Abstractions for Asynchronous User Interfaces in Web Applications

---

MSc Thesis
Version of November 1, 2009

Michel Weststrate

Abstractions for Asynchronous User Interfaces in Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Michel Weststrate
born in Bergen op Zoom, the Netherlands

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

Abstractions for Asynchronous User Interfaces in Web Applications

Author: Michel Weststrate
Student id: 1174851
Email: mweststrate@gmail.com

Abstract

The web has become a popular target platform for applications. The differences in user experience between browser based applications and desktop applications have become smaller due to the enriched user experience enabled by the Ajax technique. Nevertheless, developing Ajax based internet applications is a complex task and requires the developer to intertwine many standards and languages. This thesis presents the WebDSLx extension for the WebDSL compiler. The compiler and extension allows the developer to create rich internet applications in an intuitive way, since many details are taken care of by the abstractions provided in the model. WebDSLx allows to define a rich interaction pattern between user and application, based on the concepts of delta updates and reusable widgets. The abstractions provided are based on an analysis of proven successful abstractions in the development process of desktop applications. As a result, the WebDSLx extension provides a simple model based approach to a formerly tedious task.

Thesis Committee:

Chair:	Prof.dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Msc. D.M. Groenewegen, Faculty EEMCS, TU Delft
Committee member:	Prof.dr.ir. G.J.P.M. Houben, Faculty EEMCS, TU Delft

Preface

I am grateful to the people who supported and guided me during the process of writing this thesis. First off all, I want to thank my supervisor, Eelco Visser, for challenging me to build Ajax applications using WebDSL. His vision that models should be elegant and clear caused me to review and refine my work again and again until its current state. I want to thank Danny Groenewegen, my daily supervisor, for learning me to work with the tool set. Furthermore the many discussions were inspiring and resulted in better design decisions. I want to thank the research group for allowing me to use their facilities and for providing me a glimpse of the scientific world. I want to thank Zef, Felienne, Lennart, Sander, Underscore Sander, Maartje and Rob for sharing their knowledge and the warm welcome in their midst. I really appreciate it. Finally I want to thank my beloved wife Elise for her encouragement and God for providing me valuable talents and a good place to develop them.

List of figures

List of Listings

Contents

chapter 1 Intro

Applications such as Gmail and Google Maps introduced a new breath of web applications to many internet users. Such applications that smart and provide a rich, responsive user experience. The classic click-and-wait disappears. In recent years the number of such web-applications has grown steadily.

A technique called *Ajax* (Asynchronous Javascript and XML) enables developers to build web-pages that provides a better user experience. This technieque has resulted in numerous web applications that have the look and feel of desktop applications. Many tasks that were traditionally done using online web applications, can now be done using web applications. Good examples of this trend are Google Docs, GMail and Exact Online. The core of the Ajax technique is sending requests asynchronously to the server. By sending request in the background, the user can continue browsing while awaiting the response of the server. Although the user-experience will be improved, the use of asynchronous requests complicate the development process significantly.

From a developers view, the already complex task of building web-applications becomes a more fragile task, requiring a plethora of techniques and languages to be intertwined. Nonetheless, over the years web applications have become more sophisticated. Web applications are no longer a collection of documents, rather, they mimic the look and feel of a full-blown desktop applications. This has rendered many HTML techniques obsolete and calls for new architectures concerning the development of internet applications [18].

Traditionally, Designing web-pages was aided to a certain extend by WYSIWYG HTML editors such as Adobe Dreamweaver [45]. Enabling Ajax complicates the development of user interfaces to a level where such tools fail to predict and render the contents of web-pages. Building graphical user interfaces (GUIs) always has been an expensive process; 50% of source code and development time of GUI based software projects is dedicated to the user interface [21; 29]. This fact has caused many scientist to investigate the development of user interfaces, in an attempt to improve the process. Many successful inventions, such as interface builders and GUI dedicated scripting languages, are a result of this research.

Since the user interface of web applications become more and more similar to the look and feel of common desktop applications, the architecture of applications also becomes more similar. To avoid reinventing the wheel, and to learn from mistakes made in the past, this thesis will investigate the development process of both traditional desktop and modern web applications. We will investigate what techniques are successful in building desktops application interfaces, and which abstractions can be reused in Ajax, to provide the Ajax developer better support, enabling more rapid development. The lessons provided by a few decades of research might hint the directions that should be taken when inventing new development tools for the web.

1.1 Research question

Observing those trends in the development of web applications, namely mimicking the look and feel of desktop applications by using Ajax, brings us to the research question of this thesis:

"Can we devise a declarative model
that captures the complex asynchronous behavior and rich experience
of dynamic web applications?"

