

Separation of Concerns in Web User Interface Design

Masters Thesis, September 21, 2008



Jippe Holwerda

Separation of Concerns in Web User Interface Design

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jippe Holwerda
born in Bergen NH, the Netherlands



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



Atos Origin
Papendorpseweg 93
Utrecht, the Netherlands
www.nl.atosorigin.com

Separation of Concerns in Web User Interface Design

Author: Jippe Holwerda
Student id: 1015265
Email: mail@jippeholwerda.nl

Abstract

Web User interfaces must be able to handle the output of the web application itself and the input from its users. For this reason, user interfaces have to cope with the complexity of both the application and the users. Furthermore, user interface design is an expensive, complex, and time consuming process. Specifying the appearance of web applications is too low-level due to a lack of good abstractions. This results in undesired style diversity throughout the web application and a lot of style duplication, making the web application difficult to maintain and decreasing reusability. This thesis shows an approach to overcome these problems by separating the various user interface concerns and addressing them separately. This is done by developing language extensions for the specification of layout and style and by separating presentation from document structure. The language extensions aim to achieve separation of concerns while still having an integrated language with static verification. Furthermore, web applications are commonly styled using Cascading Style Sheets (CSS). CSS suffers from a number of shortcomings such as missing and excessive functionality and poor design. The language extensions as developed in this thesis overcome these problems.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Company supervisor:	Drs. C.J. Labrie, Atos Origin
Committee Member:	Dr. ir. C. van der Mast, Faculty EEMCS, TU Delft

Preface

From the first moment I saw a presentation about WebDSL, it had my interest. Therefore, I would like to thank Eelco Visser for offering an excellent environment in which to do my thesis project. Although doing a masters thesis is an individual track, working on WebDSL still felt like working in a team. In that respect, I would like to thank the other WebDSL developers Zef Hemel, Danny Groenewegen, Sander van der Burg, Lennart Kats, Wouter Mouw, Sander Vermolen and Michel Weststrate for their support, help and tips during my project. I would like to thank my parents for offering me the opportunity to study and for their understanding and love through thick and thin. Furthermore, a big thanks to all family and friends for their support and interest. Last but not least, I would like to thank my girlfriend for motivating me and for her never ending support. I would never have been able to finish this without her.

Jippe Holwerda
Delft, the Netherlands
September 21, 2008

Contents

Preface	iii
Contents	iv
List of Figures	vii
1 Introduction	1
1.1 Problem Statement	3
1.2 Proposed Solution	3
1.3 Contributions	5
1.4 Overview of Chapters	5
2 Web User Interfaces	7
2.1 Structure	7
2.2 Behavior	9
2.3 Presentation	10
2.4 Appearance	10
2.5 Related Work	11
2.6 Conclusion	21
3 WebDSL	23
3.1 Introduction to DSLs	23
3.2 WebDSL	24
3.3 Data Model	25
3.4 User Interface	26
3.5 Actions	28
3.6 Access Control	29
3.7 Workflow	30
4 Separating Presentation	33
4.1 A New Template Mechanism for WebDSL	34

4.2	Recursion	35
4.3	Template Overloading	36
4.4	Implementation	37
4.5	Conclusion	43
5	Layout and Styling Extensions to WebDSL	45
5.1	Layout	45
5.2	Style	46
5.3	Related Work	46
5.4	Layout Syntax and Semantics	51
5.5	Style Syntax and Semantics	55
5.6	Implementation	61
5.7	Conclusion	62
6	Evaluation	65
6.1	Questions	65
6.2	Case Study 1: ING Card	66
6.3	Case Study 2: OSB	74
6.4	Case Study 3: Petclinic	78
6.5	Conclusion	82
7	Conclusions and Future Work	83
7.1	Conclusions	83
7.2	Future work	84
	Bibliography	87
A	Overview of Style Properties	91

List of Figures

1.1	The three building blocks of the World Wide Web	1
2.1	Overview of concepts of user interfaces	8
2.2	Various user interface models	13
3.1	The WebDSL generator	24
3.2	WebDSL entities example	26
4.1	Page displaying a list of authors	33
4.2	Page structure	34
4.3	Simple list of authors	34
4.4	Table of authors	34
4.5	Recursive data structure	35
4.6	Showing a <i>Document</i>	35
4.7	Template for showing a <i>Section</i>	35
4.8	Showing a <i>Paragraph</i>	36
4.9	<i>showDocument</i> with dynamic overloading	36
4.10	The Java Server Faces life cycle [11]	38
4.11	Translation of a template to Facelets	40
4.12	Mapping from a WebDSL page to Facelets	41
4.13	Example of passing arguments to a template	42
4.14	Example of using a passed argument	42
5.1	Pseudo syntax for the Layout Extension	52
5.2	A concrete example of the usage of the layout syntax	52
5.3	Pseudo syntax for selectors	53
5.4	Examples of selectors	53
5.5	Float statement in action	54
5.6	Layout example	55
5.7	Pseudo syntax for the Style Extension	56
5.8	A concrete example of the usage of the style syntax	57

5.9	Example of the addition of style expressions	59
5.10	Styling stages	61
6.1	Impression of the ING card case study	66
6.2	Layout skeleton of the ING Card case	66
6.3	WebDSL code of the form	67
6.4	A form in the ING card case	67
6.5	A form in the ING card case	68
6.6	A form in the ING card case	68
6.7	Sidebar menu	69
6.8	WebDSL code of the sidebar menu	69
6.9	Styling code of the sidebar menu	70
6.10	A form in the ING card case	70
6.11	A form in the ING card case	71
6.12	The customer page in the ING card case	71
6.13	customerDetails template	72
6.14	Layout for customerDetails template	72
6.15	customerDetails forms	72
6.16	customerDetails forms with small browser viewport	72
6.17	Styling code for the tab bar	73
6.18	OSB case study	74
6.19	Layout of the OSB case study	75
6.20	Layout code for the OSB case study	75
6.21	Style code for the navigation templates	76
6.22	Two navigation templates	76
6.23	Style code for the navigation templates	76
6.24	A form in the OSB case	76
6.25	Style code for the form in the OSB case	77
6.26	Petclinic case study	78
6.27	Layout of the Petclinic case study	78
6.28	List of navigate elements in the footer	79
6.29	Style code for the navigation elements in the footer	80
6.30	Table in the Petclinic case study	80
6.31	Style code for the list inside a table	80
6.32	WebDSL code for the list inside a table	81

Chapter 1

Introduction

The World Wide Web is taking a more prominent place in our lives every day. Whereas the original function of the web was to display static information in the form of documents, nowadays it is used to check the current news, read email, make appointments, maintain a social network, order the latest books and so on. While the usage of the web may have changed, the elementary techniques have not. The core of the web is still made up of Uniform Resource Identifiers (URI), the Hypertext Transport Protocol (HTTP), and the Hypertext Markup Language (HTML) [45].



Figure 1.1: The three building blocks of the World Wide Web

Hypertext connects various sections of text to form a dynamic organization of information, instead of the traditional way of displaying a large piece of text as one static page. The connections between pieces of hypertext are called hyperlinks. A hyperlink can link to a particular section in the same document or to another document and is often represented as a navigational element such as an HTML anchor.

The World Wide Web (WWW) is made up of various resources. In order to access these resources, they need to be identifiable. In other words, a uniform naming scheme for locating resources on the web is necessary. For this purpose, Uniform Resource Identifiers (URIs) are used. A URI consists of three parts [45]:

- The naming scheme of the mechanism used to access the resource.
- The name of the machine hosting the resource.

- The name of the resource itself, given as a path.

These three parts can be identified in the following example:

`http://www.webdsl.org/webdslorg/page/Download`

which states that the resource `Download` which can be found on the machine `www.webdsl.org` via the path `webdslorg/page` is accessible through the HTTP protocol.

The Hypertext Transfer Protocol (HTTP) [6] is a generic, stateless, application-level protocol that supports information exchange on the internet. A user agent, often a web browser, sends a request to a web server, which replies with the requested information.

HTML [37] is a markup language for the description of web pages. Originally intended to describe the structure of text documents, it now functions as the underlying technology of practically every web page, including the most interactive and dynamic of web applications. One of the design goals is that HTML documents should work well across different browsers and platforms. Achieving interoperability lowers costs since only one version of a document has to be constructed. A web site is a collection of HTML documents or web pages that inform users on a specific subject. The pages are requested through and displayed in a web browser. A web application is similar to a web site, with the exception that a web application is used frequently by a specific group of people in order to perform certain tasks, whereas a web site is often visited only once. Examples of web applications are Google Mail, Flickr, Amazon, Hyves, etc.

The user interface is arguably one of the most important parts of a web application. Not so much in a technical sense, but more from a users point of view. No matter what function a web application fulfills, what business process it supports, how many people it aims to connect, or how technically ingenious it is, if it is not usable people will be reluctant to use it. As the name suggests, the user interface is the users main point of interaction with the web application. It is the first thing he or she sees, and as an old cliché states: the first impression is the most important! The role a user plays on the web has changed. Nowadays, the user acts as an application user, information consumer, information provider and would like to be entertained on the internet [4]. The web is no longer read-only, but is full of user-generated content. In addition, the web is becoming more and more a social platform. In order to perform all these tasks, users demand simple, intuitive, interfaces that should look great as well. This poses great challenges to web developers.

The user interface of a web application consists of various concerns. For instance, the presentation specifies in what form content or an application's structure is presented to the user. The layout specifies how components on a page are arranged. And the appearance deals with things as colors, fonts and borders. Each of these terms are often used (incorrectly) to indicate the user interface as a whole. Chapter 2 will discuss these concerns in more detail and establish a common vocabulary.

User interface design is a complex, time consuming and therefore expensive process. To give an indication of the efforts needed to design user interfaces, a study [3] shows that an average of 48% of the application code is necessary for describing the user interface. Furthermore, the average time spent on the user interface portion is 45% during the design phase, 50% during the implementation phase, and 37% during the maintenance phase. No

such data is available for web applications in particular, but one may assume that because the appearance of web applications has a larger impact on users than that of desktop applications (since for web applications it is easier to close the browser or switch to the competition) the efforts for constructing web user interfaces can only be greater. Furthermore, especially in data-intensive websites there is a lot of code repetition. Basic CRUD (Create, Read, Update and Delete of data entities) operations in particular cause code repetitions in the user interface. Another issue is that popular scripting languages like PHP, although simple and easy to use, suffer from the lack of support for the development of complex and reliable software systems [21]. This is due to a lack of high-level abstractions and no static type checking. Styling suffers from the same issues as well. Styling of web applications is too low-level due to a lack of good abstractions, resulting in unwanted style diversity throughout the web application, a lot of style repetition and makes web applications hard to maintain. Often, the styling process is influenced by style guidelines and certain design principles. Most of these guidelines and design principles are hard to translate into code, and good user interface design is often evaluated according to human aesthetics and experience.

1.1 Problem Statement

As indicated in the introduction above, web user interface design is a time consuming, expensive, and complex software design process. User interface design is both art and science in that we use objective and subjective design metrics to evaluate interfaces. User interfaces for web applications need to be able to look good on multiple devices with varying screen sizes. New corporate identities means the look and feel of a web site or application has to be updated accordingly while the structure and content often remains intact. Furthermore, styling is often neglected in the design process as well as in model-based code generation, which could make it necessary to style the generated code. This results in styling code that is hard to maintain and reuse.

The main research questions of this thesis are:

- Q1:** How can the level of abstraction for describing and styling web-based user interfaces be raised in order to reduce development efforts while improving maintainability, reusability and quality?
- Q2:** Using WebDSL as a case study, what are good abstractions for describing the user interface of web applications and how can these abstraction be generalized for other environments?
- Q3:** How much of the user interface for web applications can eventually be generated using WebDSL on real-life case studies from the business domain.

1.2 Proposed Solution

In order to solve the research questions stated above, this thesis proposes to separate the various concerns involved in describing web user interfaces. Addressing the various concerns separately allows for better reusability and maintainability. Separation means that a

particular concern can be easily replaced by a different implementation and that it can be taken out of its original context and placed in a different one without much hassle. Giving a site a new appearance or layout without having to adapt the structure considerably reduces maintenance efforts. Separation of concerns also contributes to a reduction of the complexity of designing web user interfaces. An iterative development process is particularly well-suited to support the design process. First, the user interface can be defined by describing the structure of all pages. In later iterations, the user interface can be refined by adding presentation elements to support the page's structure, adding layout and styling the appearance.

Separation of concerns can be achieved by developing domain-specific languages that support the concepts related to that specific concern. This thesis uses WebDSL as a solution to the research questions and to support the separation of concerns in web user interfaces. By capturing user interface domain knowledge into a domain-specific language and by abstracting from low-level details, the quality of a user interface can be increased. For instance, usability guidelines and best practices can be easily incorporated into the language. Furthermore, frequently occurring code patterns can be captured by abstractions and integrated into the domain-specific language as well. Since the scope of the user interface of web applications is rather wide, this thesis will focus on presentation, layout and appearance aspects of the user interface.

1.2.1 Extending WebDSL

In previous work [44], WebDSL has been developed. WebDSL is a domain-specific language for the development of web applications with a rich data model. The language supports separation of concerns by providing sublanguages catering for the different technical domains of web engineering. This thesis will therefore describe extensions to WebDSL to support:

- the description of a page's structure;
- in what form that structure is presented to the users;
- how the elements a page is composed of are arranged (layout);
- how the elements are styled.

One of the threats of separation of concerns is that the various concerns are hard to integrate into a coherent definition. However, the sublanguages of WebDSL are unified at compile-time to form a single cohesive core language. Linguistic integration of these sublanguages ensures seamless integration of the aspects comprising the definition of a web application.

1.2.2 Case Studies

In collaboration with Atos Origin, two case studies have been selected from actual client projects of Atos Origin. These case studies are used in order to validate the approach taken

in this thesis and to validate it against projects from the 'real world'. In addition, the PetClinic is an application Atos Origin uses to introduce new techniques. The PetClinic applications will therefore be included in the case studies.

1.3 Contributions

This thesis makes the following contributions. First, it makes a clear distinction between the various concerns in web user interfaces and introduces proper terms for these concerns. Second, it gives a description of the design of sublanguages for expressing the presentation, layout and style of web pages as separate concerns. This is realized by means of a transformational semantics that reduces separately defined aspects into an integrated implementation. Next, the concept of separation of concerns is used to reduce development effort and increase the quality, reusability and maintainability of web user interfaces. Finally, the thesis describes a number of real-life case studies from the corporate domain in order to evaluate the proposed solution.

1.4 Overview of Chapters

The remainder of this thesis is structured as follows. Firstly, Chapter 2 will introduce the various concerns involved in the specification of web user interfaces. Background information and related work will also be discussed in that chapter. Chapter 3 will describe WebDSL and its sublanguages for access control and workflow. Chapter 4 explains how the current WebDSL template mechanism is extended in order to facilitate the separation of presentation from its structure. The implementation of the template mechanism used to accomplish this is explained as well. The sublanguages for the description of layout and style, as an extension to WebDSL, is described in Chapter 5. How the proposed techniques, as described in this thesis, can be used to specify user interface for three case studies will be shown in Chapter 6. Finally, Chapter 7 will conclude this thesis.

Chapter 2

Web User Interfaces

User interfaces process the output of applications and the input from its users. For this reason, user interfaces have to cope with the complexity of both the application and the users. Graphical user interface design is a time consuming, expensive, and complex software design process. User interface design is both art and science in that we use both objective and subjective design metrics to evaluate interfaces.

The design of web based user interfaces is influenced by several interrelated factors: the nature and domain of the application, a users' domain knowledge, the users' comfort level with computers in general, and how closely the application needs to match the mental models that users already have of the domain. Another important concept in web user interfaces is usability. In [33], nine usability principles are identified. The heuristics that determine good usability of a user interface are: simple and natural dialog; speak the user's language; minimize user memory load; be consistent; provide feedback; provide clearly marked exits; provide shortcuts; good error messages; prevent errors. A good user interface design will have to adhere to these usability principles.

Unfortunately, the terminology for describing web user interfaces is often confusing. Therefore, in this chapter, concepts used in the description of web user interfaces are introduced. The terminology introduced in this chapter will be used throughout the rest of this thesis. Figure 2.1 gives a graphical overview of the various concepts involved in the form of a four-layered user interface model, which is based on the models described in [4] and [17, 12]. This model will be used as a reference model to compare a number of User Interface Description Languages and approaches to generating user interfaces, as discussed in Section 2.5, as well as the final solution as described in this thesis. Some more background information regarding these models will be provided in Section 2.5. In the following sections, the four layers of Figure 2.1, the Structure layer, Behavior layer, Presentation Layer and Appearance Layer, will be discussed in greater depth.

2.1 Structure

As can be seen from figure 2.1, the Structure tier consists of the Conceptual Model, Task Flow and the Organizational Model. This is the most abstract tier, and provides the back-

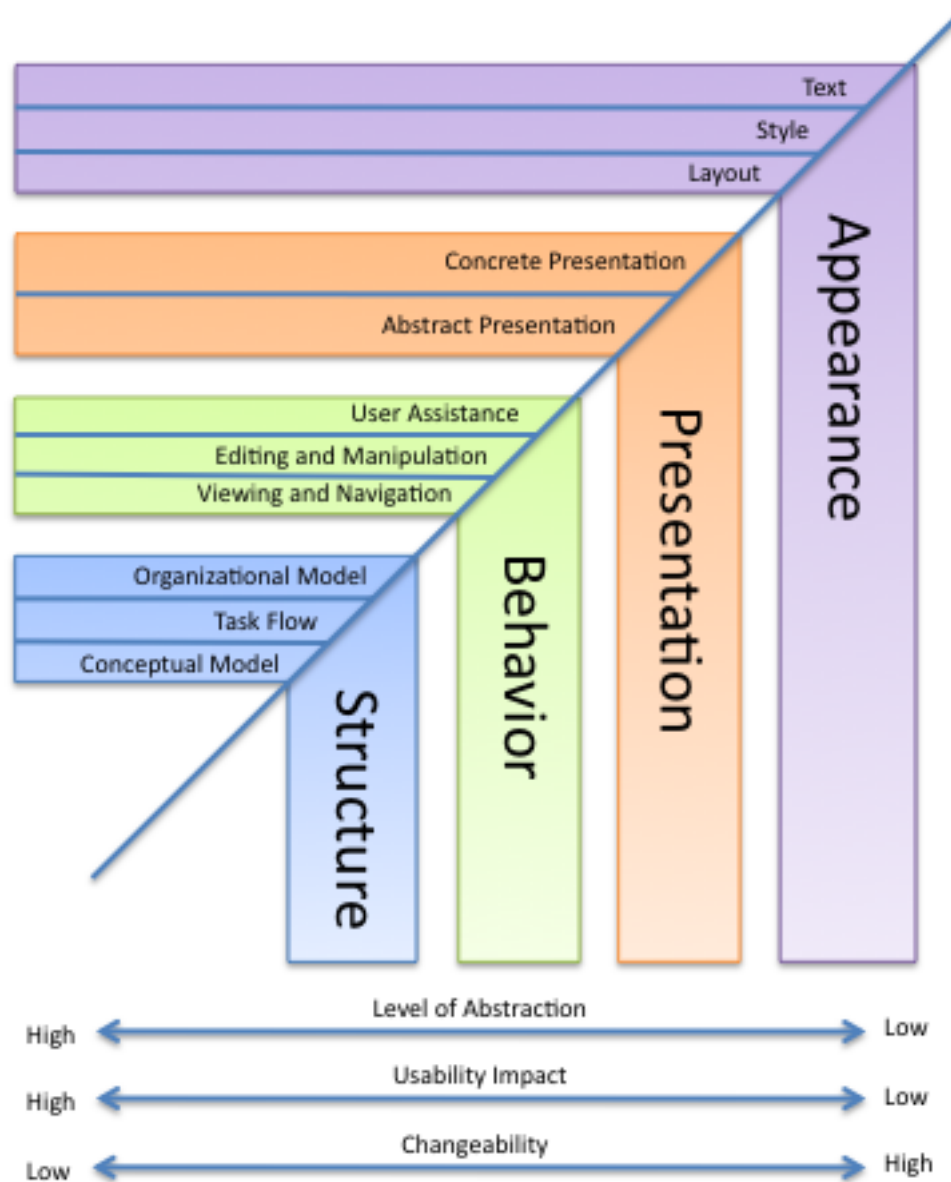


Figure 2.1: Overview of concepts of user interfaces

bone of the web user interface. Because the Structure tier is at the highest level of abstraction, users will not be able to perceive it directly, although a good structure will lay the foundations for the rest of the user interface.

