not the case!).

What should happen is that when a user goes from one page to the next, only those parts of the screen that change should be sent over the network and rendered by the browser. The delta (*diff*) between the current and the next page is derived and only this change is

communicated. [21]

This delta can be the exact HTML code that has changed from the old to the new version. The solution in this thesis is a bit coarser: a page is split up into a number of ContentPanes, each of which is either kept or completely replaced upon a page change request. This is much easier than going over each component on the page to determine whether it should be kept.

In designing this solution, the idea of *switch* nodes is adapted from [18]. In their work on the DaVinci web engineering framework, the authors predefine a hierarchical GUI tree, with the complete set of views in the application; one page is then the rendered version of the current state of the view tree. In figure 5.1 a simplified such GUI tree is shown, in which the "body" switch node can switch between the `homepage` and `addressbook` views.

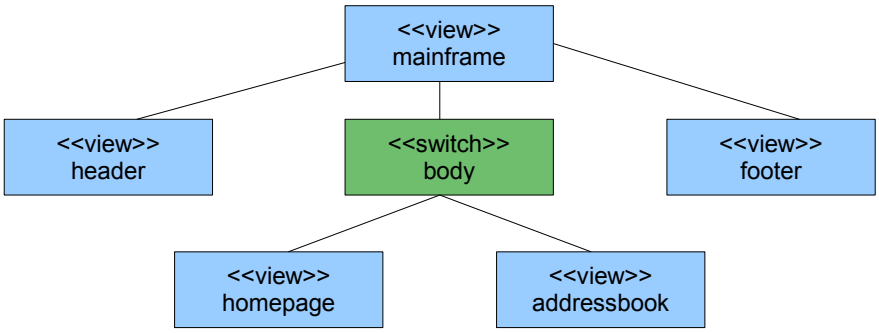


Figure 5.1: DaVinci GUI tree with view and switch nodes

A switch node can also contain other switch nodes. One of the main points of the DaVinci article, is that deeper switch nodes are not changed upon a change at a higher level in the tree. This has the advantage that, to the user, navigational paths are kept very short. Changing back the higher switch node will give back exactly the previous view. In the Echo2 solution, the concept of switch node nesting is not kept, but going back to a previous view will render the same Java objects, thus ensuring that the user will have the exact same page view.

5.2.1 Modelling pages or changes

In retargeting the DSL, an interesting question is whether the current modelling of pages is adequate. In RIAs, one thinks in terms of changes to the screen, while WebDSL enforces modelling complete pages. I would contend that modelling pages is the better solution, for reasons of ease and scalability. Note that this is my opinion, based on experience but not on literature.

Concerning ease, a developer (or a user) may very well think in terms of changes to the screen, but only for parts of an application. Application-wide, I believe one thinks about what should be shown on the complete screen. Only within a view, one would think of the

changes to that particular view. This means that a developer can more easily describe what should be put where on the screen, instead of thinking about detailed changes and needing a way to describe the changes in WebDSL.

Scalability is the other factor to take into account. The amount of pages modelled grows linearly with the application's functionality. Deciding what to change, however, is context-dependent, so the amount of modelled changes between different views grows quadratically. Figure 5.2 shows my mental model of how the amount of modelling grows with the number of functional views.

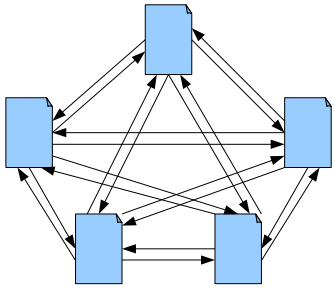


Figure 5.2: Number of differences grows quadratically

An interesting option would be a hybrid approach, modelling pages at high level and changes to pages at detailed level. Another question is whether *rich* functionality is still possible when modelling complete pages. I would say that modelling pages does limit the expressivity in terms of richness, but modelling and generating rich functionality at detailed level is very hard anyway. I will come back to both points in chapter 7.

5.3 Database update listeners

In a RESTful application, every part of a page 'notices' a database change immediately, because a complete page request is rendered after the change (or it is noticed upon the next page request, if the change is performed by another user). However, as a RIA uses delta updating, a database change needs to be communicated explicitly. For example, when you change a user's name in the middle of the screen, the text `"welcome, <username>"` at the top should be changed too.

Getting the update can be accomplished either by data pull or by data push. Data pull means that a page, or a component in it, periodically checks whether updates have been made. Data push, in turn, means that changes are pushed to the page. [4] compares push and pull techniques for Ajax applications. The conclusion to their experiment is that in general, a push approach gives better data coherence (every change is pushed) and better network performance (only communicate when a change is performed), at the cost of scalability issues. The pull approach, conversely, uses much less of the server's CPU, but setting the pull frequency is difficult: setting it too high is bad for network performance, while setting

it too low will cause data misses. At the end of the article, the authors suggest researching a hybrid approach.

The solution used here is actually a more or less hybrid approach. When a ContentPane – part of a page – does a database lookup, it is registered as interested in that kind of Entity. When an Entity of that kind changes, the ContentPane is notified that it should update itself. So a notification that a change has occurred is pushed, while the data itself is pulled. Why not push the change directly instead of the notification? Because it is more complex, needing the server to know exactly what kind of data the ContentPane wants and in what form. The chosen solution leaves these decisions to the ContentPane itself.

For efficiency reasons, a ContentPane is removed from the list of 'listeners' when it goes out of view. As soon as a ContentPane comes back into view, it will actively check for updates by redoing its database lookups.

5.4 State management

As said before, state information about the user's current session is a central aspect of interactive web applications. Probably all RIA frameworks offer support for state management. Echo2 creates an instance of the class ApplicationInstance for each user session and this instance is able to store *context properties*, which are retained over multiple requests. In fact, as Echo2 abstracts away almost the complete client/server boundary, objects and their attributes are a fine way of storing state. The reason for using context properties is that it makes state access possible from each component. For example, after storing the one LayoutManager that is used throughout the user session (described in the next chapter), it can be accessed by calling `MyApplicationInstance.getLayoutManager()`:

```
public static LayoutManager getLayoutManager()
{
    return (LayoutManager)
        getActive().getContextProperty("layoutManager");
}
```

5.5 Architecture

The architecture of the Echo2 application that is created, will be described here mostly in terms of what happens when a user accesses the application. In other words, it starts by explaining the bootstrapping mechanism of the first page and then describes what happens next, when a user clicks around.

Echo2 prescribes that a subclass is created of both its WebContainerServlet and its ApplicationInstance classes. Upon user access of the application, the former creates a new instance of the latter, calls its `init` method and places the returned Window on the web browser screen. The `init` method makes a new LayoutManager and calls its `load` method. See figure 5.3. Echo2's classes are shown in a different colour.