Since there are many successful methodologies in the history of desktop GUI development, the model will be based on the ones which haven been proven to be successful. There are two important notions that enable the characterizing vivid look and feel of Ajax applications. Those two notions will both be discussed in this document and dealt with in the proposed model. The first notion is the quick adaption of the user interface based on user input, which is enabled by asynchronous communication. The second notion is the use of rich interface widgets, which are very similar to the ones used in desktop applications, and provide better interaction than the limited set of widgets available in the HTML standard.

The ultimate goal of the model is to hide the fact that one is developing a web applications as much as possible. For this reason the model should abstract over the complex interaction model of Ajax and hide any state management problems introduced by the client-server model and delta updates. Furthermore defining user interfaces should not be a more complex level than creating user interfaces for ordinary desktop applications, which requires to be able to define complex reusable widget libraries.

This research will be executed in the WebDSL environment, which provides a declarative model for synchronous web applications. The complete set of functionality added to the WebDSL compiler to support Ajax, will be referred to as WebDSLx. The terms *asynchronous behavior* and *web applications* will be explained in more detail as the reader progresses through this document. As an introduction, the next section describes what we, as software engineers, try to achieve by our quest for declarative models.

1.2 Declarative models

Humans always have been trying to optimize whatever task they tried to accomplish. In the programming world, programmers try to create programming abstractions in order to reduce the amount of time spend on a certain task, or to avoid executing boring and reoccurring tasks as much as possible. One tries to program in a time and presumably money efficient way.

The process of instructing computers to perform certain calculations evolved from creating punch cards to writing sentences in a programming language in some kind of very formal English. And still, the software engineer is not satisfied; when writing a program, he desires to reduces the number of sentences required. This should increase the maintainability and (again, presumably) the readability of a program. If it is somehow possible, we like to tell a program what goal we like to achieve, without telling how to achieve it. Although it lacks an exact definition, telling a program *what* to achieve without telling *how* (or even *when*) is referred to as providing instructions *declarative.* In this thesis we try to instruct a *WebDSL model* what behavior should be achieved in the browser, without giving any details about *how* it should be achieved. Not telling *how* has some major advantages:

- The developer does not need to know how something can be achieved, only *what* he wants to achieve.
- Since the *how* part is thought through and encoded by an expert and encoded into the compiler (a program that transforms the sentences of a model into computer instructions), it might be very well optimized.
- When the target platform changes or new technologies are being developed, those external factors can be embraced by embedding the necessary translations to the compiler. After that, the programs only need to be recompiled in order to support the changes.

Listing 1.1 shows a small WebDSL snippet where the compiler is instructed that the user of our web application should somehow enter his Curriculum Vitae. How the input form looks like, how data is saved, and what the type of a CV object actually is, will all successfully be determined by the WebDSL compiler if we do not provide more information, which is a small example of more telling *what* than *how.*

```
form {
      input(user.CV)
      action("Save", save())
      action save {
            return profilePage(user);
      }
}
```

Listing 1.1: Small example of a declarative model in WebDSL

1.3 Structure of this document

This thesis starts with describing the history of internet as an application platform. Chapter 2 describes how the world wide web started as a hypertext sharing platform, but evolved to a system which can be used to run complex web applications. Chapter 3 to 6 summarize the most important trends in the development of desktop applications. Many parallels can and will be drawn between the development process of user interface for desktop and web applications. Based on this comparison recommendations will be made in chapter 7 on how asynchronous web-based user interfaces should be developed. Chapter 8 will introduce the WebDSL language for developing web applications, and evaluate it.

Chapters 9 to 11 will introduce the concept of asynchronous interaction and delta updates and the abstractions provided to deal with those issues. Chapter 12 and 13 introduce Ajax frameworks and widgets, and explain the mechanisms to embed those in WebDSL applications. Chapter 14 and 15 present two case studies built using WebDSLx. Finally in chapter 16 this work will be evaluated.

chapter 2 A brief history of the World Wide Web

Over the past two decades the World Wide Web has become one of the most popular media. The concept World Wide Web was proposed by Sir Berners-Lee in 1989 [31]. Its foundations lay in the TCP/IP [46] protocol to transport web pages (using the HTTP protocol [47]) which can be viewed in a web browser. The introduction of the first popular graphical browser, Mosaic [1], is generally seen as one of the main keys to the success of the world wide web.

2.1 HyperText Markup Language

HTML (HyperText Markup Language) quickly became the document format for web pages. HTML and XHTML, its successor, are both maintained by W3C (World Wide Web Consortium), a consortium initiated in 1994 by Sir Berners-Lee. HTML was based on SGML, and proposed in 1991. The last HTML specification was published in 1999; version 4.01 [48]. In 2000 HTML became an official ISO standard. One of the most important features of hypertext is the ability to navigate (link) to other pages. Traditionally a web-site is a collection of web-pages hosted at a (single) web-server, linking to other pages using (relative) *anchors.* A web-browser renders a single page which will completely replaced by a new page when a link is clicked. We refer to this behavior as the *classic web interaction model*. This is an important notion that directly influences the way that the behavior of websites is specified.