Conceptual Model

The Conceptual Model describes the relationship between the user interface and the universe of discourse, or the domain of the web application. This model is often composed of a number of constructors, which instantiate objects as instances of classes. These classes represent concepts from the application or domain model. Classes are defined having properties, services which specify particular behavior a class should have and connections to other classes.

Task Flow

The Task Flow describes how users perform specific operations with the application. A task is defined as a number of steps needed to accomplish a certain goal, the conditions in terms of object state before and after a task executes and an ordering among these tasks.

Organizational Model

The Organizational Model deals with issues regarding the ordering and categorization of content and functionality, the classification scheme, the labeling systems, the search systems, thesauri, controlled vocabularies and metadata. These concepts come from the field of Information Architecture [32].

2.2 Behavior

Whereas the Structure tier deals with the conceptual aspects of the web application, the Behavior tier is concerned with the interactive aspects, such as viewing and navigating a site, editing and manipulating the data and user assistance. In this tier, the developer has to be concerned with user's actions as well as the system's response.

Viewing and Navigation

The Viewing and Navigation layer is concerned with all operations and behavior that allow users to navigate through a web site or application and to affect its presentation. Behavior in this layer does not affect any persistent data.

High-level questions that can be asked here are questions such as: 'Where am I?', 'Where did I come from?', 'What can I do here?' and 'Where can I go?'. Users have more control over their navigation on the web in comparison with desktop applications. The user can navigate through pages, and can take paths never intended, such as jumping directly to a page without going to the home page or defining bookmarks. Examples of viewing or navigation behavior are navigating between pages, sorting data, browsing through a paginated list of data and selecting items from a menu.

Editing and Manipulation

The Editing and Manipulation layer specifies behavior that does affect persistent data. Behavior in this layer often requires an implicit or explicit save operation and typically requires validation of input data.

User Assistance

User Assistance deals with informing users of the application's activity and status. This can include error messages, validation messages, status alerts and online help.

2.3 Presentation

Presentation is often confused with layout and/or style, but is a very different concept indeed. Presentation is concerned with how to present content or structure to the user without concerning for layout (how a page's elements are arranged) or appearance (does it look pretty). Content can be static text or data resulting from actions in the behavior tier, such as queries, browsing through persisted data, etc. By presenting structure, one can think of a representation of a site's navigational structure in the form of a menu hierarchy. A web site's presentation consists of an Abstract Presentation model and a Concrete Presentation model.

Abstract Presentation

The Abstract Presentation layer specifies the structure or content of the user interface without specifying how this is realized. It gives a conceptual description of the visual parts of the user interface. It consists of elements that model the inputs and outputs of a task. An example of such an element is the concept of selecting one option out of a list of multiple options. The elements used in the description of the abstract presentation should avoid any premature commitment to a specific visual representation and should be independent of the modality (e.g. graphical, aural, etc.).

Concrete Presentation

In the Concrete Presentation layer, the Abstract Presentation as described above is made concrete through the definition of elements that turn the elements of the abstract presentation into visual user interface components. Examples are radio buttons, checkboxes and text input fields, which are concrete elements for the abstract concepts of selecting one option out of multiple, multiple options out of multiple and for the input of text, respectively.

2.4 Appearance

The Appearance tier is concerned with the visual appearance of the user interface. The Appearance tier comprises three concepts: Layout, Style and Text. Although all three concepts contribute to the visual appearance, they still are separate concerns. For instance, a specific

layout consisting of a header, footer, body and sidebar may be reused across several web applications. Or when a web application needs a new look, only the style has to be changed. Separation of these concerns enhances reuse and maintainability.

Layout

Layout is concerned with the arrangement of elements on a page. It has to deal with different screen sizes. Screen size varies with devices and with time, so this is an important aspect. Positioning of elements can either be done in an absolute or relative manner. Relative positioning, however, offers more flexibility and scales better. Furthermore, there are standard arrangements such as flow, border and grid, into which components can be placed. A more precise definition is provided in Section 5.1.

Style

Modeling a user interface consists of modeling the basic elements to appear in the user interface and styling their appearance. This can range from defining background colors, setting borders or choosing a specific font. Important goals of styling a web application are: enforce a consistent style across a large collection of documents; provide the ability to present a document or page specific to the end-user or the capabilities of the presentation device without altering the document itself. Since the style of a web application is the first thing users will see, it performs a very important function. It is also a very time-consuming activity. For instance, it is very difficult to specify a uniform corporate identity or house style throughout a website. Style is defined in more detail in Section 5.2.

Text

The Text layer represents all written, language-based elements of the user interface. These includes the labeling system used to represent the organizational model, names of input and navigation controls and the User Assistance elements named in the description of the Behavior tier. Internationalization, the ability to present a web site to users of different nationalities, also belongs in this layer.

2.5 Related Work

In this section, related work with respect to user interface models and user interface generation will be discussed.

As mentioned above, the user interface model of Figure 2.1 is a combination of the models described in [4] and [17, 12]. The model as described in [4] is essentially the same as the model in Figure 2.1, except for the Presentation Tier. Section 2.5.1 will give an overview of the most important user interface description languages that can be used as input for methods that automatically generate user interfaces. Section 2.5.3 discusses a number of languages that can act as the most concrete level of the user interface.

2.5.1 Declarative User Interface Models

The model-based user interface development paradigm uses a declarative specification from which the user interface is automatically generated. In this context, declarative means that the user interface is described in a model rather than by programmatically stating how to build it. According to [41, 17], describing user interfaces declaratively has a number of advantages, namely:

- being able to model at a high level of abstraction, since implementation details are not part of the model;
- the ability to model at different levels of abstraction, since models can be incrementally refined;
- reuse elements of a model;
- and to provide the foundations for generation of user interfaces.

As a shortcoming, the complexity of the models and their notations is often mentioned [17].

All facets of an application, from the input from users to the output of the business logic, come together in the user interface. To cope with this complexity, a number of different models can be distinguished [17, 12]:

Application Model: Also known as Concept Model [39] or Domain Model [34], it describes which aspects of the application for which the user interface is being defined are relevant.

Task-Dialog Model: Describes which tasks users are able to perform. A task is specified as the steps necessary to accomplish a certain goal, the conditions in terms of object states before and after a task executes and an ordering that tasks or sub-tasks must respect. Interactions such as clicking on a button or checking a radio button are the lowest level of user tasks.

Abstract Presentation Model: Specifies the structure or content of the user interface without specifying how this is realized. It consists of Abstract Interactor Objects (AIOs) that model the inputs and outputs of a task. An interactor is an abstract representation of a user task or user interaction [12]. An example of an abstract interactor is the concept of selecting one option out of a list of multiple options. The description is independent of the concrete interactors available in the target user interface libraries. AIOs are abstractions of Concrete Interaction Objects (CIOs), but avoid premature commitment to a specific visual presentation.

Concrete Presentation Model: In the Concrete Presentation Model (CPM), the Abstract Presentation Model is made concrete through the definition of a hierarchy of Concrete Interactor Objects (CIOs). A Concrete Interaction Object (CIO) is defined as any user interface entity that users can perceive [30]. The CPM is platform or toolkit independent. Concrete interactors are visual user interface components. Examples

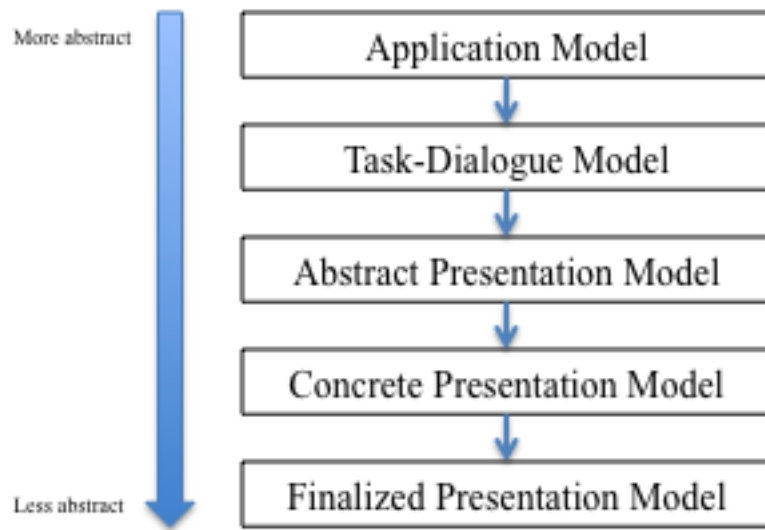


Figure 2.2: Various user interface models

are radio buttons, checkboxes and text input fields, which are concrete interactors for the abstract interactors for selecting one option out of multiple, multiple options out of multiple and for the input of a text, respectively. Layout and style information is often added to this model, although preferably in a separate model as to comply with the principle of separation of concerns.

Finalized Presentation Model: The Finalized Presentation Model is the actual implementation of the Concrete Presentation Model in terms of a programming language or graphical toolkit. It is made up of the components available in the target user interface libraries or toolkits, such as JSF, Flex or HTML. This model is often expressed as source code. Section 2.5.3 will describe a number of these languages.

The models described above can be seen graphically in Figure 2.2 in decreasing level of abstraction. At the highest level of abstraction are the Application and Task-Dialog Models. At the bottom the Finalized Presentation Model, which is the actual implementation of the user interface and which is executable.

User Interface Description Languages

In this section, a number of User Interface Description Languages (UIDL) are introduced. A definition of a UIDL is given by [38]:

A UIDL consists of a high-level computer language for describing characteristics of interest of a user interface with respect to the rest of an interactive application.

In this context, the above definition is slightly extended with the requirement that the description could act as a model for User Interface generation. The most important goals the various UIDLs pursue are:

- Increasing portability of user interfaces.
- Describe user interface requirements at a high level of abstraction.
- Increase the reusability of user interfaces.
- Use the high level user interface model to enable generation of user interface code.

UsiXML UsiXML [29] matches with Figure 2.2 in having a Final User Interface (FUI), a Concrete User Interface (CUI), an Abstract User Interface (AUI) and a Task & Concepts level. The Task & Concepts level is composed of a task model, a domain model and a context model. The domain model has the form of a UML class diagram, along with the well-known concepts of classes, attributes, methods and relationships. The context model captures the relevant entities of the user's context that may influence a particular task. It further consists of a user model, a platform model and an environment model. UsiXML allows for the definition of graph transformations between the various models in order to support continuous and seamless development of user interfaces. UsiXML is structured into four levels of abstraction that do not need to be specified in order to construct the user interface. Its strong point is that because of the definition of transformations between models, it is one of the mostly integrated User Interface Description Languages that is independent of platform, modality and context.

UIML The User Interface Markup Language (UIML) [1] is a meta language for the description of user interfaces in a device-independent way. The UIML specification is currently being standardized by OASIS. A UIML document consists of five sections: description, structure, data, style and events. In the description section, the elements which make up the application's user interface are listed. The structure section indicates which elements from the description are present for a given appliance. It also specifies how the elements are organized. The data section describes which information is presented to the user. This information is device-independent. The style section contains the mappings from interface elements to widgets that are specific for a given toolkit through the specification of rendering attributes. Finally, the events section defines user input events and actions resulting from those events. The renderer can either interpret UIML on the client device or it can translated the UIML code into another language like HTML. Another important feature of UIML is its capability to define connections to the backend logic. UIML can be summarized in being a declarative language that distinguishes which user interfaces are present in an interface, what the structure of the elements are for a variety of devices, what natural language text should be used with the interface, how the interface is presented or rendered and how events are to be handled for each user interface element. So, for the development of a user interface for another device, only the style section has to be changed. The rest of

the UIML document can be reused. However, the downside is that user interfaces for different devices still have to be created by hand. Furthermore, there is not a clean separation of structure and presentation.

XIML eXtensible Interface Markup Language (XIML) [35] aims to standardize the efforts for representing and manipulating data that defines and relates all relevant elements of a user interface. The main requirements of XIML are: a central, structured storage mechanism for interaction data; the language should support all phases of the design of a user interface; it should support abstract and concrete elements; the language should support mappings between various elements, i.e. from abstract to concrete elements; and finally the language should be independent of particular methodologies or tools. The structure of XIML is composed of components, relations and attributes. XIML can be seen as a collection of interface elements, grouped into interface components. Typical examples of components are user tasks, domain objects, user types and presentation elements. A relation links multiple elements within or across components. Attributes define properties of elements which can be assigned a value of a predefined type or as an instance of an existing element.

2.5.2 Approaches to Generating User Interfaces

Models are used to describe a portion of the world, usually called the universe of discourse. Having non-ambiguous, precise semantics, models abstract from the underlying modeling domain, creating a greater understanding of the artifact being modeled. Models can either be textual or graphical. Models can be used particularly well as input in an approach for the generation of user interfaces. This section will discuss two of these approaches: Model-based User Interface Development and WebML. WebDSL also fits in this category, but deserves a separate chapter. WebDSL will be covered in Chapter 3.

Model-based User Interface Development

Model-based user interface development provides a way for user interfaces to either be executed or transformed into source code from a high-level declarative model. Developers build user interfaces by building models that describe the desired user interface, rather than by writing a program that performs the same tasks. The advantages of describing user interfaces with declarative models are already discussed in sectionsec:declarative-ui-models, so it would be interesting to see if there are concrete methods that actually implement these ideas in the generation of user interface code. In this section, a number of Model Based User Interface Development Environments (MB-UIDEs) that support this process are described. MB-UIDEs provide an environment in which a number of complementary declarative models can be used for the purpose of user interface generation [18]. MB-UIDEs have been around for quite some time, dating back to the end of the last century. Even though they don't specifically target web user interface, there is still a lot to learn from the methods.

The use of MB-UIDEs offers some benefits [41], namely:

- The declarative model is a common representation of the user interface that tools can use to make statements of. For instance, it is possible to use design assistants, or design critics which detect faults in designs.
- Increased consistency and reusability. Since all components of the system use the same model, interface consistency is promoted. Furthermore, certain model elements can easily be reused since the models abstract from implementation details.
- Iterative development and support for rapid prototyping. Models can be defined in increasing levels of abstraction, which makes it ideal for iterative development. Also, since not all details need to be known in advance, the method is very suitable for making early prototypes.

However, [41] also identifies a few shortcomings: a lack of flexibility because the modeling language is not expressive enough; difficult to use because specialized modeling skills are needed, especially when compared to visual interface builders.

MASTERMIND MASTERMIND [41] is intended to support the rapid production of high quality and advanced user interfaces. An implementation is described in [10] where the three most important types of models that are supported are: a dialog, interaction and presentation model. In addition, there is a user interface state model and a context model. Furthermore, the interface between the user interface and the application must be specified by the designer. The Dialog Model (DM) plays the role of the Task Model, and describes the various low-level input activities that may be performed by the user. This includes their relative orderings and the resulting effects they have on the presentation and the application. The Interaction Model (IM) specifies the set of low-level interactions between the end user and the MASTERMIND runtime environment. A Presentation Model (PM) is built up from smaller presentation parts, called presentation objects. Presentation objects can have standard parameters or application-specific parameters. Standard parameters can be things like font or color, while application-specific parameters can be a list of objects to display. Things like font, color and layout are attributes of a presentation object. Furthermore, MASTERMIND supports constraints to define layouts. The MASTERMIND models are expressed in the MASTERMIND Dialog Language (MDL) [40] which is an extension of the Corba IDL. MASTERMIND uses the three distinct models to generate C++ source files which are compiled and linked with a number of runtime libraries.

Teallach Teallach [19] uses task, domain and presentation models to describe user interfaces. The models can be constructed in any order. In the presentation model, abstract and concrete grouping objects are used to combine interaction objects. Teallach targets the Java platform using the Swing widget library. Furthermore, it uses an extensive class library in order to reduce the complexity of the generation of the user interface code. This way, complex classes do not have to be generated and instead, the complexity is captured in easy to manage libraries. A unique feature of Teallach is that the generated java code can interact with existing object databases through the domain model interface. In addition, it is possible to use existing Swing interaction objects that are accessed through the presentation model.

Mecano The Mecano Project [34] is a research effort for a model-based user interface development environment that extends the notion of automating interface design from data models. Mecano uses a domain model to generate automatically both the static layout and the dynamic behavior of an interface. The Mecano project consists of two phases. In the first phase, a generic interface model is developed. The second phase comprises the implementation of an open model-based development environment based on such an interface model. The results of phase one are the Mecano Interface Model (MIM) and its associated interface modeling language (MIMIC). The second phase has resulted in MOBI-D, a model-based development environment for user-centered design [36]. MIMIC is an object-oriented modeling language that follows the general principles of C++, and is even implemented in C++. The main models MIM consists of are a user-task model, domain model, presentation model and dialog model. In addition, MIMIC can specify a design model, which defines a connection between user tasks, domain objects and presentation elements. Furthermore, a user model can be defined which is just a collection of users. The presentation model is formed by examining objects in the domain model and assigning widgets to them. The widgets then are placed on its corresponding window by a layout algorithm that observes interface design guidelines. Mecano produces textual interface model instances that are implemented by a run-time system. The runtime system then executes the refined model as a running interface. Mecano already has been successfully applied in the field of medical applications to generate user interface of relatively large and complex applications.