Next, the LayoutManager is injected with a subclass of Layout, in this case Serg2Layout. This class is generated from the DSL and puts a number of Switches on the screen. See fig-

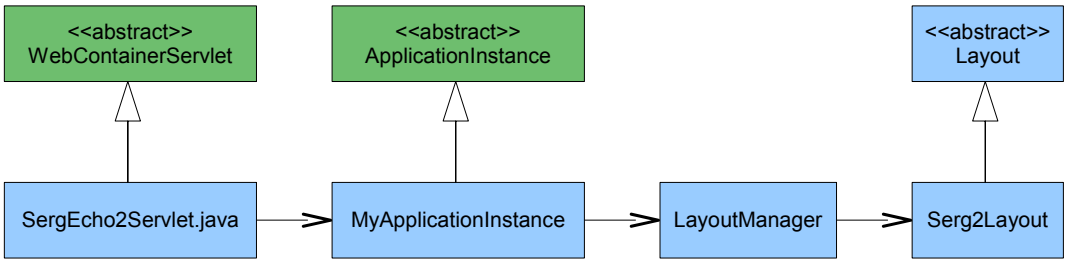


Figure 5.3: Bootstrapping, part 1

ure 5.4. The role of a Switch is that it contains one ContentPane at a time and that it can switch between different ContentPanes. A Switch is not statically assigned its alternative ContentPanes, as in DaVinci. The implementation of the delta updating for switching from one page to another is explained in the next chapter.

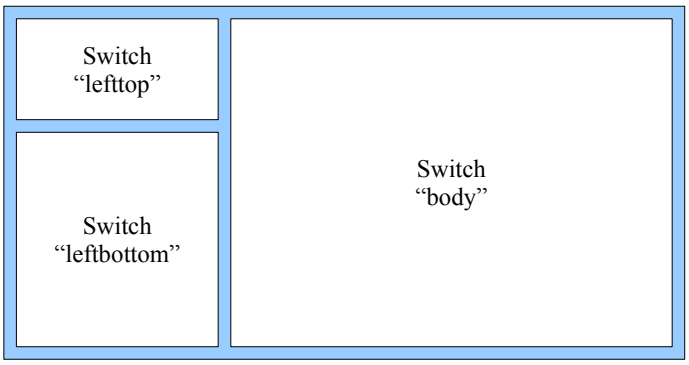


Figure 5.4: Bootstrapping, part 2

Finally, each ContentPane can access a number of Java utilities, the most important of which are:

- MyApplicationInstance, that holds state information.
- LayoutManager, that can be gotten from the MyApplicationInstance and knows which ContentPanes are in which Switches at the moment.
- MyEntityManager, that offers database access.
- Styles, that contains style information for rendering components.

Chapter 6

WebDSL to Echo2 mapping - Implementation

This chapter describes the implementation of the mapping from WebDSL to Echo2. The first three sections deal with the general RIA issues discussed in the previous chapter (see sections 5.2, 5.3 and 5.4). The other sections each deal with part of the implementation, often describing both Stratego transformation code and Java target code. Chiefly, this chapter describes the problems encountered and their solutions, stressing the difference with the Seam/JSF solution where appropriate.

6.1 Page deltas

Echo2 offers only a limited number of components that are capable of storing other components, most notably ContentPane and SplitPane. The latter is able to store two ContentPanes, either horizontally or vertically. A subclass of ContentPane is made, called Switch, that is able to store a ContentPane and can switch to another ContentPane. Because of the necessity to store multiple Switches on the screen simultaneously, a subclass of SplitPane is made. This class, MultiPane, also has a horizontal or vertical orientation, but is able to store any amount of ContentPanes by recursively adding MultiPanes to itself.

The templating mechanism described in section 3.2, uses a CSS style sheet to prescribe *where* these parts should be placed on the screen. Echo2 is not able to use CSS however. Some attempts have been made at creating a CSS parser for Echo2. For instance, EchoPointNG has a class called CssStyleSheetLoader. Its JavaDoc says that this class "will load a CSS 'like' style sheet", but the apostrophes around the word 'like' are an understatement. The style sheets it parses are nothing like CSS. No reasonable implementation of a CSS parser for Echo2 could be found, so a small Pane Location DSL has to be created to fix this problem.

A design decision is made that the layout of the screen can contain rows and columns of Switches, that a row cannot be put directly into another row (this is not usually useful anyway) and that a column analogously cannot be put directly into another column. This decision means that a 'circular' screen layout as in figure 6.1 cannot be created in the ap-

plication. However, it is possible to define a fairly complex layout as in figure 6.2 and it is done as follows:

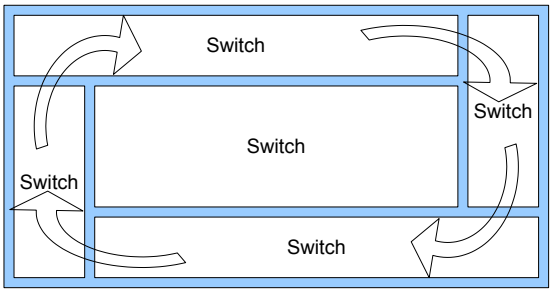


Figure 6.1: 'Circular' layout that the Pane Location DSL cannot create

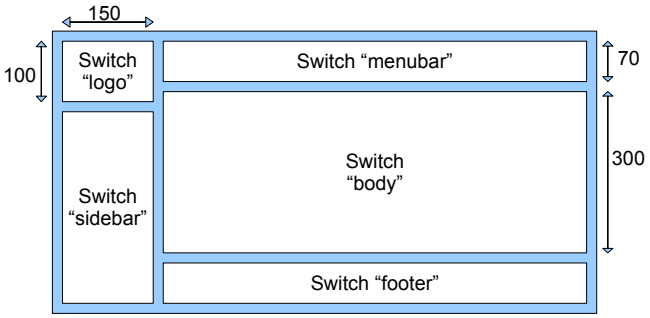


Figure 6.2: Example layout created by the Pane Location DSL

```
panelocation(main)  homepage(home) org.webdsl.serg2
{
    row(150, *)
    {
        column(100, *)
        {
            switch(logo)
            switch(sidebar)
        }
        column(70, 300, *)
        {
            switch(menubar)
            switch(body)
            switch(footer)
```

```
        }
    }
}
```

The first line says that this is the layout for the template called "main", that the first page to load is called "home" and that this Layout should be generated in the package `org.webdsl.serg2`. Next, it says that the layout consists of a row containing two columns, each with a number of Switches. Finally, the numbers define the widths of a row's elements, or the heights of a column's elements[1].

When parsing this DSL code (whose simplified syntax was shown in section 2.3), a subclass of Layout is generated, which is the class injected into the LayoutManager to get the initial screen layout, as explained in the previous chapter. This generated class basically maps the rows and columns one-to-one onto MultiPanes and adds Switches to the MultiPanes. The result:

```
public class Serg2Layout extends Layout
{
    public Layout load(...)
    {
        MultiPane mainrow = new MultiPane(
                  MultiPane.ORIENTATION_HORIZONTAL, 150);
        add(mainrow);

        MultiPane column0 = new MultiPane(
                  MultiPane.ORIENTATION_VERTICAL, 100);
        mainrow.addMultiPane(column0);

        Switch logo = new Switch();
        column0.addContentPane(logo);

        Switch sidebar = new Switch();
        column0.addContentPane(sidebar);

        MultiPane column1 = new MultiPane(
                  MultiPane.ORIENTATION_VERTICAL, 70, 300);
        mainrow.addMultiPane(column1);

        Switch menubar = new Switch();
        // etc
        return this;
    }
}
```

[1]The asterisk is added for an HTML-frameset feeling, though it carries no information in itself. Echo2 is not, as HTML, capable of rendering widths like (100, *, 50). Only the distance from the left or top can be used.