A web-page consists of a hierarchical ordered set of objects. Objects can either be meta-information (e.g. content-type, search keywords), visual objects (tables, lines, images) or text. As most software engineers are very familiar with HTML, we will not discuss it in more detail. Listing 2.1 is an example of a small HTML page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
  <meta content="text/html;charset=ISO-8859-1"
    http-equiv="Content-Type">
  <title>Hello world</title>
  </head>
  <body>
    <br>
      <table style="width: 600px;" border="1">
      <tr>
        <td style="vertical-align: top;">
          this is a
          <big><big><big>big</big></big></big>
          <span style="font-weight: bold;">fat</span>
          text
        </td>
        <td style="vertical-align: top;">
          <img style="width: 300px; height: 201px;"
            alt="composer image" src="composer.png">
          <br>
        </td>
      </tr>
    </table>
  </body>
</html>
```

Listing 2.1: Small HTML page

2.2 XHTML

W3C introduced XHTML in 2000 [41] as being the redefinition of HTML in XML (eXtensible Markup Language (section 4.1). XHTML is fully XML compliant and therefore XHTML can be parsed by any XML parser. XML is more strict than SGML based HTML, so by default HTML pages are not XHTML compatible, although the transformation to XHTML is quite trivial in most cases. For example the HTML break <*br*> needs to be written as <*br/*> in XHTML, since it has no child nodes. Furthermore all tagnames should be in lowercase.

Besides the generic format, one of the main advantages of XML is extensibility; XML enables the embedding of other XML based formats, such as MathML (Mathematical Markup Language) or SVG (Scalable Vector Graphics), in XHTML.

2.3 Server side scripting languages

HyperText has a very static 'program flow'. It can navigate from one page to another, and optionally submit some data to the web-server. In addition to the static behavior pattern of HTML, pages themselves also tend to be very static. In the early days of the internet, HTML pages were stored as text files. Such pages usually were created by using some offline (WYSIWYG) HTML editor (section 5.3). For this reason changing a page is not trivial, as someone needs to fetch the file from the server, edit the contents using an editor and upload it again. Furthermore, HTML provides

support for defining forms, by which users could input data and submit it to the server. To generate HTML pages dynamically, based on the input of users, server-side scripting languages have been invented. The first approach to handle user input was by executing Perl or C programs using the Common Gateway Interface (CGI)[38], which was introduced in 1993. This protocol described how requests from, and submits to the server are translated into program invocations. The raw data from the incoming connection is fed to the program's input stream, and the output stream will be send back to the client, usually containing HTML and some HTTP headers.

Shortly after the introduction of CGI, dedicated scripting languages appeared, enabling the dynamic generation of HTML pages. Besides providing many web-related functions (like URL encoding) and abstractions for reading input data (cookies, POST and GET data etcetera) those languages made it easier to generate HTML. Using a GPL, HTML codes need to be printed into the output stream as strings which requires complex escape codes to avoid ill-formed source code.

Dedicated languages such as PHP (1995) and ASP (1997) provide syntactic escapes to HTML. This not only avoids juggling with quotes, but also keeps the pages partial readable by HTML interpreters, that can omit the scripting parts. The "welcome" header in Listing 2.2 for example, can be edited using an arbitrary HTML editor. WYSIWYG editors like Dreamweaver [45] can handle such files and render the HTML parts of PHP files, although rendering without executing the script often results in elements such as tables being empty.

Server-side scripting provides one of the foundations of dynamic websites and web applications. As *Content Management Systems* (CMS) powered by these scripting languages became more popular (2000 and onward), the usefulness of WYSIWYG editors partially disappeared, since pages can be edited online using the web-browser, which avoids the long download-edit-upload round-trip.