WebML

The Web Modeling Language (WebML) [13] is a notation for specifying complex web sites at the conceptual level. A WebML model consists of a structural model, a hypertext model, which in turn consists of a composition model and a navigation model, a presentation model and a personalization model. WebML models can be defined using an XML-based textual notation. Furthermore, it is possible to represent models visually, by using the graphical notation. The graphic representation is suitable for use in development tools, while the textual notation can be used for generating code for the web site. WebML aims to support advanced features like multi-device access, personalization and evolution management.

WebML is only a language specification. There is a WebML implementation in the form of a commercial product named WebRatio Site Development Studio. WebML supports the following models:

Structural model: The structural model describes the data content of the website in terms of entities and relationships among those entities. WebML is compatible with notations like the entity-relationship model and UML class diagrams. Entities consist of attributes, each with an associated type. Attributes whose value is calculated from other attributes are called derived attributes. The notation for expressing calculated or redundant information is a simplified Object Query Language notation. The structural model resembles the application model of Figure 2.2.

Hypertext model: The hypertext model specifies which pages compose the hypertext. The basic building blocks of a hypertext are site views, areas, pages and content units

(ranging from a coarse to a fine granularity, respectively). A hypertext defines a site view. A site view addresses a specific set of requirements and consists of areas which in turn are made out of sub-areas or pages. Once the site view is designed in terms of pages and areas, the following two models are constructed.

Composition model: Content units are the elementary building blocks of which a page is composed of and represent actual information from data sources that is displayed to the user. There are six types of content units: data, multidata, index, filter, scroller and direct units. Data units are used to display single objects, while the rest of the content units work on a set of objects. The multidata unit enables to select a single object from a set of multiple objects and display information about that object. The index unit outputs a whole set of objects, by denoting each object as an entry in a list. A filter unit allows a given set of objects to be filtered according to some search criteria, such as a given search string. Usually, a filter unit is used in conjunction with an index unit. A scroller unit is used to scroll through a collection of objects, showing the details of one object at a time. Finally, the direct unit is a syntactic construct to specify a one-to-one relationship between two objects. The data, multidata, index and scroller units include a source, which is the name of an entity from which the results are collected, and a selector, which is a predicate indicating which of the collection of source entities make up the actual objects. Together, the content units should be able to logically express arbitrary content for a web application.

Navigation model: The navigation model defines the topology of links between pages. It shows how pages and content units form the hypertext. Links can either be contextual or non-contextual. A link is contextual if it passes an identifier of an object displayed by the source unit from the source to the destination unit. This identifier is the binding between the source and destination unit, and is represented by a link parameter. Links between pages are non-contextual.

Presentation model: Presentation is the task of defining the look and feel of pages in a site view. It is orthogonal to the other models. The presentation model represents the layout and graphic requirements for page rendering, independently of the output device and rendering language. The page is the basic unit of presentation and each page can be linked to multiple style sheets. Style sheets are formally specified by means of an abstract XML syntax. The style sheet language is complemented by the following sublanguages [14]:

- A sublanguage to define the regions of the screen that can be used for the presentation of a page.
- A sublanguage to define the internal structure of screen regions, based on an extended notion of a grid.
- A sublanguage to define presentation panels, such as the content of a screen region. The goal of this sublanguage is to allow the designer to define panels at a varying level of granularity to allow the construction of reusable style sheet libraries.

Personalization model: Personalization is the definition of user specific data according to a user's profile data. This can be defined not only at the content level but also at the user interface layout and navigation levels. The WebML constructs that can be defined according to user-specific data are content units, pages, site views and the presentation style of a page. Furthermore, there are two complementary ways of specifying personalization [14]:

- Declarative personalization: content can be specified according to certain derived information taking the user's profile data into account. An example of this is the recommendation section of Amazon.com. This list of recommended books is composed according to the user's purchase history. Another example is a personal message to the user on it's birthday.
- Procedural personalization: it is also possible to specify certain business rules that compute and store user-specific information. A business rule can monitor some event. When the event fires, some precondition is checked and action is taken when the condition is true. WebML supports an XML syntax for describing business rules. An example is the assignment of users to some user group based on dynamically collected data.

2.5.3 User Interface Target Languages

In this section, a number of languages are presented that can act as the most concrete level, the finalized presentation level, of user interface descriptions in the generation process. Java Server Faces also belongs in this section but is treated more extensively in Section 4.4.1 since it is part of the WebDSL seam backend, as described in Chapter 3. The reason JSF is chosen as implementation language is that in the end it yields HTML which is supported by all browsers. This cannot be said of the languages described here. By targeting XForms or XUL as the finalized presentation model, the level of abstraction is slightly raised, because some of the complexity is captured inside the browser. This can result in less complex transformations, since the gap that needs to be bridged between the concrete presentation layer and the finalized presentation layer is smaller. However, bad browser support make these languages a poor choice.

Programming the User Interface

The current practice for graphical user interface is to be specified by creating objects, calling methods to place them in the correct position in a window, and then linking them to code that will process any actions required. User interfaces for Java (non-web) applications are often written this way. If coded by hand, such a process is tedious and error-prone; if a builder or designer program is used, hundreds of lines of code are generated. The generated code cannot be modified by hand, without destroying the end result. Either approach violates the software engineering principles of efficiency and maintainability.

JSF

Java Server Faces (JSF) is a user interface framework for Java web applications. Its goal is to simplify writing and maintaining web applications that run on a Java application server and render their UIs back to a target client. JSF is described in more detail in Section 4.4.1.

XUL

The main aim of XUL, the XML User Interface Language [?], is to develop cross platform applications. Its scope is therefore broader than most declarative user interface languages, which mostly focus on describing web applications. It provides the same rich user interface and user experience as desktop applications. XUL makes a clear separation between the client application definition and programmatic logic, presentation and language-specific text labels.

A XUL document consists of a set of structured elements, along with a predefined list of attributes. Scripts are added that allow interaction with the user. XUL describes the user interface at the level of the abstract user interface. The concrete user interface is often the result of interpreting the abstract user interface at the browser level.

The main selling points of XUL are: separation of presentation from logic makes applications less susceptible to change; it is very easy to port XUL applications to other platforms; XUL provides a rich set of UI components for describing (web) applications. One of the downsides of XUL is that event-handling code is embedded within the user interface declaration, which breaks the separation of concerns between the user interface and the program logic.

XForms

XForms defines a consistent, declarative language for the description of dynamic forms. One of the key goals is to decouple data, logic and presentation. XForms is an embeddable XML language and can be used inside another XML language. It is comprised of the XForms Model, which is a device-independent XML form definition, and of the XForms User Interface, which provides a standard set of visual controls for describing a document's user interface. XForms captures form data as XML instance data. The XForms Model describes the structure of the instance data. Furthermore, the XForms Submit Protocol specifies how XForms send and receive data. A unique feature is the ability to suspend and resume the completion of a form.

By using XForms, it is possible to separate the data representation in the client from that of the server. For example, form data is represented as XML in the client side, while on the server side it is represented in programming language constructs native to the server software.

A number of interesting features of XForms are: perform data validation on the client; submit the same form to different servers; save and restore form values to and from a file; calculate submitted values from other values; use the result of a submit as input to a further form. There already are a number of XForms implementations, including plugins for the major web browsers.

2.6 Conclusion

The terminology for describing web user interfaces is often vague and confusing. The aim of this chapter was therefore to introduce the correct terms and describe the various concerns that a web user interface consists of. Figure 2.1 clearly shows all different aspects of a web user interface. The model in Figure 2.1 describing the concerns in web user interfaces has merged various concepts described in the related work section.

Now that all concerns have been identified, the rest of this thesis can focus on how language extensions for the description of these concerns can be developed and used. Emphasis will lie on language extensions for style and layout and on the separation of presentation from document structure.

The next chapter will introduce WebDSL for which the language extensions will be developed.

Chapter 3

WebDSL

In this thesis, WebDSL is extended with a sublanguage for styling web applications and specifying layout. This chapter will explain WebDSL, and the elements it is composed of. First, this chapter begins by providing a short introduction to Domain-Specific Languages (DSLs). Next, the WebDSL data model and entity definitions are discussed. Then, the user interface is covered by describing page and template definitions. The next section shows how operations are performed on data using actions. Finally, access control and workflow definitions, two recent contributions to WebDSL, are discussed. When referring to WebDSL, the current version at the time of this writing is implied.

3.1 Introduction to DSLs

With the ever increasing size and complexity of software projects, a lot of software projects exceed their budget and delivery dates. A possible explanation for this is the fact that a lot of things can go wrong in the translation of the problem description in terms of the customer to a solution described by technical terms.

To cope with the increasing complexity, the abstraction level at which software is developed needs to be raised. Domain-specific languages [42, 43, 31] try to accomplish this by specifying an, often declarative, language with high expressive power, focusing on a specific problem domain. A problem domain can be a vertical domain or a technical domain. A vertical domain focuses on a vertical market in which customers share common needs and problems share common concepts. Examples of such vertical markets are markets that deal with insurances, pensions and banking matters. A technical domain focuses on certain technical aspects of the world, such as databases, web applications, user interfaces and computer graphics. Domains can also intersect, for instance when a DSL focuses on web applications for handling insurance requests.

So, domain-specific languages differ from general purpose languages, such as C, C++ and Java, in that DSLs focus on a narrow problem domain, while general purpose languages, as the name suggests, aim at generic approaches that provide general solutions. DSLs have a number of advantages [42]. Because solutions described by a DSL are expressed in terms of and at the level of abstraction of the problem domain, domain experts can understand and

validate DSL models. DSLs increase productivity, reliability, maintainability and portability. DSL programs are concise, self-documenting and encapsulate domain knowledge that can be reused and preserved.

Application Programming Interfaces (APIs) try to raise the level of abstraction as well, by offering a set of services to accomplish specific goals. They try to hide the implementation and lower level details as much as possible from the software developer. In this way, APIs encapsulate domain knowledge as well. The main difference with a DSL however, is that a DSL uses a syntax specifically tailored towards the problem domain, whereas APIs are often described by general purpose programming languages. The expressive power of a DSL is thus far greater and solutions described by a DSL can be easily communicated to all actors involved.

Examples of domain-specific languages are SQL, HTML, CSS and WebDSL.

3.2 WebDSL

Problems identified in the development of web applications are that often a web application consists of a number of different languages, each addressing a specific issue. For instance, a Java web application can be based on the Seam framework, use SQL for querying the database, JSF for specifying web pages and CSS for styling the pages. These languages are loosely coupled, making it very difficult for a web application to be checked for errors at compile time. Furthermore, a lot of code is boilerplate code and a lot of code consist of slightly different patterns there are used over again. WebDSL [44] tries to overcome these issues by offering a set of integrated Domain-Specific Languages. WebDSL targets interactive dynamic web applications with a rich application-specific data model. WebDSL consists of multiple sublanguages that describe different aspects of web applications. For instance, it has sublanguages for describing the data model, pages, access control [20] and workflow [25]. The various sublanguages are unified at compile-time to form a single cohesive core language. WebDSL aims at, and al-

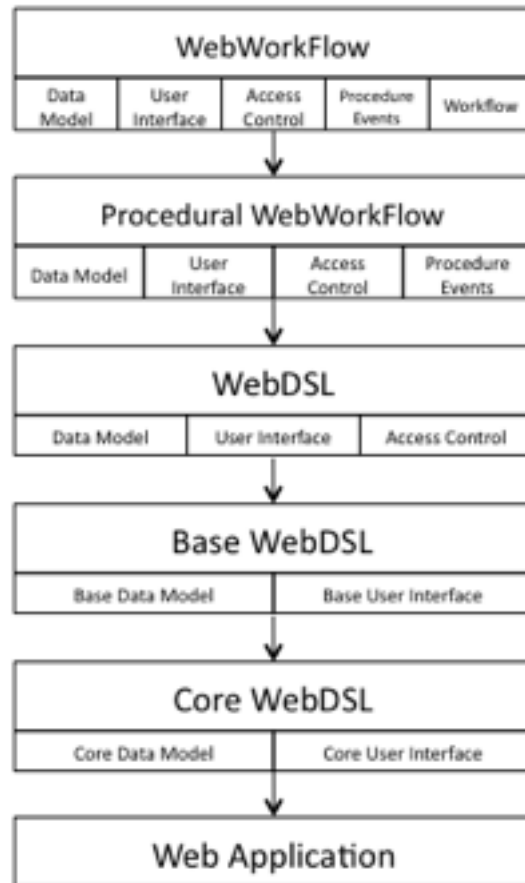


Figure 3.1: The WebDSL generator

ready achieved, an order of magnitude reduction in web application code. WebDSL offers a core language upon which higher-level abstraction or even other DSLs can be built that are transformed back to the core language.

WebDSL is implemented by a code generator that transforms a WebDSL application into an executable web application targeted at a specific web platform. At the time of writing, WebDSL can target Java web applications based on the Seam framework, Python web applications and Java web applications without using Seam, but relying on lower level Java Servlets instead. The latter two were developed during the final phase of this thesis, and this thesis will therefore focus on the Seam back-end. By back-end is meant the final translation of Core WebDSL to an executable application, as can be seen in figure 3.1. WebDSL is implemented as a model transformation pipeline, which can be seen in figure 3.1. The code generator itself is implemented with the syntax definition formalism SDF [23] and the transformation language Stratego [9]. These will not be discussed in detail. WebDSL definitions are translated to an application based on the Java architecture using the Seam framework. Additional frameworks that the generated application is based on are the Java Persistence API (JPA) which in turn uses Hibernate as the implementing technology and Java Server Faces (JSF). For each entity definition, a corresponding entity class is generated with fields, getters and setters for the entity's properties. The entity class is annotated for object-relational mapping according to the JPA as implemented by Hibernate. For each page definition, a JSF XHTML page, a Seam Java bean class and an accompanying interface are generated.

In the following sections, the various sublanguages for defining the data model, describing pages, declaring access control and describing workflows are discussed more thoroughly. Chapter 5 will describe the styling sublanguage that is an integral part of this thesis.

3.3 Data Model

The WebDSL data model consists of entity definitions, declaring entity types. Entities themselves consist of properties with a name and a type. Properties can either have value types or associations to other entities in the data model. Associations can be composite, where the referer owns the object, or referential in case the object is shared. Value types and a referential association are indicated by `::` and `->`, respectively. A composite association is indicated by `<>`. Deleting an object which has a composite association to another object will cause the other object to be deleted as well. This is called a cascading delete operation. Examples of built-in value types are String, Date, Int and Bool, but also domain-specific types such as WikiText, Password and Image, that have extra functionality and semantics. Collection types are used to define one-to-many and many-to-many relationships between entities and can refer to either value types or defined entities.

The inverse annotation can be used to declare a relation between two properties of different entities, which will synchronize them whenever changes are made to either of them. Additional annotations that can be set on properties are: name, unique and required. Setting a name annotation has the effect of adding a derived property name. By convention, any WebDSL object can be asked for its name, which provides a generic interface to naming an

```
entity Owner {
  name :: String (name)
  address :: String
  city :: String
  telephone :: String
  pets -> Set<Pet>
}

entity Pet {
  name :: String (name)
  birthdate :: Date
  type -> PetType
  visits -> Set<Visit>
  owner -> Owner (inverse=Owner.pets)
}
```

Figure 3.2: WebDSL entities example

object, for instance when producing links. A unique annotation ensures that the value of a property for all entities of a particular type is unique. A requires annotation implies that an entity cannot be persisted when a property with a required annotation has no value.

Entities can be extended with extra properties by using `extend entity`. This performs a simple insertion of the extra properties in the original entity declaration in a model-to-model transformation step. A special type of entity is the session entity. This entity becomes a (not persisted) singleton in the session of the user with the WebDSL application. Figure 3.2 shows an example of two WebDSL entities.

3.4 User Interface

The current design of WebDSL is page-oriented, with actions and navigations leading to requests of new pages. Page definitions are the main building blocks for the description of the user interface. Pages are composed of basic markup elements and templates. Both pages and templates will be discussed in more detail below.

3.4.1 Pages

Page definitions consist of the name of the page, page parameters represented by objects with a name and a type, and a presentation of the data contained in the parameter objects. For defining the basic structure of a page, WebDSL provides basic markup elements representing concepts such as section, header, list, listitem, table, row and title. The markup is solely intended to describe the structure of the document. Data from the parameter objects can be displayed by using the output element. The output element uses the type of the property in order to determine by what lower-level (HTML) presentation construct the property has to be represented. For example, in the current implementation of WebDSL, using the

output element with a property of type Password results in a HTML component showing the plain text representation of the password (which consists of only stars, of course). Collections can be displayed by iterating over them using the `for` construct. A simple example of a page that shows a list of Owner entities could be:

```
define page allOwner() {  
  header { "Show all owners" }  
  list() {  
    for (o : Owner) {  
      listitem { text(o.name) }  
    }  
  }  
}
```

In addition to the displaying of data, it is also possible to use pages for collecting data. For this purpose, forms can be defined that collect input data and send it to the server for further processing. In the same way the output element can be used for displaying data, the input element is used for collecting data. For example, using the input element on a property of type String yields a simple text input component within a form, while an input element for a property of type Date results in a calendar component from which a particular date can be selected.

3.4.2 Templates

In order to increase reuse of code within the language, it is possible to define templates. Templates allow a general setup for a page that is shared among multiple pages of the WebDSL application. For instance, a main template can be constructed that defines the web sites structure consisting of a header, sidebar, body and footer:

```
template main() {  
  header()  
  sidebar()  
  body()  
  footer()  
}
```

When the template calls inside the main template have sensible defaults, and the main template is called from a page, it automatically gets the default template behavior. For instance, the default sidebar template could consist of:

```
define sidebar() {  
  header { "Default Side Menu" }  
  
  block { navigate(findOwner()) { "Find owner" } }  
  block { navigate(vets()) { "Display veterinarians" } }  
}
```

A template definition closely follows the definition of a page, and consists of a name, object parameters and markup elements. In essence, page definitions are a special form of template definitions with the restriction that pages cannot be included inside other pages.

Templates, on the contrary, may call other templates. Furthermore, templates called from other templates may be redefined locally in order to redefine their behavior:

```
define page home() {
  header { "Side Menu" }

  main()

  define sidebar() {
    header { "Home Side Menu" }
    menubar {
      menu {
        menuheader { navigate(findOwner()) { "Find owner" } }
      }
      menu {
        menuheader { navigate(vets()) { "Display veterinarians" } }
      }
    }
  }
}
```

3.4.3 Styling

WebDSL currently does not support styling of web applications in the language itself, since the markup elements are only used to describe the page structure. It is possible to style the generated code by defining style rules for the lowest-level presentation code, in this case ordinary HTML. CSS, for instance, can be used for this purpose.