Next comes navigation by assigning a ContentPane to a Switch. In DaVinci [18], a view in a switch node is changed by calling the GUI tree's method

```
switchNode("mainframe.body", findItem(nextView)).
```

At first, a similar approach was used here. The DSL was extended with an extra `navigate` term. This term used to have two arguments: the text or icon for output and the page it would navigate to upon a mouse click. An extra argument was added with the name of a Switch, so that the method call above would be written in WebDSL as follows. This effectively puts the nextView page in the Switch called "body":

```
navigate("go to next view", nextView(), "body")
```

Up to this point, the original DSL structure with its reliance on page templates is not used yet. A page defines its complete contents and that is what is generated. However, one of the design criteria for this project was to try to keep the DSL intact and be able to generate both Seam/JSF and Echo2 front-ends from the same DSL code. So the previous extension of `navigate` has to be undone and page templates have to be incorporated. At the same time, delta updating still has to be possible.

The Seam/JSF mapping expands templates completely, effectively creating complete pages. For delta updating, however, it is useful *not* to expand the template. This makes it possible to notice the difference between consecutive pages more easily.

Let us use a running example. The following is the definition of a template `main`:

```
define main()
{
    div("sidebar")
    {
        logo()
        sidebar()
    }
    div("mainpane")
    {
        menubar()
        body()
        footer()
    }
}
```

Now let us define two pages `home` and `information` as in figures 6.3 and 6.4.

From the definition of a page, a PageCreator and a number of ContentPanes are generated, namely one for each Switch (template hook) that the page redefines. See figure 6.5. The PageCreator Home only knows that it uses Template Main and that it redefines Switch "sidebar" with HomeSidebar and Switch "body" with HomeBody.

```
define page home()
{
    main()

    define sidebar()
    { "this is home's sidebar" }

    define body()
    { "this is home's body" }
}
```

Figure 6.3: Page home

```
define page information()
{
    main()

    define menubar()
    { "a menu" }

    define body()
    { "very useful info" }
}
```

Figure 6.4: Page information

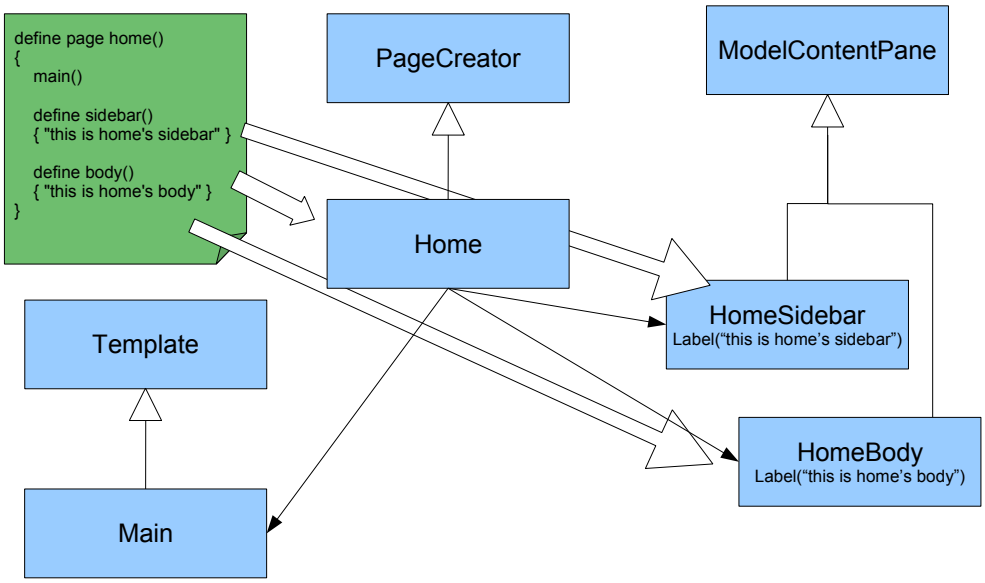


Figure 6.5: From page definition to PageCreator and ModelContentPanes

Now, when going from `home` to `information`, PageCreator Information will ask Home which Switches it redefines. Armed with this knowledge, Information tells Main to reset Switch ”sidebar” to its default value, to put InformationMenubar in ”menubar” and to put InformationBody in ”body”. Switches ”logo” and ”footer” are unaffected by this page change, so only the Switches that *needed* to be changed were in fact changed: delta updating, at last.

Finally, a word on bootstrapping to round up this section. The LayoutManager calls the generated Layout class, which puts the Switches in MultiPanes, as shown before. It then puts the template's Switches on the screen, which may be updated according to the first PageCreator called (Home, in the example above). The updated result is put on the screen. From this point on, switching between pages is done with delta updating.

6.2 Database update listeners

A simple listener system is implemented for database updates. When a ContentPane calls, e.g. `getAll(Person.class)` – which in turn calls `entityManager.createQuery("from Person").getResultList()` – the pane is added to a list of panes that are ‘interested’ in updates on Persons. This list is actually a `Map<String, List<ContentPane>>`, where the String is the classname (e.g. “Person”).

When a Person is subsequently persisted or removed, the Map is checked for all panes that need to be notified. The following method is then called on these panes:

```
public void notifyOfUpdate()
{
    if(wantNotification)
    {
        // Lookup Entities in database
        //   that are relevant to me
        // Re-initialize myself.
    }
}
```

The `wantNotification` checks to see if the pane is in a state in which it wants to be updated. It is sometimes useful to set this to false, for example when the user is in the process of editing a form, as the pane should not be re-initialized at that moment.

For efficiency reasons, a pane is removed from the listener list when it is removed from the screen, because in that case it is not useful to update it.

Finally, the implemented listener system only works for each user separately. The listener list is only kept inside the user's session. Future work will certainly need to update this, for changes should be communicated to all users, at least after a certain amount of time.

6.3 State management

A small number of objects is stored in context properties by MyApplicationInstance and can be retrieved by asking MyApplicationInstance for them. These are currently:

- The current PageCreator. Used when switching pages, to lookup which parts of the screen to update.
- The LayoutManager, that can be told to switch pages.
- The Map of ContentPane listeners, as described in section 6.2.
- The current Template used, which is also used for delta updating.
- A properties file with user settings for the application. This XML file is parsed by the java.util.Properties class and then stored. It contains such information as the location of the images used in the application and the name of the implementation of Layout that is generated, for injection into the LayoutManager.
- The current User, which is not used yet, but should be stored, for instance for authorization purposes.

The MyApplicationInstance itself can be retrieved by any component by calling the static method `MyApplicationInstance.getActive()`.

6.4 Page element generation

This section explains the Stratego implementation of how the page elements of WebDSL are transformed to Java code. The list of page elements discussed here is not exhaustive: a selection of interesting elements has been made. What is worth noticing, is that a number of elements can be mapped pretty much one-to-one from DSL to Java code (e.g. the For loop), while some undergo heavy transformation (e.g. the `listitem`).

Extra attention in this chapter is given to those elements that have a rather different transformation than their counterparts in the Seam/JSF mapping. In that mapping, a lot of elements are transformed both to Seam code (e.g. for initializing values and performing actions) and to JSF code (for representation on the screen), while the Echo2 mapping collects these in one place, which is often clearer. On the other hand, JSF follows a very different element order scheme than Echo2, and the combination with Seam's bijection (see section 3.1) has a profound impact on the transformations. Some constructs are very different in Echo2 and take extra effort to accomplish. See e.g. sections 6.5 and 6.6. Let us say that Seam and JSF solve a number of common web application programming issues very well.