```
<h1>Welcome</h1><hr/>
<?
  $visitor = 173;
  //printing HTML using the echo command, note the quote escapes
  echo '<span style=\'color: red\'>you are visitor '.
    $visitor.'</span>';
  //printing HTML using a syntactical escape
?>
<span style='color: red'>
  you are visitor <? echo $visitor;?>
</span>
<?
//more code..
?>
```

Listing 2.2: PHP and HTML escaping

2.4 Stylesheets

The development of Cascading Stylesheets (CSS) is mainly driven by a major drawback of HTML; HTML is hardly able to style a web-page, especially when separation of style and layout is preferred. CSS was introduced in 1994, and quickly adopted by the W3C. However the adoption of CSS by browsers was very slow. Internet Explorer 5 (march 2000) was the first to support CSS 1 for 99%. The previous HTML Listing 2.2 shows some in-lining of CSS using the *style* attribute. CSS can be used to manipulate position, dimension, float, color, border, font (etcetera) of HTML elements.

Besides some shortcomings in the constructions provided by CSS [14] the major drawback of CSS is the inconsistent implementation in several main-stream web-browsers. This requires developers to use a plethora of CSS hacks or distinct CSS definitions for different browsers. Those particular issues, however, are beyond the scope of this research.

2.5 Delta updates

Ajax (Asynchronous JavaScript and XML)[1] is the result of combining several existing techniques and languages: JavaScript (especially the *XMLHttpRequest* (XHR) object), (X)HTML and CSS. It radically changed the architecture and behavior of many web-sites. The core feature of Ajax is that it works asynchronously [6]. In the traditional HTTP approach pages are submitted as a whole, the web-server generates a new page and sends it back to the user. All those steps take a while, forcing the user to wait after each request.

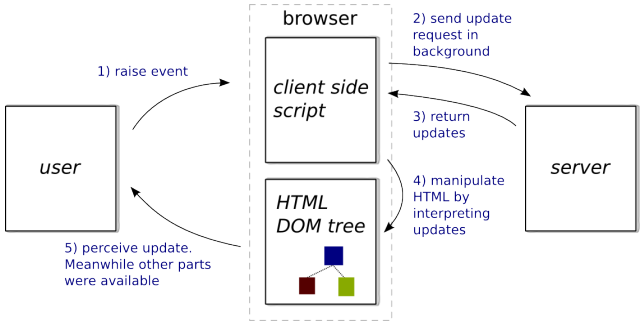

Figure 2.1: The interaction model of web applications using delta updates

Using Ajax (Figure 2.1) data can be send in the background, enabling the user to browse further (to a limited extend) until a response is received. An AJAX response usually does not replace the whole page, as in traditional HTTP, but only the part that should be changed as a result of the users action. *The term Single Page Interface* is coined for this pattern.

1 The term *Ajax* is coined by Garret [6], although there are several other terms that describe approximately the same idea, like *DHTML* (Dynamic HTML), *SPI* (Single Page Interface) or *RIA* (Rich Interface Applications).

In contrast to the traditional approach, this results in pages being send in parts -called *delta updates*- across the line. Those parts result a natural division of the user interface, since a part that is replaced or appended to the page, often forms a logical unit in the application.

From a technical point of view Ajax does not introduce many new features in the browser, since AJAX was build upon existing techniques. However, the behavior of many websites changed radically, as the look and feel of common desktop applications became available to the internet application. This introduces useful behavior patterns like suggestions while typing and displaying details of some object -which are loaded in the background- while hovering over a record.

From a user point of view, Ajax based web applications have a far better look-and-feel. In addition to nice features like auto suggestions, the delta updates should also reduce the server load, as requests and responses become smaller (unchanged parts of the page are in fact cached at the client). Combined with the fact that requests are executed in the background, delta updates should result in a reduced user-perceived latency [18].

2.6 Evolution