3.5 Actions

A WebDSL application can have operations that modify data as well. These are defined in actions, which support a Java-like imperative language for defining operations with a simple API. The actions are nested in pages or templates:

```
define page createOwner() {
  title("Create new owner")

  main()
  define body() {
    var o : Owner;
    header{ "Create new owner" }
    form {
      group("Owner details") {
        groupitem { label("Name:") { input(o.name) } }
        groupitem { label("Address:") { input(o.address) } }
      }
      group("More details") {
        groupitem { label("City:") { input(o.city) } }
        groupitem { label("Telephone:") { input(o.telephone) } }
        groupitem { label("Pets:") { input(o.pets) } }
      }
    }
  }
}
```



```

    group {
        action("Save", save())
        action("Cancel", cancel())
    }
}

action save() {
    o.save();
    return owner(o);
}
action cancel() {
    cancel home();
}
}
}

```

A form is needed to provide a connection between a button or link and an action. The `input` element can be used to connect data from variables to form elements. These are submitted and synchronized with the data of the page before the action is executed. A `return` statement is used to control the page flow once an action is executed. In the example above, the user is redirected to the view page of the owner after it has been saved. The `cancel` statement, on the other hand, does not save the form and redirects the user to the home page.

3.6 Access Control

WebDSL has a declarative sublanguage for defining access control [20]. Access control rules define who can access a page, see a template or perform an action. The sublanguage supports the definition of a wide range of access control policies concisely and transparently as a separate concern. Access is governed to the various resources in a WebDSL application, most importantly the pages and page actions. Rules are woven into the corresponding WebDSL components during compilation. For instance, when a user does not have access rights to edit an entity, the properties can only be viewed and not changed. Navigation options are influenced based on page checks, which results in links to inaccessible pages being hidden. Rules can be combined into policies to easily introduce exceptions to the normal rules, such as an administrator mode. The following example shows access control in action:

```

section data model

    entity User {
        name :: String
        password :: Secret
    }

access control rules

    principal is User with credentials username, password

    rule page pet(p : Pet) {

```

```

    true
  }

  rule editPet(p : Pet) {
    principal = p.owner
  }

```

The first statement in the example above states that the security principle is being represented by a User entity with the username and password properties as credentials. The first access control rule gives unlimited access to the `pet` page, because the condition specified for that rule always evaluates to true. The second access control rule states that it is only editable when the principle is the owner of the pet.

3.7 Workflow

Besides access control, WebDSL also provides a workflow modeling sublanguage for the high-level description of workflow in web applications. From the definition of procedures operating on objects, and a control flow description to connect these procedures, complete custom web applications can be generated.

Workflow descriptions define procedures operating on domain objects. Procedures are composed using sequential and concurrent process combinators. The extension is implemented by means of model-to-model transformations, as can be seen from figure 3.1. Rather than providing an exclusive workflow language, WebWorkFlow supports interaction with the underlying WebDSL language. Most of the basic workflow control patterns are supported.

A workflow procedure describes activities to be performed by one or more participants (who) in a particular order (process), under certain conditions (when). A procedure may consist of a single step, or may be a composition of other procedures. For activities that require data entry views can be defined and action code to be executed (do). A workflow description generates task lists, a status page and navigation rules. An example of the workflow sublanguage can be seen below.

```

procedure meeting(p : ProgressMeeting) {
  process {
    (writeEmployeeView(p) |AND| writeManagerView(p));
    repeat {
      writeReport(p);
      (approveReport(p) |XOR| commentReport(p))
    } until finalizeReport(p)
  }
}

procedure commentReport(p : ProgressMeeting) {
  who { principal = p.employee }
  when { ... }
  view {
    derive procedurePage from p
    for (view(employee),

```

```
        view(report),  
        comments)  
    }  
    do { email(commentNotification(p)); }  
}
```


Chapter 4

Separating Presentation

As described in Section 3.4.2, templates are an important WebDSL concept in order to reduce boilerplate code and increase reuse. Chapter 2, more particularly Section 2.4, already described the various tiers involved in the description of web user interfaces. The presentation tier is an important aspect in this description. This chapter focuses on the separation of a page's presentation from its structure.

There are a couple of reasons why this is good. Firstly, it increases the possibility to reuse pages. For instance, consider the code fragment of Figure 4.1.

```
define page authors(l : List<Author>) {  
  main()  
  
  define body {  
    for (a : Author in l) {  
      text(a.name)  
    }  
  }  
}
```

Figure 4.1: Page displaying a list of authors

The page shown in Figure 4.1 simply iterates over a list of authors and displays each author's name. However, what if we want to show the list of authors in a different way? For instance, by showing it in a table, showing it in a paginated list or simply by printing the names separated by commas. These are all different representations of the same structure: a page which shows a list of authors. The code examples in Figure 4.2, 4.3 and 4.4 show a more flexible way of defining a page and defining the representation of the list of authors separately. Figure 4.2 now only shows the structure of the page without specifying in what way to output the list of authors. This was already possible in WebDSL since the generator would check the type of the argument of the output template call and desugar the call to the appropriate WebDSL code. The problem with this approach is that when the default behavior of the output template call for a specific type has to be changed, the generator

itself has to be adapted to accommodate this. Obviously, we can do better than that and it is desirable to be able to specify and even override standard output behavior. Figure 4.3 and 4.4 are examples of two different implementations of the output of a list of authors. This improved flexibility does come at a price: since default output behavior is not desugared in the generator anymore, it has to be specified manually for every data type for which output is called. A solution for this is to develop a library of WebDSL code that can be imported inside new WebDSL applications which includes standard implementations of output behavior for various data types.

```
define page authors(authors : List<Author>) {  
  main()  
  
  define body {  
    output(authors)  
  }  
}
```

Figure 4.2: Page structure

```
define output(l : List<Author>)  
{  
  for (a : Author in l) {  
    output(a.name)  
  }  
}
```

Figure 4.3: Simple list of authors

```
define output(l : List<Author>)  
{  
  table {  
    for (a : Author in l) {  
      row(output(a.name))  
    }  
  }  
}
```

Figure 4.4: Table of authors

Furthermore, separating page structure from presentation also supports a more iterative development process. When defining pages, only the more abstract structure has to be described, without having to commit to early implementation choices as how to represent certain aspects of the structure.

4.1 A New Template Mechanism for WebDSL

The current implementation of templates in WebDSL, where template calls are expanded inside pages as described in section 3.4.2, has a few drawbacks. The first issue is that it is possible to define recursive data structures but that it is not possible to have recursive template calls on instances of those data structures. Furthermore, related to the recursion problem, it is not possible for templates to be overloaded; i.e. templates having the same name with different signatures. Section 4.2 and 4.3 respectively will cover these issues.

In order to achieve the goals as stated at the beginning of this chapter and in order to overcome the two issues described above, the WebDSL template mechanism has to be reimplemented. This chapter will discuss the new implementation.

4.2 Recursion

The current WebDSL syntax allows the definition of recursive data structures. Consider for example the WebDSL code of figure 4.5 where there are three entities defined: *Document*, *Section* and *Paragraph*, with *Section* and *Paragraph* subtypes of *Document*. Notice that *Section* has a recursive property named *content* which consists of a list of entities of type *Document*. This data structure could be part of the representation of a book which is composed of documents which in turn is composed of sections and paragraphs. Sections can consist of other sections as well as paragraphs.

```
entity Document {}

entity Section : Document {
  title  :: String
  content <> List<Document>
}

entity Paragraph : Document {
  text :: Text
}
```

Figure 4.5: Recursive data structure

When defining the user interface for this demonstration application, one might imagine to have a page or template which takes a document as argument and displays its properties. The code fragments of figure 4.6 and 4.7 illustrate this.

```
define showDocument(d : Document) {
  if (d is a Section) {
    showDocument(d as Section)
  }
  if (d is a Paragraph) {
    showDocument(d as Paragraph)
  }
  if (d is a Document) {
    text("This is a document") } }
```

Figure 4.6: Showing a *Document*

```
define showDocument(s : Section) {
  section {
    header {output(s.title)}
    for (d : Document in s.content) {
      showDocument(d)
    }
  }
}
```

Figure 4.7: Template for showing a *Section*

The template in figure 4.6 takes a document as argument. It then checks the type of the argument and depending on the type casts the argument to that type and calls *showDocument* again. The template in figure 4.7 takes an argument of type *Section*. It prints the title of the section, iterates over the *content* property, which is a list of documents, and calls *showDocument* for each of them. This is a very elegant way of specifying templates over a recursive data structure. For completeness, figure 4.8 also shows the code of showing a paragraph.

The reason this document example will not compile with the current version of WebDSL is that templates are expanded inside pages at generation time. For each template call, the called template is retrieved, variables in it are renamed, and the template call is substituted by the template code itself. However, generating the document example as provided here will result in the generator looping endlessly. This is because the generator handles the recursion on the template structure instead of on the data itself. Since the template structure is recursive, the generator will loop. The solution to this problem is to postpone the handling of the recursive templates to run time, when the data is available. This means that the template structure cannot be expanded at generation time and will have to remain intact until run time. The details of this solution will be described in the sections below.

```
define showDocument(p : Paragraph) {  
  section {  
    output(p.text)  
  }  
}
```

Figure 4.8: Showing a *Paragraph*

4.3 Template Overloading

Another issue with the current template mechanism in WebDSL is that template overloading is not possible. Template overloading, in this context, means that templates can be defined having the same name but with a different number of arguments or with arguments of different types. The same code fragments as in section 4.2 about recursion apply here as well.

Template overloading resolution can be implemented statically or dynamically. In this context, static overloading means that at generation time for each template call it is decided which template is called by their signature, i.e. the combination of template name and the types of its arguments. A simple implementation would be to rename all templates having the same name and rename all template calls to the new names by comparing template signatures of the call and the defined template.

Dynamic overloading in this case means that the decision of which template to call is postponed until run time. Java will then check the type of the arguments and based on that the correct template is included. In case of dynamic overloading, the code of *showDocument* of figure 4.6 can be

```
define showDocument(d : Document) {  
  text("This is a document")  
}
```

Figure 4.9: *showDocument* with dynamic overloading

reduced to the code in figure 4.9. The rest of the example code remains unchanged. As can be seen from the examples, dynamic overloading even further reduces the amount of code needed to display a document. The manual casts shown in figure 4.6 are not needed anymore. Therefore, static overloading is a bit more verbose but has a very simple implementation. As indicated above, overloaded templates can simply be given a unique name in the model-to-model transformation phase of WebDSL. The downside of dynamic overloading is that it is much harder to implement.

Dynamic overloading in this case means that the decision which template to call is postponed until run time. Java will then check the type of the arguments and based on that, the correct template is included. In case of dynamic overloading, the code of *showDocument* of Figure 4.6 can be reduced to the code in figure 4.9. In this case, it is unnecessary to add casts and manually dispatch template calls based on an argument's type. Nevertheless, static overloading is chosen in the current implementation of WebDSL. The reasons for this are twofold. Firstly, the amount of code reduction for dynamic overloading compared to static overloading isn't worth the extra complexity of the implementation of dynamic overloading. Second, in not all cases can be determined which template to call based on an argument's type. For instance, when an overloaded template is defined as having an argument that takes a generic set of a particular entity, hibernate wraps that set in a custom class. The result is that any type information is lost. The only way to determine which types a hibernate wrapped set accepts is to check the type of the elements of that set. However, this still poses trouble for empty sets. Also, a generic set for a particular type can accept subtypes of that type, so then you still cannot know which types a hibernate wrapped set accepts when one knows the type of an element.

Nevertheless, just because overloading resolution is implemented statically, this doesn't mean that an implementation for dynamic overloading is impossible.

4.4 Implementation

Currently, work is in progress on different back-ends for WebDSL. Implementations based on Python and Java Servlets are being developed. However, at the time of implementing the new template mechanism described in this chapter, the only back-end available was the Seam back-end. The implementation described in this section will therefore only focus on the Seam back-end. The Seam back-end is implied when referring to the implementation.

The implementation is based on Java Server Faces (JSR 252) [11] and on an extension of JSF named Facelets. Facelets has an excellent template mechanism, which makes the mapping of WebDSL concepts to a Facelets implementation rather straightforward.

Below, a short introduction to Java Server Faces will be provided, after which Facelets and the actual implementation of WebDSL templates are discussed.

4.4.1 Java Server Faces

As already explained in section 3.2, the Seam implementation of the WebDSL back-end uses Java Server Faces [11] to translate page and template definitions to. JSF is a Java standard from the Java Community Process (JCP) program for presentation tier development. Its goal is to simplify writing and maintaining web applications that run on a Java application server and render their user interfaces back to a target client. JSF makes it easy to construct a user interface from a set of reusable components. It simplifies migration of application data to and from the user interface. Furthermore, it helps manage user interface state across server requests and provides a simple model for wiring client-generated events to server-side application code. Finally, it allows custom user interface components to be easily built and reused.

Each request that involves a JSF component tree goes through the JSF life cycle, which is made up of various phases. The JSF component tree is also known as the view, and consists of all the JSF components. The life cycle can be seen in Figure 4.10.

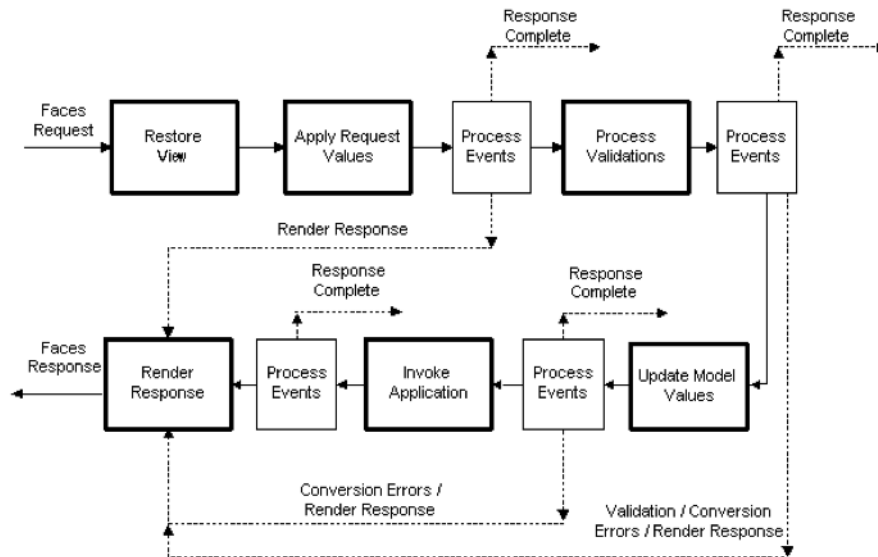


Figure 4.10: The Java Server Faces life cycle [11]

In the *Restore View* phase of the life cycle, a request is received from the FacesServlet controller. The request is examined and the view ID is extracted. The view ID is determined by the name of the requested page and is used to look up the components for the current view. This phase consists of three view instances: new view, initial view and postback. Each instance is handled differently. In case of a new view, JSF builds the view of the requested page and wires the event handlers and validators to the components. The view is saved in a FacesContext object. The FacesContext object contains all the state information needed in order to manage the JSF component's state for the current request in the current session. The FacesContext stores the view in its viewRoot property, which contains all the JSF components for the current view ID. The first time a page is loaded, an initial (empty) view is constructed. The empty view will be populated as the user causes events to occur. After the initial view is created, JSF advances directly to the *Render Response* phase. When the user returns to a page previously accessed (a postback), the view corresponding to the page already exists and only needs to be restored. In this case, JSF uses the existing view's state information to reconstruct its state. The next phase after a postback is the *Apply Request Values* phase.

In the *Apply Request Values* phase, the current state for each component is retrieved. First, the components are retrieved from the FacesContext, after which their values are retrieved. The values are retrieved from either the request parameters, cookies or headers.

The *Process Validations* phase is where the first event handling is taken care of. For each component, its values are validated against validation rules, which can be standard

rules belonging to a particular component or specified by the developer. Invalid values will result in error messages which are added to the FacesContext object. The component itself is marked as invalid, and JSF will proceed to the *render response* phase.

The next phase is the *Update Model Values* phase, where the properties of the backing beans that are bound to a component are updated. In the *Invoke Application* phase, the component values will have been converted, validated, and applied to the backing bean, so now they can be safely used to execute the application's business logic. The outcome of a particular action determines which view is processed next. In the last phase, the *Render Response* phase, the view with all its components in their current state is displayed.

One of the strong points of JSF is its extensibility. Facelets uses this extensibility to implement its features.

4.4.2 Facelets

Facelets [26] is a templating language built from the ground up with the JSF component life cycle in mind. With Facelets, it is possible to produce templates that build a component tree. This allows for greater reuse because components can be composed out of a composition of other components.

This section will describe the process of building a view in relation to Facelets. A JSF ViewHandler is a plug-in that handles the *Render Response* and *Restore View* phases of the JSF life cycle for different response-generation technologies. Facelets extends the JSF ViewHandler class with the FaceletViewHandler to tweak the rendering behavior and the strategy for saving and restoring the view state.

First, when a request comes in for a new view, the FaceletViewHandler creates a new UIViewRoot for this view. Next, the view needs to be populated with components. Before rendering the view, a Facelets instance is created by the FaceletFactory class that is used to populate the components of the view. Next, the UIViewRoot is rendered to the response, and the page is shown to the user. The essential information, such as form data, from the state of the component tree is saved for subsequent requests. In-line text and other transient components are not stored.

When a subsequent request comes to the JSF application, the view is restored with the state information saved from the previous request. This view is then passed through the JSF life cycle, which will end, depending on the action that was fired, in the creation of a new view or a rerendering of the same view (if there were validation errors). If the view is rerendered, the Facelets class is used to complement the restoring of the view with the unsaved in-line text and transient components. The UIViewRoot is then asked by the view handler to render itself to the response. This process is repeated for subsequent requests.

Facelets distinguishes templates and template clients. A template defines hooks where content can be inserted. Which content is used for those hooks is defined by the clients. In other words: a template is anything that uses a `ui:insert` tag to insert content. Template clients use `ui:component`, `ui:composition`, `ui:fragment` or `ui:decorate` tags. The `ui:define` tag is used in conjunction with the target template's `ui:insert` tag in order to insert the defined content into the template. Templating in Facelets is not just limited to one level. It is possible to have multilevel templating, as a client for one template can be a template for

other client templates. Hence, the Facelets template mechanism allows for the creation of complex composite applications.

4.4.3 WebDSL to Facelets

Since Facelets constructs the component tree for each request by compiling the templates during the render phase, it is possible to implement features in WebDSL such as overloading and recursion. The following sections show how the various WebDSL elements map to Facelets. This will first be shown for templates, since they have the finest granularity in this context. Next, pages will be covered followed by how arguments are passed.

Template to Facelets

Figure 4.11 shows the translation of a template named *main* to Facelets. Each template definition is translated to a `ui:composition` that resides in its own separate Facelets file named `{template name}.xhtml`. Template calls are translated to `ui:include` constructs, which simply include to contents of the `src` attribute. However, templates can be locally redefined. In that case, the local redefinition has to be used over the toplevel template definition in the construction of the component tree. This is solved by using the `ui:insert` tag, which specifies a hook where content defined using a `ui:define` tag and having the same name is inserted, as already explained in Section 4.4.2. When no `ui:define` is found, the included content of the toplevel template definition is used.