6.4.1 For and ForAll

```
add-components :
    For(x, SimpleSort(x_Class), var, elems) ->
```

```
    bstm*
    |[
        for(x_Class x : ~list)
        {
            bstm1*
        }
    ]|  where list := <arg-to-java-exp> var
      ; bstm1* := <filter-concat-warn(add-components|"")> elems
```

In the very similar ForAll loop, the first line of Java is replaced by:

```
for(final x_Class x : em.getAll(x_Class.class, this))
```

Here, `final` is added to give possible ActionListeners inside the loop access to the Entity, `em` is an instance of the class MyEntityManager and `this` is added to the method for registration of this class as a listener for updates to this Entity class.

This implementation is much shorter than the one in the Seam/JSF mapping. There, in the Seam code, a class attribute has to be created and initialized for the list of Entities, which can then be iterated over by the JSF code.

6.4.2 Action

Two constructs are called `actions` in WebDSL. First, a button that calls a method:

```
action("save person", save())
```

It is transformed into an Echo2 Button with an ActionListener that will call the method on a mouse click (an `actionLink` is transformed the same way, only with an HTML-like link instead of a Button):

```
add-components :
    TemplateCall("action", [e1, e2], []) ->
    bstm*
    |[
        Button x_button = new Button("~x_text");
        x_parent.add(x_button);
        x_button.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent actionEvent)
            {
                x_action;
            }
        });
    ]|  where x_text := <arg-to-value-string-echo2> e1
          ; x_parent := <Parent>
          ; x_action := <arg-to-value-string-echo2> e2
```

The second `action` element is a method:

```
action save() { person.save(); return viewPerson(person); }
```

which is transformed into a Java method:

```
action-void-to-java-echo2 :
    Action(x_action, args, Block(stm*)) ->
    |[
        public void x_action(param*)
        {
            firePropertyChange("~x_action", null, null);
            bstm*
        }
    ]|  where param* := <map(action-arg-to-method-arg)> args
            ; bstm*  := <statements-to-java> stm*
```

The call to `firePropertyChange()` is explained in section 6.6. In short, it tells other elements in the page that this Action has been called, so that they may perform some operation if that is relevant to them.

The setup above, with buttons calling Java methods is very similar to the one in the Seam/JSF mapping. The main difference, is that the two parts are in separate files there: the button is in the JSF page, while the action method is in the Session bean.

6.4.3 Return

The transformation rule above, for an `action` method, calls `statements-to-java`. This strategy handles the few imperative statements that are allowed in WebDSL.

One of these is the `return` statement, like the `return viewPerson(person)` we saw before. This is very close to the Seam/JSF solution, but not difficult for Echo2 to handle.

A JSF submit button can call a method in a Seam session bean. If this method is not `void`, it returns a page String, like `return "/viewPerson.seam?person=523";` and JSF will go to that page. In Echo2, a method returning a String makes no such thing happen. So the action does not need to return anything, but it needs to call LayoutManager and tell it to switch pages. In the first rule shown here, the page has arguments; in the second rule, it doesn't.

```
return-to-java-echo2 :
    Return(ThisCall(page,[e1*])) ->
    |[
      MyApplicationInstance.getLayoutManager().switchTo("~page", e2*);
    ]| where e2* := <map(arg-to-java-exp)> e1*

return-to-java-echo2 :
    Return(ThisCall(page,[])) ->
```

```
    |[
      MyApplicationInstance.getLayoutManager().switchTo("~page");
    ]|
```

6.4.4 List and listitem

A Switch in the Echo2 implementation contains a ModelContentPane, a new subclass of ContentPane to which multiple components can be added. It effectively creates a column of components on the screen. In that sense, `lists` and `listitems` do not need to be transformed, except for styling purposes. The Seam/JSF mapping creates simple unordered HTML lists, for example. *As an experiment*, such a bulleted list was also created in the Echo2 mapping, needing a lot of code:

```
add-components :
    TemplateCall("listitem", [], elems) ->
    bstm*
    |[
        Button button = new Button(
               Styles.getImage("/img/bullet.png"));
        Button button2 = new Button();
        button2.setWidth(new Extent(5));
        Column x_column = new Column();
        bstm1*

        Row row = new Row();
        row.add(button);
        row.add(button2);
        row.add(x_column);
        x_parent.add(row);
    ]|
    where x_parent := <Parent>
    ; x_column  := <newname> "column"
    ; {| Parent : rules ( Parent := x_column)
       ; bstm1* := <filter-concat-warn(add-components|"")> elems
      |}
```

Three items are placed horizontally to make a `listitem`: an image of a bullet, an empty button (for space between the bullet and the third item) and a column which contains the actual `listitem`, as this might be more than just one component. See section 6.4.6 for more information on the `<Parent>` rule used here.

This code renders a list that resembles an HTML unordered list, but it leaves little room for styling. A more context-sensitive implementation might, for example, render the `listitem` differently when it is placed inside a menu, as happens in the CRUD generating code. Actually, the Seam/JSF mapping indeed renders the `listitem` without a bullet when

inside a menu, but it does so with CSS styling. Since CSS cannot be used in Echo2, a new `menu` element is suggested in section 7.2.1.

6.4.5 Navigate

A `navigate` element (similar to an HTML link) has an output text and a page (or URL) to go to when clicked. The page can also have arguments. Because a link needs an ActionListener to switch pages, and because links are used very often in a web application, an Echo2 component Link was made. It is a subclass of Button, looks like an HTML link (blue and underlined) and it knows to call LayoutManager to switch pages.

```
public Link(String text, String page, Object ... args)
{
    //save arguments

    Styles.setLinkStyle(this);
    addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            MyApplicationInstance.getLayoutManager()
                    .switchTo(page, args);
        }
    });
}
```

As an example, the navigate

```
navigate(person.name) { viewPerson(person)) }
```

is transformed into the following Java code:

```
Link link = new Link(person.getName(), "viewPerson", person);
parent.add(link);
```

To create a link with an output image instead of text, an extra rule is needed in the Echo2 mapping. In JSF, as in HTML, a link with text or an image is created with a construct similar to `<create_a_link>output_text_or_image<end_link>`. In Echo2, a Button can have an image or text, but it needs to be told explicitly which one it gets. So, a link with an output image creates the following Java code:

```
Link link = new Link(Styles.getImage(img), "viewPerson", person);
parent.add(link);
```

leading the Link to set its icon to `img`.

As a side note, by default Echo2 crashes when an image is not found, because it gets a NullPointerException. In HTML, some picture with a question mark (or similar) is placed on the screen at that position. For the Echo2 code, care has to be taken to check whether the image exists and otherwise explicitly put a default image or text on the screen.

6.4.6 Table and row

In WebDSL, rows are stored in tables, as in HTML. Though rows could just as well be placed outside tables, adding them to a table helps aligning their cells horizontally. Otherwise, the following ugly layout could appear:

```
first name  Albert
surname  Cliphard
homepage  www.albc.nl
```

A second point to make needs some background information first. As seen in most examples above, components are added to a *parent*. At the top level, components are added directly to the ModelContentPane, creating a column of components on the screen. So by default, components are added by `this.add(component)`. Components inside a `row`, however, must use the `row` as their parent. A dynamic rule is used to accomplish this, setting the `row` as the current Parent:

```
add-components :
    TemplateCall("row", [], elems) ->
    bstm*
    |[
      Row x_row = new Row();
      x_parent.add(x_row);
      bstm1*
    ]|
    where x_row := <newname> "row"
    ; x_parent  := <Parent>
    ; {| Parent : rules ( Parent := x_row )
       ; bstm1* := <filter-concat-warn(add-components|"")> elems
      |}
```

The same holds for `listitem`, that, as seen in section 6.4.4, has a Column as its third element, to which components can be added. There too, Parent is set to the Column in a dynamic rule, so that components inside see the Column as their parent.