The world wide web evolved from a hypertext serving platform to an application serving platform, in which the browser can function as a generic user interface. Although the enhancements build on top of the hypertext system enables to write more complex applications, the development model is also severely complicated since the plethora of techniques introduced during this evolution are responsible for many of the quirks web developers encounter.

In the next chapters we will investigate which abstractions and approaches are provided to handle the complexity of the user interface of (administrative) desktop applications, and we will discover how those abstractions have been, or can be applied to the domain of web-applications. Interface composition in synchronous interfaces.

chapter 3 Synchronous interface development

The quest for an effective and efficient GUI (graphical user interface) development process is ongoing since the seventies. Many successful and some less successful attempts to improve the process have been made. As the window-manager based operating systems appeared (the most famous being MacOS, Windows and Linux/X) many toolkits to develop GUIs have been introduced. This chapter introduces the functionality provided by the operating systems to create graphical user interfaces. Traditionally *general-purpose languages* are used to define the GUI, but HTML allows for a more declarative approach.

3.1 Desktop application foundations

Nowadays, most computer users use graphical applications to execute administrative tasks (reading e-mail, writing documents, fill out tax-forms etcetera). In modern operating systems, those applications are managed by the window-manager. The OS enables multi-tasking by providing a multi-threaded model and event dispatching. Furthermore, it provides libraries to manipulate the screen using primitive draw functions or complex controls (widgets).

In Windows, as displayed in Figure 3.2, most activity starts as a response to a *Message* send by the OS (steps *1* and *2*). Messages consists of a *window_handle* a *message_ID* and two *parameters.* Using the *focus state* (which object is currently selected by the user) and/or the window_handle the message is dispatched to the targeted control (steps *3* and *4)*. In the event-driven model (Section 6.2) this results in a callback being invoked (step *5*). Those callbacks are provided by the application developer.

All operating systems provide an API to enable the usage of controls and to specify callbacks. For instance, Microsoft provides a widget library in form of the Microsoft Foundation Classes. Those controls abstract over the message system and primitive draw functions. They provide a consistent layout and process all common messages, repainting or firing events when necessary.

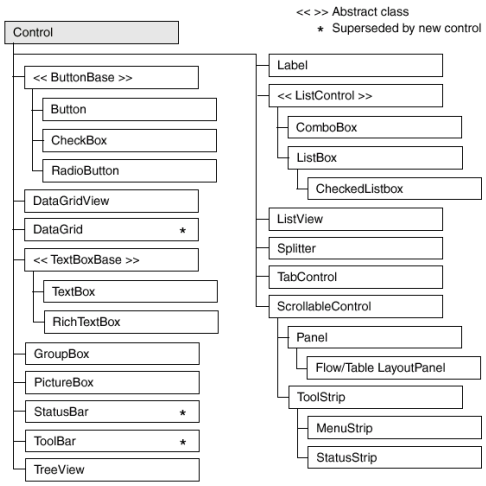

Figure 3.1: Inheritance tree of the MFC controls

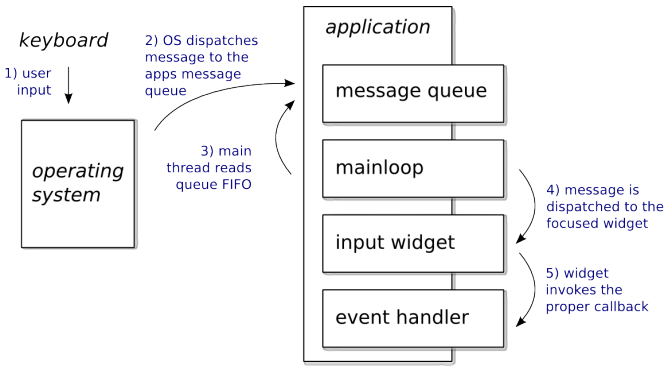

Figure 3.2: Event Dispatching

By using those controls the application developer is only required to set common properties and write event handlers. To extend behavior or to compose different controls, controls can be sub-classed to override specific parts of their behavior. In fact, most MFC controls are build by sub-classing other controls, as shown in Figure 3.1. The effectiveness of this sub-classing system is demonstrated by the numerous third-party widget libraries available, based on those classes.

3.2 Programmatic interface specification

The most direct approach to invoke functions of a system library is by using a general-purpose language (GPL). For this reason the programmatic approach is the oldest approach for specifying user interfaces. Despite the vast number of tools that aid the creation of user interfaces, many user-interfaces are still 'hand-written' using a GPL. Besides some initialization code, writing a user-interface generally consists of the three steps demonstrated in Listing 3.1.