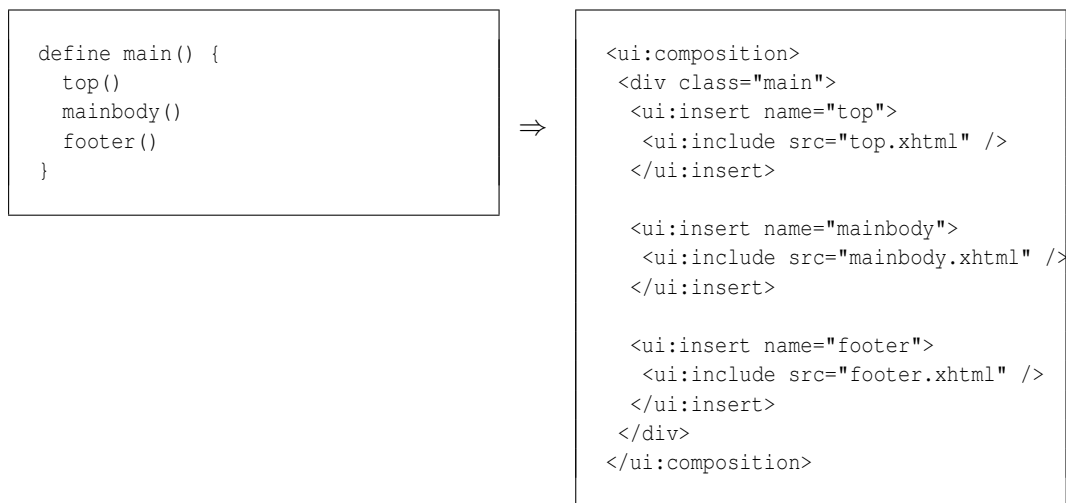


Figure 4.11: Translation of a template to Facelets

Page to Facelets

Figure 4.12 shows how a WebDSL page is translated to a Facelets template named *home-Template.xhtml*, and a Facelets template client named *home.xhtml*. The template provides hooks named `title` and `includes`. Furthermore, because the *home* page calls the template

main, all hooks specified in *main* are imported as well. These imported hooks can then be defined in the template client *home.xhtml*. The client uses *homeTemplate.xhtml* as a template by specifying it in the `template` attribute of a `ui:composition`. This attribute causes the `ui:insert` and `ui:define` tags to be linked together. In this example, a custom title, custom body and even which stylesheet must be included are specified.



Figure 4.12: Mapping from a WebDSL page to Facelets

Passing arguments to templates

It is possible to pass arguments from a page or template to another template, as can be seen in figure 4.13. A template that includes another template using the `ui:include` tag can pass parameters to the that template by specifying one or more `ui:param` tags, each with a specified value. In the example, a collection of `Owner` entities is iterated and for each `Owner` the template `ownerDetails` is called, which simple outputs the owner's properties. In the side

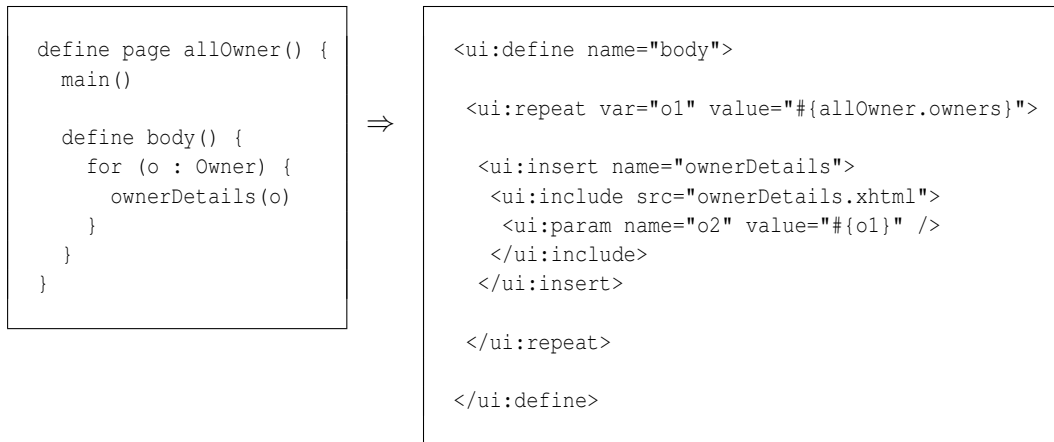


Figure 4.13: Example of passing arguments to a template

on the right of Figure 4.13 the owner is passed to the template `ownerDetails` as a parameter named `o2`. Inside the `ownerDetails` template the argument can be accessed by using this name, as can be seen in Figure 4.14. The parameter is renamed to have a unique name. This is necessary, since the argument name has to be the same for each call to `ownerDetails`.



Figure 4.14: Example of using a passed argument

4.5 Conclusion

This chapter focused on the separation of presentation from document structure. Presentation deals with the question as how document content and document structure is presented to the user. Treating presentation as a separate concern has the advantage that a developer only has to focus on the structure and content and does not have to worry about how it will look like, resulting in better structured documents of higher quality. Another advantage is increased reusability. It is easy to present the same page in different ways, since only the presentation has to be changed, and not the document structure itself. Being able to overload templates in WebDSL has helped the separation of presentation from structure. Pages or templates can simply be specified by input and output calls on entities. Input and output behavior can be specified separately for specific arguments.

Chapter 5

Layout and Styling Extensions to WebDSL

This chapter starts by providing a detailed explanation of what concepts layout and style precisely consist of. After the introduction, related styling languages are discussed along with an overview of the problems with CSS. This is important because the actual implementation of the layout and styling extensions to WebDSL rely on CSS. The chapter continues by describing the WebDSL extension for specifying layout. First, the syntax and semantics of the layout extension are shown, after which the same topics for the styling extension are covered. Next, the most important implementation issues will be explained. The chapter will end by presenting some final conclusions.

5.1 Layout

As already explained in chapter 2.4, layout deals with the arrangement of user interface components on a page. More specifically, the following requirements, which a language for specifying layout has to conform to, have been identified:

1. Specify the natural flow of elements in a page or region of a page.
2. Specify how one element is positioned in relation to other elements.

The traditional way of specifying layout using CSS has always been a hassle. The current CSS specifications were never designed to create the visually rich and complex interface layouts that modern web applications demand. The current methods, floats and positioning, were never intended as layout tools. However, there currently is no good alternative for specifying layout with CSS, as is discussed in Section 5.3. A common approach is to specify layout through HTML `<table>` tags. However, this practice suffers from a number of drawbacks. Firstly, embedding layout into pages makes them hard to reuse. If the same content has to be displayed with a different layout, the document has to be copied and the layout has to be altered for each individual page. Furthermore, using `<table>` tags to describe the layout may corrupt the structure of pages, giving the wrong semantics to tables

or losing valuable semantic information. This may result in aural (audio-only) browsers reading documents in the wrong order.

5.2 Style

What exactly is style? The World Wide Web Consortium (W3C) defines style as:

e.g. fonts, colors, spacing¹

This is a rather vague definition at best. This thesis will try to come up with a better definition. A first attempt may be to define style as any concept that deals with appearance and which does not deal with layout. Appearance is simply any concept that deals with what a page or document should look like.

Style is often specified in the form of a style sheet which captures style rules. A style sheet is a specification that describes how documents are presented to readers. Style sheets provide a mechanism for separating content from appearance in order to improve reuse and device-independence of content [45]. The following is a precise definition of a style sheet [28]:

A set of rules that associate stylistic properties and values with structural elements in a document, thereby expressing how to present the document. Style sheets generally do not contain content; are linkable from documents; and are reusable.

What is interesting in this definition is not the concept of style sheets, but rather the description of rules that associate stylistic properties and values with a pages' structural elements. This concept captures the essence of styling nicely. One may wonder what exactly is meant by these stylistic properties. A stylistic property, or style property, defines one aspect of the appearance of the element to be style. An overview of style properties supported by the styling extension described in Section 5.5 is given in Appendix A.

Since the average web user only visits a page for a couple of minutes, it is important to catch and keep the users attention and to make the user experience as good as possible. The styling of a web site can contribute to those issues and is therefore an important component of the web architecture [45]. Usability is an important aspect of web design quality but a sound aesthetic design can overcome deficiencies caused by usability problems [22].

5.3 Related Work

In the next sections, a number of styling languages are introduced that aim to solve the issues presented above. Of these, Cascading Style Sheets (CSS) is by far the most used in web development to date. Furthermore, the Extensible Style Language, another W3C standard, is discussed. Two lesser known style languages are presented next. Constraint Cascading Style Sheets (CCSS) is an extension of CSS which enables the use of constraints

¹<http://www.w3.org/Style/CSS>

to describe a style. The Presentation Specification Language (PSL) resembles CSS but has a couple of high-level abstractions. Finally, this section is concluded with the differences between the languages and some final remarks.

5.3.1 Cascading Style Sheets

The most widely used method of styling web documents these days is Cascading Style Sheets (CSS) [7]. CSS is used to attach style to structured documents, like HTML or XML documents. One of the key features of CSS is its aim to separate the presentation style of documents from their content. The basic components of CSS consist of style rules. A style rule is composed of a selector, a property and a value. The selector indicates which elements are affected by the rule. The property is the name of an identifier representing some aspect that can be styled. Each property has its own syntactic and semantic restrictions on the values it accepts.

Most properties are orthogonal and do not influence each other. However, especially for specifying layout, there are sets of properties that are related and together describe one particular layout aspect. The responsible rules are often associated with different components and can be scattered across the style sheet.

Rules can be grouped by selectors, by properties and by stylesheets. Rules sharing the same selector can be put into one block sharing the same selector. Some properties, such as `border`, can be composed of other properties. In this case `border-color`, `border-width` and `border-style`. This notation is compact but makes it more difficult to search for a specific property in a style sheet. Lastly, rules and blocks can be grouped in a stylesheet. One document may be linked to multiple stylesheets.

Another interesting feature of CSS is cascading. In CSS, a component can be affected by several rules each specifying the same property. The mechanism to resolve these conflicts is known as the cascade. It assigns a weight to each style rule. In case of a conflict, the rule with the greatest weight takes precedence.

The final value of a property is determined by calculating the following values:

- Specified value: the value that is specified in the stylesheet, if necessary as determined by the cascade. Otherwise, in case the property is inherited, use the computed value of the parent element. Otherwise, use the property's initial value.
- Computed value: during the cascade, specified values are resolved into computed values. For example, `em` values are computed to pixel or absolute lengths.
- Used value: some values can only be determined when the document is rendered. The computed value is transformed into an absolute value by resolving any remaining dependencies. For example, this is the case when the width of an element is set to be a percentage of the width of the containing element. Then, the width of the containing element first has to be determined.
- Actual value: if a used value cannot be used in some cases, for example when only black and white colors can be used while another color is specified, the actual value is used. It is determined by approximating the used value.

Problems with CSS

Although CSS is a major improvement compared to embedding style information inside documents, there still are a number of areas where CSS falls short [28]. A couple of examples are provided below.

Missing functionality: While one of the design goals of CSS has been simplicity, there could have been more features in a number of areas. To name a few: it is not possible to ensure a certain contrast between the text color and the background; it is not possible to specify text size in terms of the containing elements width; it is not possible to center an element vertically. CSS has always focused on styling documents. Styling user interfaces therefore falls a bit short, more specifically: multi-column layouts where content flows automatically from one column to another is not supported; headers and footers are poorly supported.

Excessive functionality: In contrast to missing features, excessive functionality can overly complicate a language. This will cause problems to authors of style sheets as well as implementers of the CSS specification, causing interoperability problems. For example, the box model includes padding, border and margin areas around elements. However, for inline elements a padding area would suffice. Also, properties like `text-shadow` and `first-line` are often difficult to implement and only add very limited functionality.

Poor design: While missing and excessive functionality is relatively easy to fix by respectively adding and removing functionality, poor design decisions are more difficult to overcome. Examples are the `white-space` and `text-decoration` properties that both describe multiple unrelated concerns. These separate concerns should be extracted and it should be possible to specify them individually. Another issue is positioning. Layout information is difficult to specify, and the positioning properties are not well suited for cascading.

Implementation problems: The most important problem of the acceptance of CSS on the web is its implementation in the different user agents. Especially the dominant web browser, Internet Explorer from Microsoft, lacks in its CSS support. It is a real hassle for web developers to support multiple browsers, since none of them supports the full CSS specification. One specific example is the support for contextual selectors, which allow elements to be selected based on their place in the document's tree structure. This demanded a complex implementation which delayed the acceptance of CSS1.

5.3.2 Extensible Style Language

The Extensible Style Language (XSL) is being developed by the W3C and consists of three specifications:

1. XSL Transformations (XSLT) [15] for transforming XML documents
2. XML Path Language (XPath) [16] to access or refer to parts of an XML document

3. XSL Formatting Objects [5] (formerly known as XSL-FO) for formatting XML documents

XSL is a transformation-based style sheet language and does not use declarative statements to describe the style of a tree, instead it transforms the logical structure into a presentational structure. XSL goes beyond CSS in some respects that are important for high-quality printing. For example, XSL offers multiple column layouts; can condition formatting on what is actually in the document; allows you to place footnotes, running headers, and other information in the margins of a page; supports page numbers and automatically cross-reference particular pages by number. An XSL document is passed to an XSL processor which formats the document for output. At the time of writing, no XSL processor exists for the major web browsers. Its primary use is for generating PDF documents from XML documents.

5.3.3 Constraint Cascading Style Sheets

Constraint Cascading Style Sheets (CCSS) [2] is an extension to CSS which combines style sheets with user preferences and browser restrictions in a more flexible and predictable way. CCSS uses constraints to declaratively specify the layout of a web document. It is possible to specify the layout partially. Partial specifications can be combined with other partial specifications in a predictable way, through the use of preconditions. Preconditions can specify if a certain style sheet is applicable. For instance, the precondition

```
@precondition Browser[width] >= 800px
```

in a style sheet indicates that this style sheet is only applicable to browsers with a width of 800 pixels or more. A constraint is a statement of a relation. Constraints result from browser capabilities, default layout behavior according to the type of element, the document tree structure and from the application of style rules. Each style property can be represented by a variable. CCSS supports equality or inequality constraints and allows constraints to refer to other variables. Variables can correspond to elements other than parent elements and can also be global. An example of simple layout constraints in CCSS:

```
@variable table-width;
table { constraint: width = table-width }

@constraint #c1[width] = #c2[width]
```

Constraints may conflict, this is allowed by using a constraint hierarchy. A constraint hierarchy consists of a collection of constraints, each labeled with a strength that denotes preference. CCSS is implemented using constraint solving technology. Given a system of constraints, the constraint solver must find a solution to the variables that satisfies the required constraints exactly, and satisfies the preferred constraints as well as possible.

A question remains how well the constraints system and solver scale to larger, more complicated designs. Furthermore, browsers must explicitly implement the constraint solver techniques, making the language hard to adopt.

5.3.4 Presentation Specification Language

The Presentation Specific Language (PSL) [27] is a declarative language that has the following design goals: independent of any medium; easy to use syntax and semantics; the language has very few special cases.

PSL provides three presentation services: property propagation, tree elaboration and box layout. Property propagation assigns values to the formatting properties of each element. Tree elaboration enables the addition of content to the presentation by adding elements to the document tree. For instance, it can be used to create borders around elements, or the precede list items with numbers or bullets. Tree elaborations consist of declarations and creation commands. Declarations specify the node types that can be generated and can either be of primitive type Content for text, Markup for tags and Graphic for graphical objects like lines, rectangles and circles. Elements are generated by the creation commands which create and attach elements to the defining element. Box layout is a constraint-based layout system. The box layout system is comprised of a model of nested boxes. Each element has a bounding box. A layout is constructed by defining constraints between bounding boxes of elements. It is possible to define a layout in which the order of the elements is different from the order in a traversal of the document tree.

PSL distinguishes between the specified and actual size of an element. Style rules have the form: `<property> = <expression> ;`. An expression can be anything that has the same datatype as the type of the property, including arithmetic, comparison, boolean and common mathematical functions. Property values can also depend on the value of other elements. Several tree traversal functions are available that return immediate neighbors, ancestors, the generating node of an elaborated element and the current element.

The order of evaluation for PSL style rules is:

1. Element-specific rules
2. DEFAULT section rules
3. Inherited values
4. Medium specified values (initial values in CSS jargon)

The following piece of PSL code for achieving a layout with N list items where each item follows its predecessor, except for item $(N/2 + 1)$, which is treated as an exception and is placed at the top of the second column:

```
li {
  if (ChildNum(Self) == round(NumChildren(Parent) / 2 + 1)) then
    VertPos: Top = Parent.Top;
    HorizPos: Left = LeftSib.Left + LeftSib.Width;
  else
    VertPos: Top = LeftSib.Actual Bottom;
    HorizPos: Left = LeftSib.Left;
  endif
  Width = 200;
}
```

5.3.5 Conclusion

CSS has a simple syntax, simple semantics but limited expressive power. It is very intuitive to express style in the form of style rules. However, this intuitiveness comes at the cost of consistency. The main problem with CSS is that it originally has been design to style documents and not user interfaces.

The constraint style language, CCSS, as described in section 5.3.3 is an interesting approach. It offers a nice and intuitive way to express style rules. One downside of the approach is its need for a constraint solver in the web browser. Although great in theory, it will take a lot of effort to convince the world to implement and support such a feature. Perhaps a better approach is to translate the constraints into ordinary CSS rules.

Compared to CSS, PSL has a greater expressive power at the expense of slightly more complex syntax. For example, PSL stylesheets consist of multiple sections for default, normal and elaboration rules. CSS only has one section for normal rules. CSS only allows constant values as property values, whereas PSL also allows expressions. Furthermore, PSL has consistent semantics that are uniformly applied throughout the language. For instance, CSS has different semantics for the percentage value of the `line-height` and `font-size` properties. Finally, PSL is more suitable for specifying layouts, which are still hard to express in CSS.

However, no matter how good the alternatives to CSS like PSL and CCSS are, they suffer from one major flaw: browser support. CSS is currently the only styling language which enjoys support from all major browsers and will probably remain so for the next decade. Even implementing CSS uniformly across all browsers is a great ordeal. One may assume PSL and CCSS will not receive browser support in the near future and are therefore no serious candidates to base the implementation of the layout and styling extensions upon.

Thus, CSS is the language of choice used in the implementation of the layout and styling extensions, which will be described in detail in the rest of this chapter.

5.4 Layout Syntax and Semantics

The syntax for the Layout extension can be seen in figure 5.1. For a better understanding, the syntax is described using a pseudo syntax. Syntax rules with a `w-` prefix indicate which parts of the WebDSL syntax are reused.