6.5 DSL scopes

WebDSL has a `header` and a `section` element. The first creates a header whose size depends on how deep inside nested sections it is positioned. The second is used to count the section depth and it can also be stylized with CSS. The following code could be entered in the DSL:

```
section
{
    header  { "first header" }
    section { header { "second header" } }
    header  { "third header" }
}
```

This creates the following HTML code:

```
<h1>first header</h1>
<h2>second header</h2>
<h1>third header</h1>
```

The Seam/JSF mapping handles the DSL code easily:

```
elem-to-xhtml :
    TemplateCall("header", [], elems) ->
    %>
        <~n:tag><%= <elems-to-xhtml> elems ::*%></~n:tag>
    <%
    where n := <SectionDepth>
        ; tag := <concat-strings>["h", <int-to-string> n]
```

And a similar transformation could be used for the Echo2 mapping:

```
add-components :
    TemplateCall("header", [], [text]) ->
    bstm*
    |[
        Label x_label = new Label("~text");
        Styles.setHeaderSize(x_label, x_size);
        x_parent.add(x_label);
    ||  where x_size := <int-to-string> <SectionDepth>
```

The difficulty is that the DSL tags are defined to work *from here* and *until here*, as are JSF tags, whereas Java tags say *do this here*. In this example, the target code needs to realize that the "third header" is also an `<h1>`. This is solved by using a dynamic rule for the SectionDepth variable, that is incremented when a new `section` is started and decremented at its end.

But the real problem with these elements, is that a `header` can contain any number of elements, instead of just one text element (note how I cheated by using a single `text` element in the left-hand side term above, instead of a list of elements `elems`). In my opinion, WebDSL follows JSF's style a bit too closely here. The following bit of legal DSL code is *not* usual in web pages. It can easily be converted to JSF, but it creates difficulties for the Echo2 mapping.

```
section
{
    header
    {
        "first header"
        section { header { "second header" } }
        "third header"
    }
}
```

Suddenly it becomes necessary for all textual elements inside the first `header` to be explicitly told that they are inside a header, and to be set to the appropriate size. The solution is to create a Java List in which to store all textual elements inside a `header` and set their sizes afterwards:

```
add-components :
    TemplateCall("header", [], elems) ->
    bstm*
    |[
        List<Component> x_header = new ArrayList<Component>();
        bstm1*
        for(Component x_comp : x_header)
            Styles.setHeaderSize(x_comp, x_size);
    ]|
    where x_size := <int-to-string> <SectionDepth>
  ; x_header     := <newname> "header"
  ; {| HeaderCollection : rules (HeaderCollection := x_header)
     ; bstm1*    := <filter-concat-warn(add-components|"")> elems
    |}
```

Next, every textual element inside this `header` has to be added to the HeaderCollection, e.g.:

```
add-components :
    TemplateCall("text", [arg], []) ->
    bstm*
    |[
        Label x_label = new Label("~x_text");
        x_parent.add(x_label);
        bstm1*
    ]|  where x_text := <arg-to-value-string-echo2> arg
          ; x_parent := <Parent>
          ; bstm1*   := <add-in-header(|x_label)>
```

```
add-in-header(|textComponent) :
    <id> ->
    bstm*
    |[
        x_header.add(x_comp);
    ||  where x_header := <HeaderCollection>
          ; x_comp     := textComponent
          ; not(<eq>(<HeaderCollection>, "null"))
```

This adds the line `<some_header_collection>.add(<the_text_component>)`, if the text component is inside a `header` tag. A solution that is a little cleaner is to add an annotation to all elements inside the `header`, rendering the `header.add(component)` when the annotation is found.

It is true that this problem could have been solved easily, by letting a `header` have only one element, which is normal for a web page. Though keeping WebDSL intact is one of the design criteria, this would be an improvement that would not hurt the Seam/JSF mapping. But the main point made in this section, is that there is a difference in scoping style between WebDSL and JSF on the one hand and Echo2 on the other. It is an interesting task to solve this discrepancy where it occurs, either by a Stratego/XT solution, or by a target code solution.

6.6 Link between page elements and Entities

A JSF page can work directly on Entities. An input field with `value="#{person.name}"` is filled with the person's name. When the form is submitted, the (possibly changed) value of the text field is read and saved back to the person's name property.

In Echo2, on the other hand, one needs to explicitly read the new value and store it to the Entity's property. There is no link between the page element and the Entity. WebDSL, in turn, does not offer the functionality to read from an input field. And the `action` method does not know which input fields it needs to read from.

To solve this, each input field adds a PropertyChangeListener that, when called, will read the input field and save its value to the Entity. This way, each input field is itself responsible for getting and setting the property.

So, for input field `input(person.fullname)`, the following two rules apply:

```
add-components :
    call@TemplateCall("input", [e], []) ->
    bstm*
    |[
        TextField x_field = new TextField();
        x_field.setText(x_text);
        x_parent.add(x_field);
        bstm1*
```

```
    ||  where x_field  := <newname> "field"
            ; x_text   := <arg-to-value-string-echo2> e
            ; x_parent := <Parent>
            ; bstm1*   := <add-input-listener(|x_field)> call

add-input-listener(|x_field) :
    TemplateCall(_, [e], []) ->
    bstm*
    ||
        formElements.put("~x_formElement", x_field);
        addPropertyChangeListener(new PropertyChangeListener(){
            public void propertyChange(PropertyChangeEvent arg0)
            {
               String text = ((TextField)
                  formElements.get("~x_formElement")).getText();
               x_set;
            }
        });
    ||  where <eq>(<Form>, True())
            ; <reversable-field-access> e
            ; x_formElement := <newname> "formElement"
            ; x_set         := <reverse-field-access(|"text")> e
```

The second rule rule only applies when used inside a `form`, as the semantics of a `form` are that it can change data. Furthermore, argument `e` is the Entity's property (e.g. `person.name`), that in `x_set` is changed to `person.setName(text)`.

Each action then fires a PropertyChange, as seen in the explanation of the Action page element in section 6.4.2, so that all listeners know that they can perform their operation.

Two small issues still exist in this implementation. First, the PropertyChange fires twice, as a consequence of some implementation detail in Java. It is only a small performance penalty: the propertyChange method will act twice, setting the property twice. Second, the PropertyChange fires on *every* action, also on, e.g., `cancel()`[2]. This second problem also exists in the Seam/JSF mapping, where it is solved temporarily by checking whether the action is not `cancel()`. But as the Entity is not persisted in `cancel()` anyway, it is not a problem to set its properties.

[2]In its current use, `cancel()` only contains a statement for going to another page.

Chapter 7

Discussion

7.1 Regarding research questions

1. **Is it reasonably feasible to generate Echo2 code from WebDSL?**
Mapping page elements in WebDSL to screen components in Echo2 is relatively straightforward. Some elements can be mapped one-to-one, while some elements take a bit of a work-around.