```
public class DisplayFrame {
  public static void main(String[] args) {
    //(1) Object creation
    JFrame f = new JFrame("A Frame");
    f.setSize(300,200);
    JButton button = new JButton("On/Off");
    button.setForeground(Color.black);
    //(2) Composition
    f.getContentPane().add(button,
      BorderLayout.SOUTH);
    //(3) Adding events
    button.addActionListener(new ActionListener() {
```

```
      public void actionPerformed(ActionEvent e) {
        //Event listener implementation goes here
      }
    });
    //Start the event-thread
    f.setVisible(true);
  }
}
```

Listing 3.1: Building an user interface using Java

3.2.1 *Object creation*

As demonstrated in Listing 3.1, line 7, the creation of objects is usually done in (a function called by) the application's *main* method. This starts with the creation of a *window* (sometimes called *form* or *frame*) and all of its children (line 7 and onward). The required widgets can both be provided by the vendor of the GPL or a third-party library. In general, there is much similarity between different widget libraries. After or during the creation of a widget properties that define the layout, style or behavior of the widget might be set, for example the size of the Frame and the caption of a button.

3.2.2 *Layout & composition*

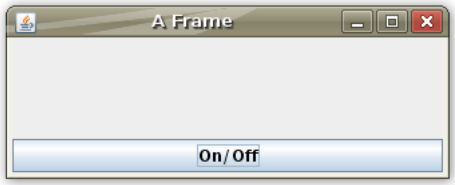

Figure 3.3: The interface generated by Listing 3.1

The process of describing the layout by defining the relations between widgets is called composition. Describing the composition consists of two steps. First the hierarchy of objects needs to be defined. In the example above the frame is made *parent* of the button, by adding the button to its contents frame (lines 12-13). This hierarchy describes the natural relation between objects. A widget that can contain other widgets is called a *container*. Containers often can deal with layout related issues automatically. For example when the frame is resized, this information is propagated to its children (the button in figure Figure 3.3). This relation might also work the other way around, for example when the button requests more space, which can be provided by the frame. When the container is hidden this automatically affects the children.

The precise relation between the container and its children depends on the *layout manager* or *layout algorithm*. For example the AWT BorderLayout anchors its children to a specific border, *South* in our example. Other layout managers might describe that all children take the full available height and are arranged next to each other. A grid layout is available in many UI library and describes a table-like layout, using rows and columns.

3.2.3 *Interaction*

In most libraries the interaction with the back-end is defined by using event-handlers (other methods are examined chapter 6). The responsibility of the event-handler is to execute a specified procedure

when the user undertakes a specific task, for example clicking a button. In Listing 3.1 an empty handler is attached to a button (lines 15-19). One procedure can be attached to multiple events, and in several languages, multiple procedures can be attached to a single event. To start handling events the *mainloop* (Figure 3.2) needs to be started after initializing all the objects.

3.2.4 *(Dis)advantages of programmatic user interface specification*

The storage of a user-interface in a general-purpose language (GPL) is very straightforward. A list of statements should reproduce the designed interface. Those statements are usually stored in a text-based file. Using source files has the advantage that no additional syntax need to be specified to store the user interface description. Furthermore, defining an interface using a GPL is very flexible since the full power of a GPL can be used to influence the user interface. For example complex size calculations can be expressed and unusual user interfaces can be developed. Such as for CAD/CAM software or games. Besides, depending on the GPL, one receives syntax and type checking for free.

However, due to the imperative nature of GPLs this approach has some drawbacks: The user interface descriptions are often hard to understand, hard to analyse and have very poor interoperability with other languages or user interface libraries. Moreover, although the programmer has full control, to build a full-blown user interface many compositions and properties need to be defined. Using the verbose code of a GPL writing a GUI is a time-consuming task [27]. Furthermore, inconsistencies in the user interface are easily introduced when all forms are build from scratch [22]. Those observations drove the search for other approaches.

3.3 Interface composition approaches

Given an operating systems that provides an event system to handle user input and libraries to compose graphical user interfaces using widgets, several approaches can be taken to simplify the specification process. The next two chapters elaborate on those approaches, and describe its implications on web interface development.

The programmatic approach Using a general-purpose language, as described in this chapter, each object in the interface is initialized, its properties are set and the event-handlers associated. This is the oldest approach, and provides the most flexibility. It is a result of directly approaching the operating system libraries using a general-purpose language.

The model approach Many textual models have been proposed to specify user interface. Such models are interpreted or compiled to generate the interface. In general models are specified by either using *XML* or a *domain specific language* (DSL).

Interface builders Interface builders provide another interface to interface specifications, enabling a more intuitive way to manage GPL- or model based interfaces. Controls are dragged and dropped on forms, enabling visual manipulation of the most important properties concerning layout and composition.

The generator approach The fourth, less common, approach is to generate the user-interface based on the semantic properties of the application. Using the domain model and the task model a user-interface is generated. Since the interface generation depends on model analysis, the application need to be described in a declarative way.

3.4 Defining user interfaces using HTML

Chapter 2 introduces the HTML format. Composing HTML based interfaces differs in two important points from the approach described in this section. First of all, HTML does not provide a library to compose controls which can be invoked by using a general-purpose language. HTML provides a textual, declarative model with a limited set of controls. Consequently, to build a HTML interface using a GPL, text needs to be generated. Listing 2.2 gives an example of this approach: HTML is generated by printing strings to the output stream of the server.

3.4.1 *Composition*

It is important to notice that rendering HTML is complicated due to the text-flow nature of documents. This in contrast to GUI libraries, where almost all widgets are rectangular and usually are positioned using absolute coordinates. As a result of this document oriented nature, HTML acts as if the vertical space is unlimited. For this reason it is, for example, very hard to define two columns with equal length, depending on the height of the longest text.

Therefore it might be hard to mimic the composition of an ordinary desktop application; since such GUIs have often a *absolute coordinate* or *constraint* based layout. Constraints relate coordinates to the dimensions of the parent widget. Anchoring at a specific distance from the edge of the parent is hard to achieve in HTML, as it often lacks the necessary styling properties, resulting in a complex composition of *tables* or *div* elements.

HTML is very powerful in determining sizes based on the contents. Fixed sizes however, are hard to achieve, as the most popular browsers, Internet Explorer and Firefox, have a different interpretation of the concepts of height and weight, which resulted in the famous box-model problem [49]: In the W3C model the *width* attribute of an element defines the width of the contents. Older versions of Internet Explorer however, interpret the width as being to total space available to the contents, borders and padding.

3.4.2 *HTML generation*

HTML is a declarative language. However, there exists two common scenarios that result in a programmatic approach of HTML: First of all, server-side scripting languages generate pages on the fly. This might be done by either filling gaps in HTML templates, or by generating HTML code directly using print statements. The latter approach is demonstrated in Listing 2.2. Using prints statements to generate HTML neglects all the advantages of the declarative nature of (X)HTML.

Secondly, client-side scripting also allows for creating HTML objects on the fly. Many Ajax-frameworks provide a JavaScript API to instantiate widgets, in a way that is very similar to the programmatic approach described in this chapter. Listing 3.2 provides an example of this imperative approach.