5.4.1 Layout Section

Layout is described in a layout section, which is a WebDSL section. This is where the layout extension is integrated into WebDSL. A layout section can therefore appear in any module just as other WebDSL sections. A layout section consists of multiple layout definitions. A concrete example of the usage of the layout syntax can be seen in figure 5.2, where a new module called *main-layout* is defined. The keyword `layout` indicates the start of a layout section containing multiple layout definitions.

```

w-section -> "layout" layout-definition*

layout-definition -> selector "{" layout-statement* "}"

layout-statement -> "float" "{" match-definition* "}" ";" | layout-expression ";"

layout-expression -> "#" w-id ":" "[" {layout-expression "|"}+ "]" | w-id |
                    match-definition

```

Figure 5.1: Pseudo syntax for the Layout Extension

```

module main-layout

layout

  template main() {
    top();
    topmenu();
    mainbody: [ sidebar() | body() ];
    footer();
  }

  template top() {
    #logo_area: [ logos | text ];
  }

  template footer() {
    #footer_area: [ footer_links | footer_text ];
  }

  page allOwner() {
    float { ownerDetails(o : Owner) };
  }

```

Figure 5.2: A concrete example of the usage of the layout syntax

5.4.2 Layout Definitions and Selectors

A layout definition specifies a selector and multiple layout statements. The selector indicates which WebDSL elements the layout statements affect. For instance, templates or pages can be selected, as well as a listitem inside a list inside a specific template. The idea of selectors is borrowed from the CSS specification [7]. The pseudo syntax for selectors can be seen in figure 5.3.

A selector consists of simple selectors. A simple selector can select WebDSL page elements, templates and pages, for example: `page home() {...}` or `template main() {...}`.


```

selector -> simple-selector (combinator simple-selector selector-attribute?)*

simple-selector -> w-id match-definition | match-definition |
                "#" w-id | "." w-id

selector-attribute -> "." w-id

combinator -> ">" | ">>"

match-definition -> match-ident "(" match-definition-args ")"

match-definition-args -> {formal-arg ","}*

```

Figure 5.3: Pseudo syntax for selectors

Furthermore, a simple selector can be a match-definition, which simply matches WebDSL template definitions by name and the types of the arguments. Finally, a simple selector can match a WebDSL element that has been given an id by using `#footer_area {...}` or an element that belongs to a particular class by `.block {...}`.

Selectors can be made more specific, for instance, by combining simple selectors that select a template only when it occurs in a specific page: `page home() >> main()`, instead of the more general: `main()` which selects all occurrences of the template call *main*. The combinators that can be used are the child and the descendant combinator, indicated by `>` and `>>` respectively. A child combinator indicates that only direct children are selected, while a descendant combinator also selects the children's children, grandchildren and so on. Furthermore, selectors can have attributes, which also is a concept borrowed from CSS. An attribute allows even more specific selection of elements. For instance, the first child of an element is selected as follows: `list() > listitem().first-child`. Examples of selectors are given in figure 5.4.

```

page home() { .. }
template main() { ... }
page home() >> main() { ... }
page home() >> main() >> list() > listitem() { ... }

```

Figure 5.4: Examples of selectors

The first example simple selects the page named *home*. The second selects all templates named *main*. The third example selects template named *main* that are called from a page named *home*. The final example matches *listitem* elements that are direct children of the *list* element which can occur anywhere inside the definition of the *main* template called from the page named *home*.

5.4.3 Layout Statements and Expressions

Layout specifies where elements must be placed on a page and how they are placed with respect to each other. This is exactly what can be specified by layout statements. A layout statement consists of either a layout expression or a float expression. A float expression is composed of multiple match-definitions, which, as already explained above, represent pages or templates. A layout expression is either an WebDSL id, a match-definition or one or more layout expressions separated by a pipe and named by a WebDSL id.

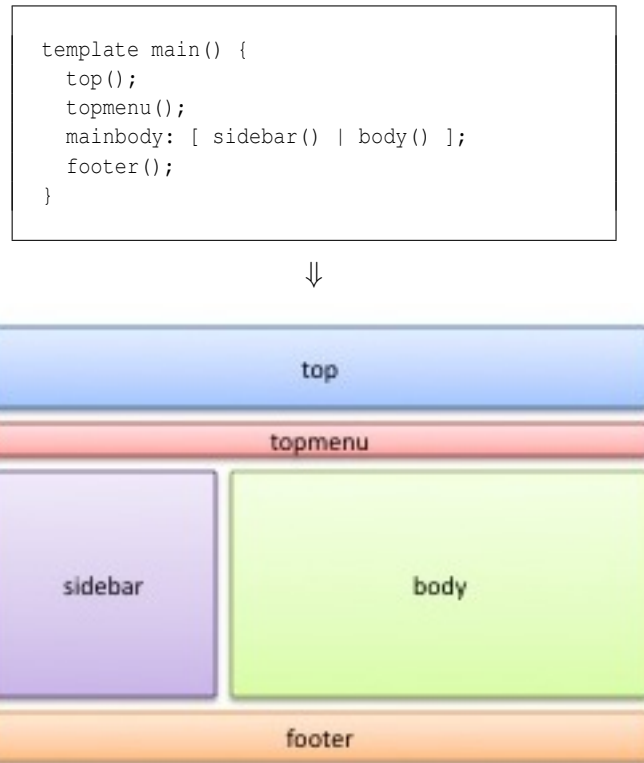


Figure 5.5: Float statement in action

First, a float expression states that all templates specified by the expression must float alongside each other. Figure 5.5 shows an example of the usage of the float expression along with an example of how it could look like. In the example, a page is defined named *allOwner* which, as its name suggests, gives an overview of all *Owner* entities in the database. To accomplish this, the page iterates over all *Owner* entities and for each *Owner* calls the template *ownerDetails* giving the owner being iterated over as an argument. The implementation of the template *ownerDetails* is irrelevant at this point. As can be seen in the

right code fragment of the example, the page *allOwner* is selected and the template call to *ownerDetails* is placed in a float expression, resulting in the bottom figure of the example. A nice feature of using a float expression is that when the browser window is resized, the number of *ownerDetails* templates placed next to each other is adjusted. For example, when the browser window is made smaller, only two columns would be placed next to each other. The remaining ones will float underneath one another.

Next, a layout statement can consist of a layout expression. A layout expression can indicate a template or the class of a block statement. A layout expression followed by a semi-colon corresponds to a horizontal arrangement. A number of layout expressions followed by a semi-colon correspond to a vertical arrangement. This is graphically illustrated in figure 5.6. In the example above, setting a width on the sidebar results in the sidebar being fixed, while the body scales according to the browser size.



5.5 Style Syntax and Semantics

The syntax for the styling extension of WebDSL can be seen in figure 5.7. Just as with the layout extension, the syntax is described using a pseudo-syntax.

5.5.1 Style Section

Again, a WebDSL section is used as the place to insert styling rules and to link it to the rest of the WebDSL syntax. A styling section has a unique name, which is used as a name for the generated CSS stylesheet. Section 5.6 will discuss how the generated CSS is linked to the generated HTML and JSF code. Figure 5.8 gives an example of the usage of the style syntax.

5.5.2 Style Definitions

A style section consists of a number of style definitions. A style definition is simply a selector and multiple style statements. A selector consists of a simple selector, followed by zero

```

w-section -> "style" w-id style-definition*

style-definition -> selector "{" style-statement* "}" | style-var-decl

style-statement -> style-var-decl | style-var-decl-init | style-declaration

style-var-decl -> "const" w-id ":" style-sort ";"

style-var-decl-init -> "const" w-id ":" style-sort ":" style-expression ";"

style-declaration -> style-property ":" style-expression ";"

style-expression -> style-property-value | style-value-expression | style-value |
    style-add | style-sub | style-mul | style-div

style-property-value -> match-definition "." style-property

style-value-expression -> "(" selector ")" "." style-property

style-add -> style-expression "+" style-expression
style-sub -> style-expression "-" style-expression
style-mul -> style-expression "*" style-expression
style-div -> style-expression "/" style-expression

style-property -> w-id

```

Figure 5.7: Pseudo syntax for the Style Extension

or more simple selectors, separated by a combinator and optionally accompanied by a selector attribute. Selectors have already been discussed in section 5.4.2. The example above shows five style statements. A style section may also contain style variable declarations, although they also can occur in style definitions. A style variable is defined as the keyword `const` followed by a name and a style type:

```
const mainColor : Color;
```

The variable can then be used throughout the section or definition in which it is declared. This increases reuse and style consistency throughout a web application. Style variable can also be declared having an initial value:

```
const mainColor : Color := Color.white;
```

Style definitions contain style statements. A style statement is either a variable declaration, initialized or uninitialized, or a style declaration. Style variable declarations have already been discussed. Style properties are set using a style declaration. The right-hand side of a style declaration is a style expression, which is covered below.

```
module application-style

style globalStyle

  const mainColor : Color := #330099;
  const globalFont : Font := Font.Lucida.Grande;
  const globalFontSize : Length := 0.75em;
  const layoutBorderColor : Color := #c0c0c0;

  section() {
    font := globalFont;
    font-size := globalFontSize;
  }

  template output(c : List<Specialty>) >> list() {
    orientation := Orientation.horizontal;
    separator := Separator.comma;
    spacing-right := 0.2em;
  }

  page allOwner() >> ownerDetails(o : Owner) {
    spacing-right := 1em;
  }

  page allPet() >> list() {
    orientation := Orientation.vertical;
    separator := Separator.none;
  }

  template footer() {
    width := top().width;
    margin-top := topmenu().margin-bottom;
  }
```

Figure 5.8: A concrete example of the usage of the style syntax

5.5.3 Style Expressions

The simplest style expression is a style value. Examples of style values are: `22em`, `100px`, `Color.white`, `#ff00ff`, etc. Style values are discussed in more detail below. A style expression can also point to a style expression of another style definition. There are two ways to reuse a style expression from another definition in the syntax definition. First, the right-hand side of a style declaration can be specified as a match definition (the name and argument types of a template definition) followed by a dot and the name of the property whose value is to be used. An example is:

```
margin-top := topmenu().margin-bottom;
```

However, a more specific selector can be used as well. In that case, the selector has to be put between braces:

```
margin-top := (page main() >> topmenu()).margin-bottom;
```

By allowing the value of a style property to depend on the values of other properties, relational style can be specified. A great advantage of capturing style relations is that when one value of a relation is changed, the other values are changed automatically. This is illustrated by the following example, where the header and footer will always have the same height and width and this only has to be changed in one place:

```
template header() {
  width := 800px;
  height := 100px;
}

template footer() {
  width := header().width;
  height := header().height;
}
```

Naturally, style expressions can be represented by style variables, as long as the variables are declared in the same definition or in the same section. An example, that already could be seen in figure 5.8, is: `font := globalFont;`

Furthermore, style expressions can be added, subtracted, multiplied and divided. This is particularly useful when a style value is a result of values of other definitions. For instance, the width of an element can be specified as the sum of the width of its children, e.g.

```
width := (child1() >> label()).width + (child2() >> input()).width
```

An application of this principle is shown in figure 5.9, where a page can be seen where a new owner can be created by filling some fields in a form. The WebDSL page structure, the styling rules and the result can be seen in the example. For brevity, not all styling code is shown. The interesting part here are the buttons at the bottom of the form, which are in this case right-aligned. Normally, the width of the form would be as wide as the viewport of the browser. The width of the group elements inside the form is as wide as the elements contained in the group. However, because a different transformation to HTML of the group with groupitems and the group where the actions are in, the width of the group of the actions is as large as the width of its parent, which in this case is the form. The result is that the buttons are right-aligned as far as the form is wide, which is far wider than the group elements containing the *Owner details* and *More details*. This can be fixed by setting a fixed width on the form. However, this is undesirable since the nice feature of the group elements scaling according to its content would be lost. The solution is to calculate the width by adding the width of all elements, margins, paddings and the width of borders and setting that width to the form. The result can be seen in figure 5.9. This is a very nice abstraction over normal CSS, where manually the width has to be calculated every time one of the parts of the composition changes.

```

define page createOwner() {
  var o : Owner;
  header{ "Create new owner" }
  form {
    group("Owner details") {
      groupitem { label("Name:") { input(o.name) } }
      groupitem { label("Address:") { input(o.address) } } }
    group("More details") {
      groupitem { label("City:") { input(o.city) } }
      groupitem { label("Telephone:") { input(o.telephone) } }
      groupitem { label("Pets:") { input(o.pets) } } }
    group {
      action("Save", save())
      action("Cancel", cancel()) } }

  action save() {
    o.save();
    return owner(o); }
  action cancel() {
    cancel home(); } }

```

```

style formStyle

form() {
  width :=
    (form() > group()).padding-left +
    (form() > group()).padding-right +
    (form() > group() >> label()).margin-right +
    (form() > group() >> label()).width +
    (form() > group() >> input()).width *
    (form() > group() >> input()).font-size +
    (form() > group() >> input()).border-right-width +
    (form() > group() >> input()).border-left-width;
}

```



Figure 5.9: Example of the addition of style expressions

5.5.4 Style Properties

In essence, the style extension works by setting style properties on WebDSL elements. WebDSL elements that are supported are pages, templates, input and output of entity properties and all basic page elements, such as `title`, `section`, `navigate`, etc. A list of properties that can be set can be found in appendix A. As already described, style properties are gathered inside style definitions, which target a specific WebDSL element specified by a selector.

5.5.5 Style Values

As stated before, style declarations set a value on a specific property. The pseudo syntax for a style value can be seen below:

```
style-expression -> style-value

style-value -> number | number unit | style-sort ( "." w-id )+ |
              "url" "(" string ")" | "#" hex | string
```

A style value can be a single number when a property has to be set to zero, for instance in:

```
margin := 0
```

Otherwise, a unit has to be specified:

```
width := 100%
border-width := 1px
```

For a list of units, see appendix A. Sometimes a value can be one of a number of predetermined values. This is indicated by specifying the type of the property, followed by a dot and the value. Examples are:

```
orientation := Orientation.horizontal
border-style := BorderStyle.solid
separator := Separator.comma
font-style := Line.under
```

Other values that can be specified are urls, hexadecimal values for colors and strings:

```
url := url("images/company_logo.png")
background-color := #ff00ff
content := '|'
```


5.6 Implementation

WebDSL supports both vertical and horizontal modularization [24]. Vertical modularization, which already has been shown in figure 3.1, is used to shorten the distance between input and output models. The WebDSL generator is implemented as a series of model-to-model transformations that transform an input model written in the WebDSL language to lower level models geared towards implementation. Vertical modularity helps in separation of concerns and retargetability. On the other hand, horizontal modularization enables extensions to the base language implemented as plug-ins through the separate definition of model-to-model transformation steps.

The styling and layout extensions have been implemented by extending the separate transformation stages of the WebDSL generator pipeline. The styling extension itself uses vertical modularization to reduce the gap between the styling model and the generated CSS code. The various stages in the styling pipeline can be seen in figure 5.10.

```
process-style-stages =  
  reset-default-style  
  ; insert-default-style  
  ; resolve-style-properties  
  ; translate-style-properties  
  ; weave-hooks
```

Figure 5.10: Styling stages

First, the default style is reset. This is done in order to reduce browser inconsistencies. All browsers have style defaults, but they differ significantly between browser families. For example, some browsers indent unordered and ordered lists with left margins, whereas others use left padding. Headings have slightly different top and bottom margins, indentation distances are different, the default line height varies from one browser to another, affecting element heights, vertical alignments, and overall feel. Resetting browser defaults is implemented by setting properties like margin and padding to zero.

Next, elements are given a sensible default style. This ensures that even when no style has been specified, a WebDSL application still looks good. Style properties that do have a value specified, no default style is generated. This way, conflicting style rules will not occur. An example where a default style is essential is a form containing various labels and input fields, such as the form shown in figure 5.9. Without a standard style, the form would look messy.

In the next stage, style properties are translated to equivalent CSS properties. The result of this is that there is a very simple mapping from the style properties to actual CSS code. For example, the style declaration `spacing := 1em;` in combination with the property `orientation` set to horizontal for a list is translated to

```
margin-left := 0.5em;  
margin-right := 0.5em;
```

which causes list elements to be spaced evenly.

The following step resolves style values for properties that consist of values from other style definitions, style variables, or compound statements. Compound statements consist of multiple style expressions, such as in the addition of style values. Selectors which consist of WebDSL page elements such as `section`, `header` and `text` are transformed into their equivalent HTML element in this phase as well. For example, the WebDSL `list` element is transformed into an element selector named `ul`. In the same way, a `listitem` element is transformed into `li`. Finally, layout rules have to be attached to the generated HTML code somehow. The layout extension requires quite a few of these HTML hooks in order to work properly. These HTML hooks are obtained by weaving WebDSL *block* elements in the page and template definitions. This is done in the last stage.

In the code generation stage, which is an extension of the WebDSL code generation stage, the target languages for which code has to be generated are HTML and CSS. Support for generating HTML was already available in the form of an XML front for Stratego [8]. No such thing existed for CSS, so that had to be developed. This is done by constructing a grammar for CSS using SDF and a pretty-printer that produces CSS.

In the generation process, multiple style sheets are generated. First of all, reset and default style rules are generated as a separate CSS style sheet. Layout definitions are also generated separately. Furthermore, each style section is also generated as a CSS style sheet. All these separate style sheets need to be included in the right HTML pages. In order to accomplish this for style sections, the selectors of the style definitions are examined. When a particular page or template occurs in a selector of some style definition, the style section in which the definition is defined is coupled to that page or template. This way, when generating pages it is known which stylesheets to include. When a style section is coupled to a template, it is checked which page calls that template and the style sheet is included. In addition, style sections are searched for general style definitions, for instance style definition which select all WebDSL `navigate` page elements. Style sheets generated from such style sections are included in all pages.

Style sheets generated for layout definitions are also coupled to templates. Pages calling thoses templates include the layout style sheets.

5.7 Conclusion

This chapter covered the details of layout and style extensions for WebDSL, which allow the definition of style and layout rules in order to specify the appearance of web applications. Especially specifying the layout of web applications is cumbersome, time-intensive and error-prone. This chapter has shown high-level abstractions for specifying a page's layout with a very simple syntax, which aims to overcome those problems. Furthermore, the ability to specify style rules that depend on each other and the capturing of recurring style values in variables is a powerful concept that contributes to a reduction of the development effort.

The design of the style extension is heavily influenced by CSS, since a lot of concepts of CSS are really useful. CSS is too low-level, however, and a lot of the problems with CSS as described in Section 5.3.1 can be overcome using the approach as described in this thesis,

with the exception of problems related to the implementation of CSS. Here, for each of the problems of Section 5.3.1 will be explained how the approach taken in this thesis can solve that problem.

Missing functionality: Some missing functionality can be implemented in the styling extension. The biggest problem with CSS is that it was originally designed for styling documents, not for complex user interfaces. This has resulted in poor support for specifying layout and things as headers and footers. These concepts can easily be integrated into the language extensions. Other missing functionality in CSS such as specifying text size in terms of the containing elements width can be added as well, since this can easily be calculated by the WebDSL generator. Even something like specifying a certain contrast between the text color and the background could be built into the styling language.

Excessive functionality: The problem of excessive functionality is easily solved by not allowing these properties in the language extension. An example that was given in section 5.3.1 where inline elements do not really need padding, border and margin areas is easily solved by checking in the typechecker whether illegal properties are set on inline elements and giving warning or error messages accordingly.