More difficult is the placement of components on the screen, because CSS cannot be used in Echo2. In combination with placing multiple ContentPanes on the screen for easy delta updating, a small Pane Location DSL is needed. This tells Echo2 how to position the initial Switches.

2. **Is it, at the same time, reasonably feasible to generate good RIA code from WebDSL?**
A Rich Internet Application is defined as combining traditional web applications with the user interface richness of desktop applications. Rich functionality can be found at application-wide level and at page element level. This work concentrates on generating smooth page transitions with delta updating, on notifying page elements of changes and on maintaining state.

Delta updating is achieved by splitting up a page in a number of Switches, checking which Switches are redefined on a page transition, and replacing those Switch in their entirety. This is a bit coarser than needs be, but much simpler to model and execute.

Concerning change notification, 'listener' lists of ContentPanes are used, to know who is interested in which Entity. A change to an Entity will cause a notification message to be pushed to the interested ContentPanes, that can then decide if and when they want to pull the changed data.

State management is easy with Echo2 and probably with any RIA framework. Here, state is stored in an instance of a subclass of Echo2's ApplicationInstance, an object unique for each user session and that can be called from any component.

Page element richness is harder to model and generate and this work does not focus on it. It should be stressed that it is not so much WebDSL that constrains rich functionality modelling, as it is that modelling richness is hard anyway. An interesting option is an ap-

proach that separates high level and low level page modelling. This could also solve the issue of whether to model complete pages or only changes to the current screen. At high level, pages could be modelled, keeping the overview of the application simple and small to the developer and to the user. At detailed level, changes to the page could be modelled, making it easier to create a rich interface.

3. **Does WebDSL cover web application modelling?**
A dichotomy between RESTful applications and RIAs is used in this thesis. Both types of web applications can be modelled in WebDSL. As they are fairly different and as, in particular, Seam/JSF differs from Echo2 a lot too, I would contend that WebDSL offers a good model for web applications.

As in much of the described similar work, WebDSL actually combines a number of models: Entity definitions; page definitions; page templates to enhance code reuse; HQL, a DSL for database querying, so as not to have to reinvent the wheel and because developers know its sibling SQL already; and a very small action language for imperative statements. As these models resemble the ones used in similar work (see sections 1.3 and 3.5), it seems that WebDSL is fairly complete.

In general, I find WebDSL easy, flexible and expressive. It is a joy to write so little code that so precisely defines what should be shown on the screen. This is very useful for data input applications, as well as content management. In its current form, it is less useful for dynamic applications. “Business logic”, for example for access control, workflow management or business constraints, is as yet not or hardly expressible in WebDSL. An extension of the action language to be able to access externally defined libraries, would make WebDSL better able to model full-scale applications.

4. **Can we learn anything new for DSL engineering from this retargeting?**
Retargeting WebDSL is a way of judging its capability of modelling web applications in general. A few WebDSL constructs were found to be too close to the JSF coding style, but most problems encountered were because of limitations of Echo2. Still, the case that WebDSL can model web applications is much stronger now that a second framework and a second interaction pattern have been found to be targetable.

It seems like a good idea to take an approach of targeting two very different frameworks from the start. This will take much more development effort (though not nearly twice as much), but chances are it will lead to a better DSL: a DSL that is generic to its domain.

7.2 Future work on the Echo2 mapping

A small number of page elements still needs a mapping to an Echo2 component. Additionally, a couple of new page elements and some new sugar are proposed in the following sections. These could fairly easily be added to WebDSL. Rich concepts like drag-and-drop, data push and movies, require some more design first, before they can be incorporated.

The hybrid approach of modelling pages and changes to pages could also be designed and implemented. Especially the second part could be difficult, but I think it could offer a

big advance to RIA modelling and development. Furthermore, a little work is left in getting the page templating mechanism to work perfectly: some templates need expanding, while others do not.

But then most work will be left in getting the Echo2 application production-ready. The following is a list of the items that need attention.

- Caching and update notifications. At the moment, a ContentPane is only notified of updates that happen within the user's session. So if another user changes anything, the pane is not notified.
- Access control.
- Exception handling is still very crude. Some exceptions generate neat pop-up windows with an explanation to the user, while others crash the application. Also, some form of modelling exception handling might be useful.
- Workflow modelling.
- Stress testing. I have a feeling that too many objects are being stored unnecessarily. Also, multiple users tests have not been done yet.
- Concurrency control.
- Layout and styling.
- Assigning processes to the client. This is probably not feasible in Echo2 though.

7.2.1 Proposed new page element: menu

In the Seam/JSF mapping, a `listitem` is rendered as an HTML unordered list: an item preceded by a bullet. To render a dropdown menu, the same `listitem` is used, changed by a CSS style sheet. Since Echo2 cannot use CSS, and as it generally seems like a good idea, I propose a `menu` element, that holds `menuitems`.

7.2.2 Proposed new page element: listbox

An Entity can have a list of associations of another Entity, for example a Publication having a list of Persons as its authors. When editing a Publication, the list of current authors and the list of all Persons are both used. The latter could also be a list of all Persons minus the current authors. In any case, the current WebDSL implementation (see the result in figure 7.1a) is to print the `list` of current authors, with a remove button behind each, that will remove that author both from the list on the screen and from the Entity's list. The `list` is followed by a dropdown `select` field that contains all Persons and calls `<entity_association_list>.add(person)`. In WebDSL, the following code is used:

```
list
{
    for(person : Person in publication.authors)
    {
        listitem
        {
            text(person.name) " "
            actionLink("[X]", removePerson(person))
            action removePerson(person : Person)
                { publication.authors.remove(person); }
        }
    }
}
select(person2 : Person, addPerson(person2))
action addPerson(person : Person)
    { publication.authors.add(person); }
```

Selecting an Entity in the dropdown list instantly adds it to the Entity's list. Even on pressing `cancel` the action cannot be undone. A better way of modelling the input of association lists is a list box, a *multiple select field* in HTML[1]; see figure 7.1b. It is a small addition to WebDSL that makes it simpler while not making it more target-specific. A list box is an element used in HTML, JSF and Echo2.