```
Ext.get('mb4').on('click', functione){
  Ext.MessageBox.show({
   title:'Save Changes?',
   msg: 'Would you like to save your changes?',
   buttons: Ext.MessageBox.YESNOCANCEL,
   fn: showResult,
   icon: Ext.MessageBox.QUESTION
  });
});
```

Listing 3.2: Programmatic instantiation of widgets using JavaScript

3.5 What is next

This chapter introduced the functionalities the operating system provides in order to render graphical user interface and to process events raised by the user. Traditionally programmers need to use a general-purpose language to design user interfaces. But, as we will discover in the next chapters, new approaches have been developed.

The HTML standard avoids the need for writing GPL code in order to produce an interface. However, the introduction of dynamic pages results a fall-back to the programmatic approach. The next chapters will provide some solutions to solve this issue. Furthermore in chapter 6 we will discover that the event dispatch model introduced in section 3.1 does not match the interaction model provided by HTTP/ HTML.

chapter 4 Textual interface models

Creating a GUI by directly calling the appropriate methods using a GPL is a tedious process for the developer. It requires a vast amount of time to write the numerous function invocations that produce the user interface. For this reason, many languages have been developed with the purpose to ease the specification of user interfaces. Those languages enable the developer to define the user interface declaratively. Interface models provide a simplified abstract view over details which are required by the GPL, but are not very interesting to the developer, such as the names to identify a control, creation order etcetera. Generally speaking, models can be defined using a textual or a visual approach. This chapter introduces textual models. Visual models are described in chapter 5. The text-based storage of user interface models can be divided into two categories:

XML based storage. By using the XML format the composition of a user interface is stored in a hierarchical structure using a generic syntax.

DSL based storage. A custom syntax is developed to model the user interface. A well designed syntax provides very human-friendly access to the model.

Note that there exist other languages than XML that can be used to store a user interface in a hierarchical structured file, for example JSON (JavaScript Object Notation). This chapter elaborates only on the first one, since it is used by many interface generators and interpreters. In the next sections the XML and DSL based approaches will be examined.

4.1 eXtensible Markup Language

XML (eXtensible Markup Language) is intended to be a general-purpose specification language and proposed by W3C [39]. XML enables textual, structured storage which is intended to be both easy to read and write by humans and programs. The structure of a XML document can be validated against a DTD (Document Type Definition) or XSD (XML Schema Definition) schema. A document which conforms to such a schema is called *valid*. An XML document can exist of definitions taken from several schemas. Namespace identifiers can be used to avoid ambiguities.

XML is a well supported standard. XML libraries are developed for virtually every programming language. Although XML can be edited using plain text editors, many useful XML editors exist. Many recent tools use XML files to store the description of a graphical user interface. Storing a user interface using XML has three advantages:

- The hierarchical structure of XML matches the hierarchical structure of graphical user interfaces.
- XML can be validated against XML schemas and analysis and transformations are easy to implement due to uniform structure of XML.
- Interoperability is achieved easily as no new parser or schemas needs to be defined for usage by other tools.

Since XML has a fixed, declarative structure, the downside of XML is the lack of imperative expressiveness. Most XML formats have no possibility to store expressions or other forms of interaction in a both concise and verifiable way. Therefore XML models usually provide hooks to which functions -defined in a GPL (usually JavaScript)- can be attached. These back-doors usually hamper the interoperability. Such code is stored in a plain string and cannot be validated against a schema. Furthermore, the usage of namespaces and both open and close tags result in a verbose language.

4.1.1 XUL

XUL A good example of an XML-based user interface model with JavaScript hooks is the *XML User Interface Language* (XUL). XUL [50] is a XML based specification for lightweight, cross-platform, cross-device user interfaces [33]. XUL was invented by the Mozilla Foundation, with the mission to make user interfaces as easy to customize as web pages. XUL applications can be interpreted by XULRunner or the Firefox web browser. The project is intended to be able to define a user interface which can be used by programs written in an arbitrary programming language, however, except for C++ and Java, the necessary libraries are not available yet.

The XML can be validated against the XUL namespace [51]. XUL is more than just XML; for styling CSS is used and arbitrary XHTML elements can be used inside a XUL document. The XML Binding Language (XBL)[42] can be used to extend existing widgets, or override properties. Interface or application logic needs to be encoded using JavaScript. XUL natively supports the Resource Description Framework (RDF)[52] to embed dynamic content. RDF files contain (self-describing) data. The relation between RDF and XUL is similar to the relation between XML and XSLT. Listing 4.1 displays an example of a simple XUL application.