Poor design: It was said earlier that while missing and excessive functionality is relatively easy to fix by respectively adding and removing functionality, poor design decisions are more difficult to overcome. Examples are the `white-space` and `text-decoration` properties that both describe multiple unrelated concerns. These separate concerns should be extracted and it should be possible to specify them individually. This is easily solved in the language extension as well, by separating these concerns and defining them in separate properties. This has already been used in the implementation of the styling extension. Another issue named earlier was positioning. Layout information is difficult to specify, and the positioning properties are not well suited for cascading. This already has been solved by the development of a separate language for describing layout information.

Implementation problems: The implementation problems of web browsers that do not implement the CSS language according to the specification is of course not solvable by using this approach. However, the burden of having to tweak the style and layout specification for every browser can be minimized by encapsulating browser differences in the style generator. The downside is that this adds extra complexity to the generator, although this effort has to be done only once.

There certainly is room for improving the current way of specifying layout and style using CSS. The missing functionality can easily be filled by using a DSL with good abstractions for the missing concepts. Even if at one point in time CSS gets support for these missing functionality, no changes to the language extensions have to be made. It only makes transformations from DSL code to CSS less complex.

The language extensions offer a mechanism for the encapsulation of browser differences. This further reduces development efforts because the effort of creating a consistent

appearance across different browsers and different platforms only has to be made once for the generator instead of for every web application. The ability to reset browser defaults and to specify default style rules contributes to this as well.

In conclusion, the foundations for building more style and layout abstractions have been laid, and the initial results are promising. Chapter 6 will demonstrate the use of the style and layout extensions on three case studies. It will demonstrate the effectiveness of the extensions compared to manually specifying the appearance of web applications.

Chapter 6

Evaluation

This chapter will evaluate the language and styling extensions to WebDSL as presented in this thesis by providing three case studies. The data of the first two case studies have been kindly provided by Atos Origin from two different projects that are currently being performed by Atos Origin. The last case study, the PetClinic, is a widely used example within Atos Origin for the demonstration of new techniques. Needless to say, this thesis will not deviate from that custom. Emphasis will lie on the styling and layout aspects of the projects, and not on their actual function. The most interesting aspects related to the user interface will be highlighted.

This chapter is composed as follows. First, an overview of the questions these case studies try to answer is provided in Section 6.1. Then the three case studies are covered in Section 6.2, Section 6.3 and Section 6.4 respectively. Finally the questions as provided in Section 6.1 are answered in Sections 6.5.

6.1 Questions

This chapter aims to evaluate the effectiveness of the language extensions to WebDSL for describing layout and style. Furthermore, it aims to find evidence to support the claims of better reusability, maintainability and productivity as stated in Section 1.2. The chapter tries to do this by stating a number of questions that at the end of this chapter will be answered. The questions that need to be answered are:

- Q1:** How well suited is the solution provided by this thesis for the description of the user interface of the case studies?
- Q2:** How well do the abstractions incorporated in the styling and layout extensions work for the case studies?
- Q3:** What is the proportion of the amount of generated code in relation to the amount of defined styling and layout code?
- Q4:** Where can the solution described in this thesis be improved? What cannot be expressed in the language?

6.2 Case Study 1: ING Card

The screenshot shows the ING Card website interface. On the left is a vertical sidebar with an orange header containing the ING CARD logo. Below the logo, the sidebar lists navigation links: Home, Create Card Request, Work Card Request, Scoring, Maintenance, Digital Applications, Pending Requests, and Letters. The main content area on the right has a light orange background. It features two search sections: 'Search Card Request' with input fields for Postcode and Achternaam, and a 'Search Card Request' button; and 'Search Customer' with input fields for Kaartnummer, Achternaam, and Bankrekeningnummer, and a 'Search Customer' button.

Figure 6.1: Impression of the ING card case study

6.2.1 Layout

The ING Card case has a rather straightforward layout. It consists of a header, sidebar and body. The skeleton of the layout can be seen in Figure 6.2.

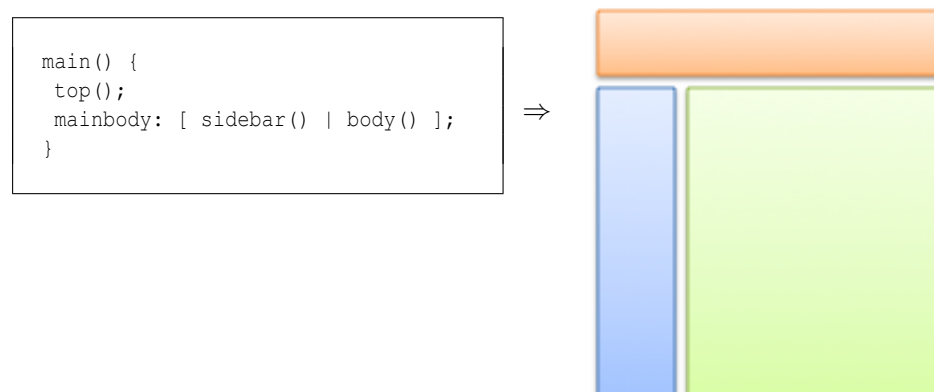


Figure 6.2: Layout skeleton of the ING Card case

6.2.2 Style

In this section, the most interesting style aspects of the ING Card case study will be highlighted. For each aspect, the defined WebDSL code is shown followed by an image of what it would look. Also, the styling code and possible intermediate stages of WebDSL code as a result of weaving operations is shown. An example of the latter point can be seen in Figure 6.5 where as a result of a model-to-model transformation a class attribute has been set on the group element.

Form

Figure 6.4 shows two forms with a number of input fields and a button. Only code of the bottom form will be shown, as the forms are almost identical.

```
form() {
  var customerByLastname : String;
  group("Search Customer") {
    groupitem { label("Kaartnummer") { input(customer.achternaam) } }
    groupitem { label("Achternaam") { input(customerByLastname) } }
    groupitem { label("Bankrekeningnummer") { input(customer.achternaam) } } }
  group() {
    action("Search Customer", search_customer()) }

  action search_customer() {
    return customers(customerByLastname); } }
```

Figure 6.3: WebDSL code of the form

Search Card Request

Postcode

Achternaam

Search Customer

Kaartnummer

Achternaam

Bankrekeningnummer

Figure 6.4: A form in the ING card case

An example of a form was already shown in Figure 5.9. The code in that example is generated automatically for each form, so it is applicable to this form as well. That means

the width is automatically calculated and set. This allows for the right aligning of the action button, as can be seen in the styling code in Figure 6.6.

Often, the group element containing the action has to be styled differently from the group element containing the form input fields. For this to be possible, the group element containing the action should be able to be specified by a selector somehow. This is solved by setting the class attribute of the group element. This is done in a transformation stage in the styling pipeline, as discussed in Section 5.6. The result can be seen in Figure 6.5.

```
group([class := fieldset_action_]) {
  action("Search Customer", search_customer())
}
```

Figure 6.5: A form in the ING card case

```
form() {
}

.fieldset_action_ >> action() {
  align := Align.right;
  margin-right := (form() >> group()).margin-right;
}
```

Figure 6.6: A form in the ING card case

Sidebar menu

Figure 6.7 shows the menu in the sidebar. The style code can be seen in Figure 6.9.

The sidebar is simply a WebDSL list, as can be seen in Figure 6.8, where each menu option is represented by a listitem. By setting

```
orientation := Orientation.vertical
```

the listitems are placed in a vertical arrangement. No separator is needed, so

```
separator := Separator.none
```

is set. The rest of the style properties are pretty much self explanatory.

Table

The home page allows for search customers by their last name, resulting in a list of customers matching the search keyword. The list of customers is provided as a table and can

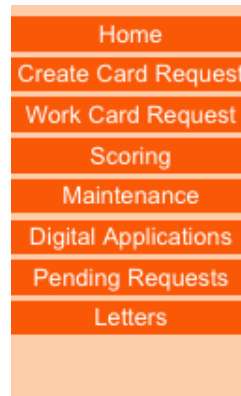


Figure 6.7: Sidebar menu

```
define sideMenu() {
  list {
    listitem { navigate(home()) { "Home" } }
    listitem { navigate(home()) { "Create Card Request" } }
    listitem { navigate(home()) { "Work Card Request" } }
    listitem { navigate(home()) { "Scoring" } }
    listitem { navigate(home()) { "Maintenance" } }
    listitem { navigate(home()) { "Digital Applications" } }
    listitem { navigate(home()) { "Pending Requests" } }
    listitem { navigate(home()) { "Letters" } }
  }
}
```

Figure 6.8: WebDSL code of the sidebar menu

be seen in Figure 6.10. The table requires no style rules at all. This is the default style as generated in WebDSL. The styling code that is generated automatically can be seen in Figure 6.11.

Customer page

When in the table of the search results a customer's name is clicked on, the Customer page is shown. The page can be seen in Figure 6.12.

Floating forms The top of the page shows an overview of the details of the customer. The details are composed of two forms that float next to each other. Figure 6.13 shows the WebDSL code. The layout code can be seen in Figure 6.14. The end result of the forms and the forms in a small browser viewport can be seen in Figure 6.15 and Figure 6.16 respectively. An illustration of the floating forms is shown in the figure as well, where the forms are split in two and placed beneath each other because of a smaller browser viewport.

```

sideMenu() > list() {
  orientation := Orientation.vertical;
  separator := Separator.none;
  align := Align.center;
  margin-top := 0.5em;
  spacing := 0.2em;
}

sideMenu() > list() > listitem() {
  background-color := #f95807;
}

sideMenu() > list() > listitem() > navigate() {
  font-color := Color.white;
  font-line := Line.none;
}

```

Figure 6.9: Styling code of the sidebar menu

Customers:

Name	Address	City	Telephone
Klant1	1953-04-23	161341341	klant1@bedrijf1.nl
Klant2	1966-08-01	161235135	klant2@bedrijf2.nl

Figure 6.10: A form in the ING card case

Tabbed list Another interesting aspect that can be seen in Figure 6.12 is the tab bar beneath the *customerDetails* forms. The WebDSL code for the task bar is nothing more than a list of navigate elements. The code of the list can be seen in Figure 6.13. For brevity, not all listitems are shown. The styling code that changes the list in a tab bar can be seen in Figure 6.17. The only piece of styling code that is needed for the list to appear as a tabbed list is the definition of property `orientation` set to `Orientation.horizontal` and property `separator` set to `Separator.tab`.

```

table() {
  border-spacing := 0;
  border-collapse := collapse;
}

table() > header() {
  border-width := 1px;
  border-style := BorderStyle.solid;
  border-color := #c0c0c0;
  padding-top := 0.2em;
  padding-right := 2em;
  padding-bottom := 0.2em;
  padding-left := 0.6em;
  text-align := left;
}

table() > row() {
  padding-top := 0.3em;
  padding-top := 0.3em;
  padding-right := 2em;
  padding-bottom := 0.3em;
  padding-left := 0.6em;
  vertical-align := top;
  border-bottom-width := 1px;
  border-bottom-style := BorderStyle.solid;
  border-bottom-color := #c0c0c0;
}

```

Figure 6.11: A form in the ING card case

ING CARD

Home
Create Card Request
Work Card Request
Scoring
Maintenance
Digital Applications
Pending Requests
Letters

Voorletters	K. P.	Sofinummer	161341341
Achternaam	Klant1	Emailadres	klant1@bedrijf1.nl
Geboortedatum	23/04/1953	Geslacht	M

Customer Card Campaign Maintenance Transaction Statement Internet Account Contact History Letters

Card data

Naam op kaart	K P Klant1
Rekeningnummer	82935923
Bestedingslimiet	1000
Rente percentage	6
Vervaldatum	04/08/2010

Figure 6.12: The customer page in the ING card case

```

define customerDetails(c : Customer) {
  group {
    groupitem { label("Voorletters") { output(c.voorletters) } }
    groupitem { label("Achternaam") { output(c.achternaam) } }
    groupitem { label("Geboortedatum") { output(c.geboortedatum) } } }
  group {
    groupitem { label("Sofinummer") { output(c.sofinummer) } }
    groupitem { label("Emailadres") { output(c.emailadres) } }
    groupitem { label("Geslacht") { output(c.geslacht) } } }

  list {
    listitem { navigate(customer(c)) { "Customer" } }
    listitem { navigate(card(c)) { "Card" } }
    .
    .
    .
  } }

```

Figure 6.13: customerDetails template

```

customerDetails(c : Customer) {
  float { group() };
  list();
}

```

Figure 6.14: Layout for customerDetails template

Voorletters	K. P.	Sofinummer	161341341
Achternaam	Klant1	Emailadres	klant1@bedrijf1.nl
Geboortedatum	23/04/1953	Geslacht	M

Figure 6.15: customerDetails forms

Voorletters	K. P.
Achternaam	Klant1
Geboortedatum	23/04/1953

Sofinummer	161341341
Emailadres	klant1@bedrijf1.nl
Geslacht	M

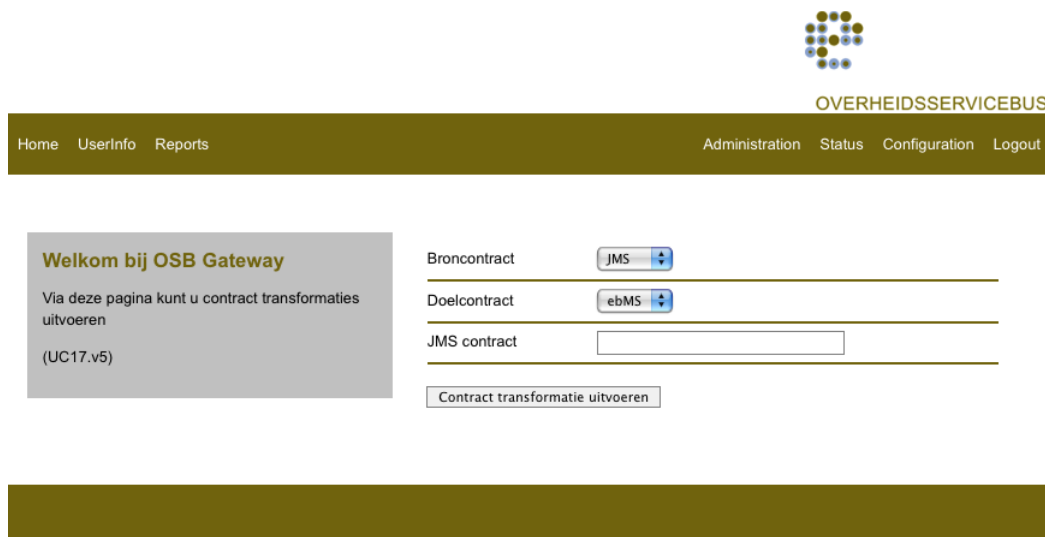
Figure 6.16: customerDetails forms with small browser viewport

```
customerDetails(c : Customer) > list() {  
  orientation := Orientation.horizontal;  
  separator := Separator.tab;  
  spacing := 1em;  
  margin-top := 1em;  
  margin-bottom := 1em;  
}  
  
customerDetails(c : Customer) > list() >> navigate() {  
  font-line := Line.none;  
}
```

Figure 6.17: Styling code for the tab bar

6.3 Case Study 2: OSB

The OSB, or Overheid Service Bus, case study will be described briefly here. The case study offers no new concepts of interest to styling and layout. It consists of a main layout and of a lot of forms. A description of the layout and an example of a form are provided.



The screenshot shows the OSB Gateway web application. At the top right is a logo consisting of a grid of colored dots. Below the logo is the text "OVERHEIDSSERVICEBUS". A navigation bar contains links: "Home", "UserInfo", "Reports", "Administration", "Status", "Configuration", and "Logout". The main content area is divided into two sections. On the left, a grey box contains the text "Welkom bij OSB Gateway", "Via deze pagina kunt u contract transformaties uitvoeren", and "(UC17.v5)". On the right, there is a form with three rows: "Broncontract" with a dropdown menu showing "JMS", "Doelcontract" with a dropdown menu showing "ebMS", and "JMS contract" with a text input field. Below these fields is a button labeled "Contract transformatie uitvoeren".

Figure 6.18: OSB case study

6.3.1 Layout

As can be seen from Figure 6.19, the layout consists of two headers, one for the logo and one for the main navigation, a sidebar and body and a footer. The code for the layout can be seen in Figure 6.20. The code also shows the definition of template *topmenu*, which consists of two navigation templates. The two navigation templates are placed together in a horizontal arrangement. The navigation templates themselves simply consist of a list of navigate elements, as we have already seen before.

6.3.2 Style

The interesting styling aspects of the OSB case study are the navigation templates in the *topmenu* template and an example of a form. These will be shown in the following sections.

Topmenu navigation templates As already explained in the layout section above, the *topmenu* template consists of two navigation templates. The styling code for the navigation templates can be seen in Figure 6.21. The idea is to keep the appearance of the two navigation templates the same. Therefore, the style of the *adminNavigation* templates refers to the properties of the *mainNavigation* template. The two templates can be seen in greater detail



Figure 6.19: Layout of the OSB case study

```

main() {
  top();
  topmenu();
  mainbody: [ sidebar() | body() ];
  footer();
}

topmenu() {
  menu: [ mainNavigation() | adminNavigation() ];
}

define topmenu() {
  mainNavigation()
  adminNavigation()
}

```

Figure 6.20: Layout code for the OSB case study

in Figure 6.22. The styling code for the alignment of the navigation templates can be seen in Figure 6.23.

Form The WebDSL code of the form as seen in Figure 6.24 has the same structure of the form in Figure 6.3 and will therefore not be shown again. Interesting aspects in the styling code of the form are the use of a style constant for the color of the borders in the form. This same constant is used in the definition of the background color of the *topmenu* and the *footer* templates. Furthermore, the default style aligns labels in a form on the right side. In this example, the labels are placed on the left by the last style definition in Figure 6.25. The default style for a group is that the top and bottom borders are set. In this example, this is undone by setting the style of the border to none.

```

mainNavigation() >> list() {
    orientation := Orientation.horizontal;
    separator := Separator.none;
    spacing := 1em;
}

adminNavigation() >> list() {
    orientation := (mainNavigation() >> list()).orientation;
    separator := (mainNavigation() >> list()).separator;
    spacing := (mainNavigation() >> list()).spacing;
}

```

Figure 6.21: Style code for the navigation templates



Figure 6.22: Two navigation templates

```

mainNavigation() {
    align := Align.left;
}

adminNavigation() {
    align := Align.right;
}

```

Figure 6.23: Style code for the navigation templates

Broncontract	JMS
Doelcontract	ebMS
JMS contract	<input type="text"/>
Contract transformatie uitvoeren	

Figure 6.24: A form in the OSB case


```
const houseColor : Color := #70630d;

form() > group() {
  border-style := BorderStyle.none;
  spacing := 0.2em;
}

form() > group() > groupitem() {
  border-bottom-style := BorderStyle.solid;
  border-bottom-width := 2px;
  border-bottom-color := houseColor;
  padding-top := 0.5em;
  padding-bottom := 0.5em;
  padding-right := 10em;
}

form() >> group() >> action() {
  align := Align.left;
}
```

Figure 6.25: Style code for the form in the OSB case

6.4 Case Study 3: Petclinic

The final case study is that of the PetClinic. The PetClinic is often used within Atos Origin for the demonstration of new techniques. It is an example web application for the registration of pet animals and their visits to the veterinarian. A screenshot of the PetClinic can be seen in Figure 6.26.