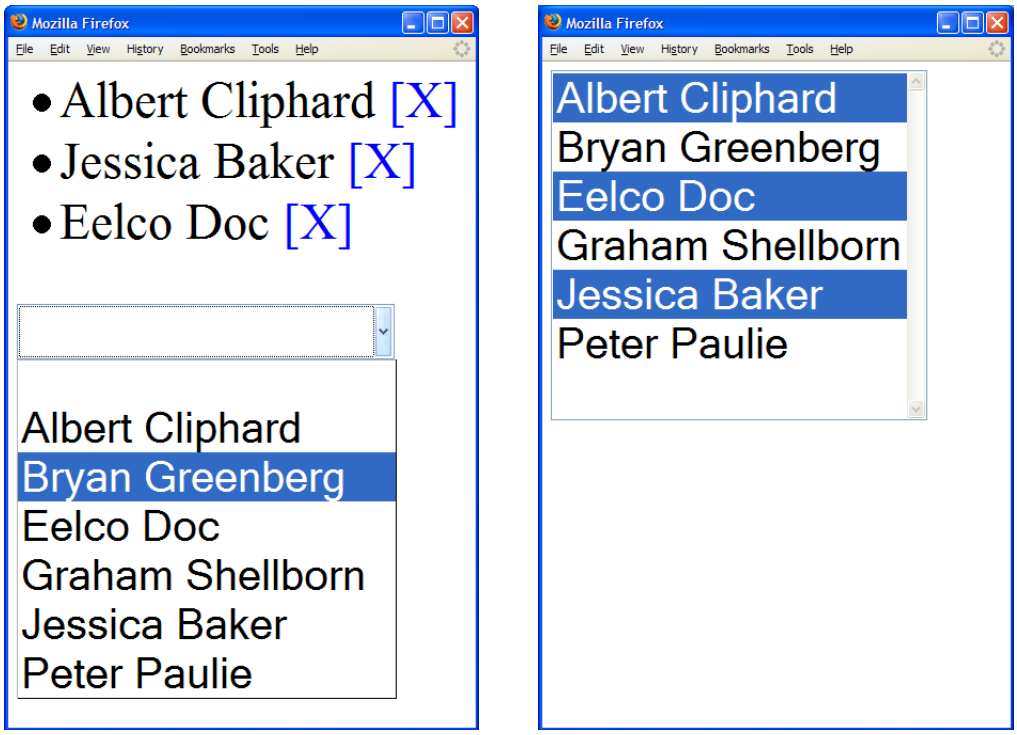
A `listbox` element could then be used in the following way:

```
listbox
{
    for(person : Person)
    {
        listboxitem(person)
    }
    listbox.select(publication.authors)
}
action("save", save())
action save() { publication.save(); }
```

7.2.3 Proposed new page element: mnemonic

A typical rich addition to a web application is the use of *mnemonics*: shortcut keys like control+s for save. JavaScript is able to listen to keys and react to them and EchoPointNG has a KeyStrokeListener as well. A mnemonic should be used to call an `action`, same as a button does. The following examples could then be typed in WebDSL:

[1] Note that for this specific example of Publication authors, a list box might not be the right solution, as the *order* of the authors matters.



(a) Listitems and select field (b) List box

Figure 7.1: Input association lists

```
mnemonic(control, n)       { newAuthor()    }
mnemonic(control, alt, R)  { doSomething()  }
mnemonic(g)                { doSomething()  }
mnemonic(f1)               { help())        }
```

Zero or more *masks* could be used, followed by a letter or function key. This leads to the following SDF definition (`mnemonic` itself does not need to be defined in SDF, because it will be parsed as a TemplateCall[2]):

```
"control"     -> Mask {cons("Control")}
"alt"         -> Mask {cons("Alt")}
"shift"       -> Mask {cons("Shift")}
[a-zA-Z0-9]   -> Key  {cons("Key")}
[fF][1-9]     -> Key  {cons("FKey")}
[fF][1][0]    -> Key  {cons("F10")}
[fF][1][1]    -> Key  {cons("F11")}
[fF][1][2]    -> Key  {cons("F12")}
```

[2]TemplateCall matches all elements resembling `id(args) {elements}`.

7.2.4 Proposed new sugar

An `action` method is often only called from one `action` button. To write this in WebDSL costs two constructs:

```
action("save person", save()) {}
action save() { person.save(); return viewPerson(person); }
```

The following line is a sugared representation of the example above:

```
action("save person") { person.save(); return viewPerson(person); }
```

The only difference is that the action's name *save* is not known in the sugared representation. This means that another button will surely not be able to call this method.

For `mnemonic` (see previous section) an analogous sugaring could be used: instead of having the `mnemonic` call an `action` method, it could incorporate the `action`'s statements. However, a `mnemonic` usually is an *addition* to a button on the screen, so then the `action` method will need to be written separately anyway.

7.3 Working with Stratego/XT

Stratego/XT is a very powerful tool set for creating a transformation system. Also, writing a syntax in SDF is very easy and concise. And the support for transformation rules and strategies in Stratego and its libraries is immense.

Three issues remain, however. First, Stratego/XT can only be run under a limited number of operating systems, notably not under Windows. Second, compiling the WebDSL transformation system takes about 40 seconds each time. Some form of modular compilation could offer relief. It is especially gruesome for a beginner who needs to try out *many* different syntactic constructs before getting it right. This brings me to the third issue, namely that Stratego has an extremely rich syntax, while its documentation hardly describes any syntax.

For example, say the rule `contents-to-java` is called on the term `Div(divname)` with value "footer" for divname, and we want to have it write the following statement:

```
Footer footer = new Footer();
```

This needs a simple rule:

```
contents-to-java :
    Div(divname) ->
    bstm*  |[  x_Div x_div = new x_Div();  ]|
    where x_div := divname
        ; x_Div := <capitalize-string> divname
```

Now this is only simple when the SDF definition of metavariables is known, namely that *x underscore id* matches a metavariable. Not realizing this, I first only used antiquotation

with the tilde (~), *fighting* to get the syntax right. Another example is `pkgname`, that can be followed by a number and is defined as a variable for a Java PackageName. Using any other syntax, like `pkgname.SomeClass` does not work and no clue is given why.

Another type of syntax problem is in calling a strategy. Which of the following is correct in which situation? This is something a beginner cannot possibly understand from the Stratego/XT manual.

- `do-something`
- `<do-something>`
- `<do-something> x`
- `do-something(x)`
- `<do-something(x)>`

7.4 Working with Echo2

Writing a web application in Java is fantastic. The solid syntax and semantics, compile-time checking, explicit method calling, object extension and excellent support by the Eclipse IDE are a relief after working with JSF, Seam, JSP, EJB, etc. to create web applications. Echo2 offers a nice framework for creating web applications, though a lot of extensions should be added to it to make it more generally applicable.

What Echo2 needs most, as does Swing, is a good tutorial or better component constructors. Adding a Table to a ContentPane will not render anything before a TableCellRenderer is added too. But this is not written in the Table documentation, nor does Table's constructor accept a TableCellRenderer parameter. Generally, components have many dependencies that their JavaDocs do not specify.

7.5 Reflection

In writing my own thesis assignment, I had a lot of freedom in what to do and how to approach it. This freedom is nice, but it has drawbacks too. Most prominently: nobody waiting for my results. This changed for the better when Nico and I decided that I should create a DSL for web application interfaces and this happened to align my work with Eelco's. Finally, someone was very interested in the details of my work, creating a great incentive to work hard and produce something valuable.

What could have helped, was if I had been more focused on an attainable goal from the beginning. Having clear what the objective is, makes the path towards it clearer too. Everyone tells me this is part of the thesis experience however.

The second driving force was having a hard deadline, a cliché found to be very true. As soon as I had set a deadline for my thesis presentation and had communicated it to the committee and my parents (who live abroad), I picked up the pace in my project enormously, making decisions and going for it.

A slowing force on my project was the programming. Fighting with Stratego's syntax cost many days. When an idea is thought through well, fighting with such details is frustratingly beside the point. By the end though, I could implement rules and strategies reasonably fast and I got more and more enthusiastic about Stratego/XT.

Finally, communication with Nico and Eelco was very good. Nico was always open for discussion or questions, giving good feedback on, e.g., the structure of the thesis. Meetings with Eelco always took a couple of hours: our conversations ranged from brainstorming about our field of work, to discussing minute details of WebDSL, to the quality of the coffee from the machine near the ladies' room. What I liked most is that both Nico and Eelco are very to the point. They do not use big words when they are not necessary and they ask direct questions when they do not understand something. Thank you again for all your help. I had fun doing this.