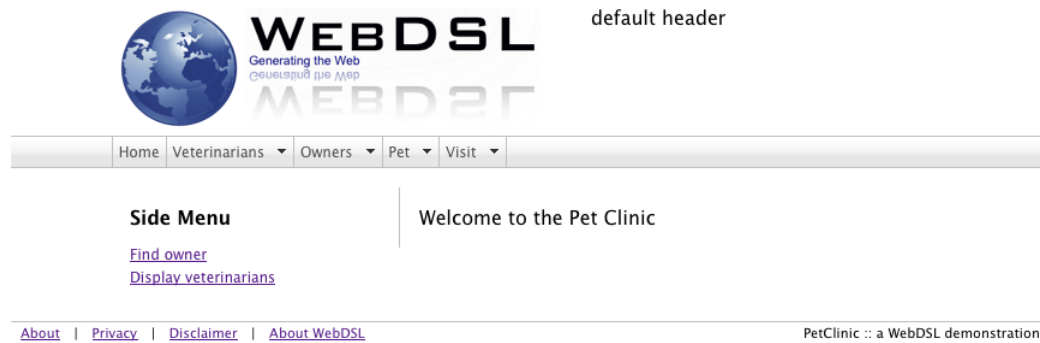


Figure 6.26: Petclinic case study

6.4.1 Layout

The layout of the PetClinic can be seen in Figure 6.27. It consists of a top, topmenu, sidebar, body and a footer. The *top* template houses the logo, while the *topmenu* template consists of the navigation menu. There is a fixed *sidebar* template with a couple of navigate elements. The *body* template is where the page specific content will reside. Finally, there is a footer with a list of navigate elements.

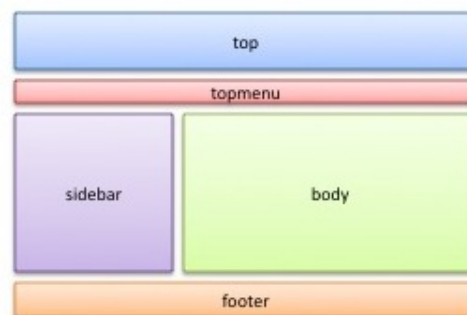


Figure 6.27: Layout of the Petclinic case study

6.4.2 Style

The interesting parts with regard to the styling of the PetClinic case study are the footer and a table. These will be discussed next.

Footer The left part of the footer consists of a list of navigate elements, as can be seen in Figure 6.28. The styling code used to accomplish this can be seen in Figure 6.29. The style property `Orientation` is again set to `Orientation.horizontal` and in this case the separator is a pipe.

[About](#) | [Privacy](#) | [Disclaimer](#) | [About WebDSL](#)

Figure 6.28: List of navigate elements in the footer

Table We have already seen an example of a table in Section 6.2.2. However, in this example there is a list inside a table. The table in Figure 6.30 shows a list of veterinarians. However, each veterinarian has a number of specialties. These specialties are presented in the same table as the veterinarians themselves. This time, the list is presented as a comma separated list. The styling code used to accomplish this can be seen in Figure 6.31. The WebDSL code for the table and the list can be seen in Figure 6.32.

```

footer() >> list() {
  orientation := Orientation.horizontal;
  separator := Separator.pipe;
  spacing := 2em;
}

```

Figure 6.29: Style code for the navigation elements in the footer

Veterinarians:

Name	Specialty
James Carter	surgery, radiology
Linda Douglas	
Sharon Jenkins	surgery
Helen Leary	dentistry
Rafael Ortega	dentistry, surgery, radiology
Henry Stevens	surgery
Jantje	dentistry

Figure 6.30: Table in the Petclinic case study

```

output(c : List<Specialty>) >> list() {
  orientation := Orientation.horizontal;
  separator := Separator.comma;
  spacing-right := 0.2em;
}

```

Figure 6.31: Style code for the list inside a table

```
define page vets() {  
  main()  
  define body() {  
    header { "Veterinarians:" }  
    table() {  
      header { "Name" "Specialty" }  
      for (v : Vet) {  
        row { text(v.name) output(v.specialtiesList) } }  
    } } }  
  
define output(coll : List<Specialty>) {  
  list() {  
    for (s : Specialty in coll) {  
      listitem { output(s.name) } } } }
```

Figure 6.32: WebDSL code for the list inside a table

6.5 Conclusion

In Section 6.1 of this chapter, a number of questions were formulated that could be answered by performing the case studies described in this chapter. In this section, an answer is provided for each of the questions.

- Q1:** With regard to the question as how well suited the solution provided by this thesis for the description of the user interface of the case studies, this chapter has shown that the styling and layout of all the cases can be expressed. The selector mechanism is an excellent way of linking styling and layout code to the structure of a page or template.
- Q2:** A number of very useful abstractions have been made: specifying layout has become much, much easier compared to specifying layout in CSS; lists can be easily styled in many different ways, including tabbed, horizontally separated by a number of different separators, vertically and inline in tables; style values can easily be reused; style definitions can be made dependent on other style definitions, increasing consistency.
- Q3:** Table 6.1 provides an overview of the generated lines of CSS code (LOC), the lines of code of styling source code and the ratio of the generated code to the source code. These numbers do not include the lines of HTML code that have been generated in order to link the CSS code to the HTML structure. Although the ratios do not seem very high, it still proves the method described in this thesis is an improvement over CSS. Combined with the advantages as described in Section 1.2, it offers a solid basis upon which more abstractions can be build to even further contribute to the goals stated in Section 1.2.
- Q4:** Since the layout and styling extensions to WebDSL generate CSS code, any CSS concept can be incorporated into the extensions. This ensures that everything that can be expressed using CSS can be expressed in the language extensions as well. This would not bring any increase in productivity, but would still offer the advantages of having the style and layout definitions typechecked and being able to specify style and layout on the level of the rest of the WebDSL code. However, a number of things could be added to the language: an easier way to specify margins and paddings in combination with layout; have more default style; have more and better encapsulation of browser differences; and a number of eyecandy, such as rounded corners and more gradient backgrounds.

	ING Card	OSB	PetClinic
LOC generated CSS	376	387	562
LOC styling and layout source code	134	173	252
Ratio generated CSS to source code	2.8	2.2	2.2

Table 6.1: Quantitative analysis of the case studies

Chapter 7

Conclusions and Future Work

This chapter gives an overview of this thesis's results. The chapter begins by restating the main research questions from the introduction. Then, for each of the research questions an explanation will be provided as how this thesis aims to solve that question. Lastly, an overview of future work is presented.

7.1 Conclusions

For the sake of completeness, the research questions as provided in chapter 1 will be given again here:

- Q1:** How can the level of abstraction for describing and styling web-based user interfaces be raised in order to reduce development efforts while improving maintainability, reusability and quality?
- Q2:** Using WebDSL as a case study, what are good abstractions for describing the user interface of web applications and how can these abstraction be generalized for other environments?
- Q3:** How much of the user interface for web applications can eventually be generated using WebDSL on real-life case studies from the business domain.

The first research question is answered by using the concept of separation of concerns to address the various web user interface concerns as described in chapter 2 separately. This thesis has identified the various concerns in order to cope with the complexity of specifying web user interfaces, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability.

In order to answer the second research question, WebDSL has been adapted to facilitate the separation of presentation from structure and content. Furthermore, language extensions to WebDSL for the specification of layout and style have been developed. This makes it possible to describe a web application's style and layout separately in a declarative way. Furthermore, a number of issues with the current way of styling web applications have been addressed. By abstracting from the underlying CSS, it is possible to encapsulate

browser differences and introduce more expressive language constructs increasing developer productivity, uniformity of appearance, maintainability and reusability. Furthermore, the approach taken enables static verification of style and layout specifications. By building a sensible default style into the language extensions, it is possible to have a great looking web application with a small amount effort.

The last research question has been answered by performing three case studies, from which two of them came from the business domain. The case studies showed that it is possible to express all style and layout aspects of the cases in the language extensions as described in this thesis. Although the ratio of lines of generated CSS code to lines of source code proved not very high, it still is a significant improvement over plain CSS. A lot of the problems CSS has, have been solved, as already described in Section 5.7. Furthermore, the case studies showed a number of useful abstractions over plain CSS. This thesis has laid the foundations for further work in abstracting from CSS.

7.2 Future work

A number of enhancements have been identified that would contribute even more to the advantages of the approach as described in this thesis.

First, the development process for the specification of the appearance of a web application is quite tedious. Style and layout is specified by means of a separate DSL as an extension to WebDSL. The code is then transformed by the WebDSL generator into target code. The code is compiled by the compiler of the target language into an executable application. The application is deployed to an application server. The application server starts the application and the result can be seen. Further changes to the appearance of a web application cause this process to reiterate. Although some people don't mind having to specify style and layout information textually, it would be interesting to offer a graphical interface that supports the WYSIWYG principle (What You See Is What You Get). Even better would be to offer the possibility to style the web application 'live'. By 'live' is meant that WebDSL style and layout can be specified from within a web browser, after which the code is generated to CSS and the newly generated CSS is included in the web application.

Related to 'live' styling is personalization where the user of a web application can choose among a number of predefined styles and layouts. These changes would immediately be reflected inside the browser. In combination with 'live' styling this could prove very powerful as a mechanism to provide an excellent user experience.

Another interesting addition to the language extensions could be to be able to specify style constraints. For instance, with style constraints it would be possible to specify that the width of a certain template can not be wider than the template above it using a syntax similar to:

```
constraint template top() {  
    width <= footer().width;  
}
```

This would allow even more dependable styles. The typechecker could check and warn for constraint violations.

By extension of the above addition, a style assistant could be added to the language which gives warnings and tips over the use of certain style properties within the code. By designing good usability practices into the assistant the developer could further be aided.

Finally, similar language extensions as for layout and style or abstractions such as the separation of presentation from structure could be build for the remaining aspects of a web user interface which were shown in figure 2.1, taking the concept of separation of concerns even further.

Bibliography

- [1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. Uiml: an appliance-independent xml user interface language. *Computer Networks*, 31(11-16):1695–1708, 1999.
- [2] Greg Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. In *User interface software and technology (UIST '99)*, Nov 1999.
- [3] Luciano Baresi and Sandro Morasca. Three empirical studies on estimating the design effort of web applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4):15, 2007.
- [4] Bob Baxley. Universal model of a user interface. In *Designing for User Experiences (DUX '03)*, Jun 2003.
- [5] A Berglund. Extensible stylesheet language (xsl) version 1.1. *W3C Recommendation*, <http://www.w3.org/TR/xsl11/>, December 2006.
- [6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – http/1.0, 1996.
- [7] B Bos, T Celik, I Hickson, and H Wium Lie. Css 2.1 specification. *W3C Candidate Recommendation*, <http://www.w3.org/TR/CSS21/>, Jul 1997.
- [8] Martin Bravenboer. Connecting xml processing and term rewriting with tree grammars. Master’s thesis, Utrecht University, November 2003.
- [9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.
- [10] T Browne, D Davila, S Rugaber, and K Stirewalt. Using declarative descriptions to model user interfaces with mastermind. *Formal Methods in Human-Computer Interaction*, Jan 1997.

- [11] Ed Burns and Roger Kitain. Javaserer faces specification, version 1.2 final draft, May 2006.
- [12] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.
- [13] S Ceri, P Fraternali, and A Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, Jan June 2000.
- [14] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Data-driven one-to-one web site generation for data-intensive applications. In *Very Large Databases (VLDB 1999)*, pages 615–626. Morgan Kaufmann, 1999.
- [15] J Clark. Xsl transformations (xslt) version 1.0. *W3C Recommendation*, <http://www.w3.org/TR/xslt>, November 1999.
- [16] J Clark and S DeRose. Xml path language (xpath) version 1.0. *W3C Recommendation*, <http://www.w3.org/TR/xpath>, November 1999.
- [17] P da Silva. User interface declarative models and development environments: A survey. In *Proceedings of DSV-IS2000*.
- [18] PP da Silva, T Griffiths, and NW Paton. Generating user interface code in a model based user interface development environment. In *Advanced Visual Interfaces (AVI 2000)*, pages 155–160, 2000.
- [19] Tony Griffiths, Norman W. Paton, Carole A. Goble, Adrian West, Peter J. Barclay, Jessie Kennedy, Michael Smyth, Jo McKirdy, Philip D. Gray, and Richard Cooper. Teallach: A model-based user interface development environment for object databases. In *User Interfaces to Data Intensive Systems (UIDIS '99)*, page 86, Washington, DC, USA, 1999. IEEE Computer Society.
- [20] Danny Groenewegen and Eelco Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In Daniel Schwabe and Francisco Curbera, editors, *International Conference on Web Engineering (ICWE 2008)*. IEEE CS Press, July 2008.
- [21] Michael Hanus. Putting declarative programming into the web: translating curry to javascript. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 155–166, New York, NY, USA, 2007. ACM.
- [22] Jan Hartmann, Alistair Sutcliffe, and Antonella Angeli. Investigating attractiveness in web user interfaces. In *Human factors in computing systems (CHI '07)*, Apr 2007.
- [23] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75, 1989.

BIBLIOGRAPHY

- [24] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *International Conference on Model Transformation (ICMT08)*, Lecture Notes in Computer Science. Springer, June 2008.
- [25] Z. Hemel, R. Verhaaf, and E. Visser. Webworkflow: An object-oriented workflow modeling language for web applications. In K. Czarnecki, editor, *International Conference on Model Driven Engineering Languages and Systems (MODELS08)*, Lecture Notes in Computer Science. Springer, October 2008.
- [26] Jacob Hookom. Facelets - javaserver faces view definition framework, developer documentation.
- [27] P Marden Jr and E Munson. Psl: An alternate approach to style sheet languages for the world wide web. *Journal of Universal Computer Science*, Jan 1998.
- [28] Håkon Wium Lie. *Cascading Style Sheets*. PhD thesis, Faculty of Mathematics and Natural Sciences - University of Oslo, May 2005.
- [29] Q Limbourg, J Vanderdonckt, B Michotte, and L Bouillon. Usixml: A user interface description language for context-sensitive user interfaces. In *Advanced Visual Interfaces Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages" (AVI 2004)*.
- [30] José Massó, Jean Vanderdonckt, Francisco Simarro, and Pascual López. Towards virtualization of user interfaces based on usixml. In *3D Web technology (Web3D '05)*, Mar 2005.
- [31] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
- [32] Peter Morville and Louis Rosenfeld. *Information Architecture for the World Wide Web*. O'Reilly, 1998.
- [33] J Nielsen and R Molich. Heuristic evaluation of user interfaces. In *Conference on Human Factors in Computing Systems (CHI 1990)*, Jan 1990.
- [34] A Puerta. The mecano project: Comprehensive and integrated support for model-based interface development. In *Computer-Aided Design of User Interfaces (CADUI '96)*, Jan 1996.
- [35] Angel Puerta and Jacob Eisenstein. Ximl: a common representation for interaction data. In *Intelligent user interfaces (IUI '02)*, pages 214–215, 2002.
- [36] Angel R. Puerta and David Maulsby. Mobi-d: a model-based development environment for user-centered design. In *CHI '97: CHI '97 extended abstracts on Human factors in computing systems*, pages 4–5, New York, NY, USA, 1997. ACM.

- [37] D Raggett, A Le Hors, and I Jacobs. Html 4.01 specification. *W3C Recommendation*, <http://www.w3.org/TR/html401>, Jan 1999.
- [38] N Souchon and J Vanderdonckt. A review of xml-compliant user interface description languages. In *Conference on Design, Specification, and Verification of Interactive Systems. Volume 2844 of Lecture Notes in Computer Science*, pages 377–391. Springer-Verlag, 2003.
- [39] Adrian Stanciulescu, Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, and Francisco Montero. A transformational approach for multimodal web user interfaces based on usixml. In *International Conference on Multimodal interfaces (ICMI '05)*, Oct 2005.
- [40] R. E. Kurt Stirewalt. Mdl: a language for binding ui models. In *Proceedings of the third international conference on Computer-aided design of user interfaces*, pages 159–170, Norwell, MA, USA, 1999. Kluwer Academic Publishers.
- [41] P Szekely, P Sukaviriya, P Castells, J Muthukumarasamy, and E Salcher. Declarative interface models for user interface construction tools: the mastermind approach. *Engineering for Human-Computer Interaction*, pages 120–150, 1996.
- [42] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.
- [43] Arie van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In D. Tamzalit, editor, *CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007)*, pages 41–49, Amsterdam, The Netherlands, March 2007.
- [44] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Berlin, 2008. Springer.
- [45] N Walsh and I Jacobs. Architecture of the world wide web, volume one. *W3C Recommendation*, <http://www.w3.org/TR/webarch>, Jan 2004.

Appendix A

Overview of Style Properties

Property	Type	Value	Example
align	Align	<ol style="list-style-type: none">1. left2. right3. center	<code>align := Align.center;</code>
background-color	Color	<ul style="list-style-type: none">• #hexcolor• e.g. white, blue, black, ...	<code>background-color := Color.white;</code>
border-color	Color	see background-color	
border-style	BorderStyle	<ol style="list-style-type: none">1. solid2. dashed3. dotted4. double5. none	<code>border-style := BorderStyle.solid;</code>
border-width	Length	e.g. 1em, 2px, 100%	<code>border-width := 1px;</code>
font	Font	e.g. Lucida, Arial, Verdana, ...	<code>font := Font.Arial;</code>
font-color	Color	see background-color	<code>font-color := #ffff00;</code>
font-line	Line	<ol style="list-style-type: none">1. under2. over3. through4. none	<code>font-line := Line.under;</code>
font-size	Length	see border-width	<code>font-size := 12pt;</code>

Property	Type	Value	Example
font-style	FontStyle	1. italic 2. bold 3. normal	<code>font-style := FontStyle.bold;</code>
height	Length	see border-width	<code>height := 100px;</code>
image	Uri	<code>url('url string')</code>	<code>image := url('image/logo.png');</code>
image-repeat	Repeat	1. horizontal 2. vertical 3. both 4. none	<code>image-repeat := Repeat.horizontal;</code>
line-height	Length	see border-width	<code>line-height := 12pt;</code>
margin	Length	see border-width	<code>margin := 1em;</code>
orientation	Orientation	1. horizontal 2. vertical	<code>orientation := Orientation.vertical;</code>
padding	Length	see border-width	<code>padding := 1em;</code>
separator	Separator	1. pipe 2. dot 3. comma 4. semicolon 5. tab 6. none	<code>separator := Separator.tab;</code>
spacing	Length	see border-width	<code>spacing := 2em;</code>
width	Length	see border-width	<code>width := 100%;</code>