Bibliography

[1] Jeremy Allaire. Macromedia Flash MX – A next-generation rich client. Technical report, Macromedia, March 2002.

[2] João Paulo A. Almeida, Marten van Sinderen, Luís Ferreira Pires, and Dick Quartel. A systematic approach to platform-independent design based on the service concept. In *EDOC '03: Proceedings of the 7th international conference on Enterprise Distributed Object Computing*, pages 112–123, Washington, DC, USA, September 2003. IEEE Computer Society.

[3] Joern Bettin. Practical use of generative techniques in software development projects: An approach that survives in harsh environments. *OOPSLA'2001 Workshop on generative programming*, 2001.

[4] Engin Bozdag, Ali Mesbah, and Arie van Deursen. A comparison of push and pull techniques for AJAX, 2007.

[5] Alessandro Bozzon, Sara Comai, Piero Fraternali, and Giovanni Toffetti Carughi. Conceptual modeling and code generation for rich internet applications. *Proceedings of the 6th international conference on Web engineering*, pages 353–360, 2006.

[6] Mark van den Brand, Hayco de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software Practice and Experience*, 30(3):259–291, 2000.

[7] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-Specific Language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[8] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[9] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[10] Jesse James Garrett. Ajax: A new approach to web applications. Online: http://www.adaptivepath.com/publications/essays/archives/000385.php (last accessed Sep 2007), February 2005.

[11] John Hunt. Echo2 versus GWT. Online: http://www.regdeveloper.co.uk/2006/08/24/echo2_framework/ (last accessed Sep 2007), August 2006.

[12] Jonathan Joubert. Echo2: Swing programmeren voor de browser (2) (in dutch). Online: http://blog.finalist.com/2007/05/23/echo2-swing-programmeren-voor-de-browser-2 (last accessed Sep 2007), May 2007.

[13] Jonathan Joubert. Model-driven development, deployment and maintenance of web applications. Unpublished literature study, Delft University of Technology, March 2007.

[14] Nico Klasens. AJAX is een keuze (in Dutch). Online: http://blog.finalist.com/2006/10/02/ajax_is_een_keuze (last accessed Sep 2007), October 2006. Published before in September 2006 in *IT Monitor* by Studievereniging Bestuurlijke Informatiekunde, Tilburg, The Netherlands.

[15] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained: The Model Driven Architecture: Practice and promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] Nora Koch and Andreas Kraus. The expressive power of UML-based web engineering. In *Proceedings of the second International Workshop on Web-Oriented Software Technology (IWWOST'02)*, volume 16, pages 105–119, 2002.

[17] Thomas Koch, Axel Uhl, and Dirk Weise. Model Driven Architecture. Online: http://www.omg.org/docs/ormsc/02-09-04.pdf (last accessed Sep 2007), November 2001. Interactive Objects Software GmbH.

[18] Andreas Langegger, Jürgen Palkoska, and Roland Wagner. DaVinci – a model-driven web engineering framework. *International Journal of Web Information Systems*, 2(2):119–132, June 2006.

[19] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[20] Ali Mesbah and Arie van Deursen. An architectural style for Ajax. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53. IEEE Computer Society, 2007.

[21] Ali Mesbah and Arie van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, March 2007.

[22] Granville Miller, Scott Ambler, Steve Cook, Stephen Mellor, Karl Frank, and Jon Kern. Model Driven Architecture: The realities, a year later. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 138–140, New York, NY, USA, 2004. ACM Press.

[23] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194, New York, NY, USA, 1997. ACM Press.

[24] Jeffrey C. Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and Daniel M. Hellerstein. RFC 3229: Delta encoding in HTTP. RFC Editor, United States, 2002.

[25] Pierre-Alain Muller, Philippe Studer, Frédéric Fondement, and Jean Bezivin. Platform independent web application modeling and development with Netsilon. *Software and Systems Modeling*, 4(4):424–442, 2005.

[26] Kevin Mullet. The essence of effective rich internet applications. *Macromedia Experience Design*, 2003.

[27] Linda Dailey Paulson. Building rich web applications with Ajax. *Computer*, 38(10):14–17, 2005.

[28] Juan Carlos Preciado, Marino Linaje, Fernando Sánchez, and Sara Comai. Necessity of methodologies to model rich internet applications. In *WSE '05: Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, pages 7–13, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, February 2006.

[30] Tilman Seifert and Gerd Beneken. Evolution and maintenance of MDA applications. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 269–286. Springer, 2005.

[31] Tony C. Shan and Winnie W. Hua. Taxonomy of Java web application frameworks. In *ICEBE '06: Proceedings of the IEEE International Conference on e-Business Engineering*, pages 378–385, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Diomidis Spinellis. Notable design patterns for Domain-Specific Languages. *J. Syst. Softw.*, 56(1):91–99, 2001.

[33] Eelco Visser. Scannerless generalized-LR parsing. Technical report, Programming Research Group, University of Amsterdam, July 1997.

[34] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[35] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[36] Eelco Visser. Domain-specific language engineering. In R. Lämmel and J. Saraiva, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'07)*, Lecture Notes in Computer Science. Springer Verlag, Braga, Portugal, July 2007.

[37] Joost Visser and Jeroen Scheerder. A quick introduction to SDF. *CWI*, April 2000.

[38] Jos Warmer. A model driven software factory using domain specific languages. In *Model Driven Architecture – Foundations and Applications*, volume 4530, pages 194–203. Springer-Verlag, 2007.

Appendix A

Java utilities reference

The Java utilities and Echo2 extensions used in the implementation of the Echo2 mapping are described here. This is not an exhaustive list. Neither is it meant as a complete documentation of the classes. It is merely a reference for use in reading this thesis.

Layout extends ContentPane (abstract): Has one abstract method, `load`, that should position the initial Switches on the screen. The Pane Location DSL creates a subclass of Layout, implementing this method.
LayoutManager: A stateful class that contains the ContentPanes and Switches used. It is mostly used to make one ContentPane switch to another.
Link extends Button: Looks like an HTML link (blue and underlined) and has an ActionListener, switching pages when clicked.
ModelContentPane extends ContentPane (abstract): A ContentPane to which multiple components can be added (as opposed to one). It puts these components in a Table that has one column, adding a row for every array of components it receives as a parameter to its `add` method.
MultiPane extends SplitPane: A SplitPane that can contain multiple ContentPanes (as opposed to two), adding them recursively. A fairly complex screen layout can be accomplished by combining a number of horizontally and vertically oriented MultiPanes.
MyApplicationInstance extends ApplicationInstance: Is created for each user session and holds state information for that session. Through MyApplicationInstance all other stateful classes, like the LayoutManager, can be reached.
MyEntityManager: Offers database access.
PageCreator (abstract): Analogous to a WebDSL page. A PageCreator defines which ModelContentPanes to put in which Switches.
SergEcho2Servlet extends WebContainerServlet: Creates a new MyApplicationInstance for each user session.
Styles: A utility class that contains a number of styling methods, like `setLinkStyle(Component c)`.
Switch extends ContentPane: A ContentPane that can switch to another ContentPane.
Template: Analogous to a WebDSL page template. A Template is used for delta updating. It has hooks in the form of Switches and knows how to find their default definitions.