```
<?xml version="1.0"?>
<window id="vbox example" title="Example" xmlns=
  "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <script language="JavaScript">
    function yesclicked(){
      alert("You have clicked the 'yes' button.");
    }
  </script>
  <vbox>
    <button id="yes" label="Yes" oncommand="yesclicked();"/>
    <button id="no" label="No"/> <button id="maybe" label="Maybe"/>
  </vbox>
</window>
```

Listing 4.1: A simple XUL application

As shown in the same figure, event handlers are defined using JavaScript. Therefore any interpreter or compiler for XUL needs to support the JavaScript language. Furthermore, analysis of the model becomes harder. For example, usage analysis of a certain widget becomes nearly impossible as an arbitrary JavaScript event can refer to it.

4.1.2 *XHTML*

As mentioned in section 2.1, HTML provides an hierarchical approach to the composition of pages. Because of the success of XML and its well defined syntax and schemas, XML is more easy to parse and manage than HTML. For this reasons the XHTML standard (Section 2.2) was developed. XHTML uses the generic XML syntax and thereby inherits all of its advantages.

In the next section we will elaborate on a less generic approach to store interface specifications, which should provide the possibility to avoid the drawbacks introduced by using a general-purpose language or XML.

4.2 Domain Specific Languages

Domain Specific Languages (DSL) are languages which are, in contrast to general-purpose languages, designed for a very specific domain. For a small domain (user interfaces in our case) this should allow for powerful languages with a well defined syntax and high expressibility. Developing a Domain Specific Language to store user interface descriptions comes with high costs initially. It requires a new parser and (preferable) a syntax checker. However, in the end a DSL can combine the best of two worlds.

A DSL might describe the structure in a very declarative and intuitive way, avoiding the verbose tags of XML. This results in the best human-readable model format. Besides, the DSL might support a simple expression language to describe dimension calculations or the interaction with the back-end. Due to the limited domain, analysis and transformations can be achieved easily. Besides the initial effort there are several other downsides. DSL's can provide good interoperability, but not as good as XML. To support very rare expressions, one might need extensions points to a GPL or the limits of the language have to be accepted. Except for those disadvantages, DSLs are capable of combining the best aspects of several worlds, despite the fact that they often have a steep learning curve (developers need to become familiar with the syntax and semantics). In the end, using a DSL might be the most elegant and efficient approach to compose user interfaces.

4.2.1 *The Delphi Form format*

Delphi (Borland's IDE around Object Pascal) provides a simple DSL (*.dfm* file) for specifying the user interface, which gracefully integrates in the Pascal language. Listing 4.2 shows the source of a simple form in this language (which result is displayed in figure Figure 4.1). It supports a hierarchical structure, and makes the configuration of properties trivial.

```
object Form1: TForm1
  Height = 300
  Width = 400
  Caption = 'A Form'
  object Panel1: TPanel
    Height = 300
    Width = 93
    Align = alLeft
    object Button1: TButton
      Height = 25
      Width = 91
      Align = alTop
      Caption = 'Hello'
    end
  end
  object Panel2: TPanel
    Left = 98
    Height = 300
    Width = 302
    Align = alClient
  end
  object Splitter1: TSplitter
    Left = 93
    Height = 300
    Width = 5
  end
end
```

Listing 4.2: A form (figure 4.1) description in Delphi

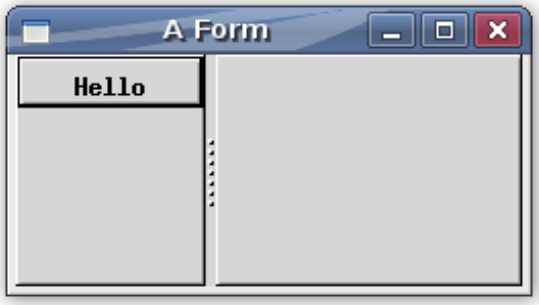

Figure 4.1: A form constructed using DFM

4.3 Template engines

In the previous chapter (Section 3.4.2) we discovered that both client side and server side scripting often results in programmatic interface definitions. This chapter indicated that using declarative languages for interface composition has many advantages. A common approach to combine scripting and the declarative (X)HTML interface definitions is the usage of templates. This approach potentially combines the best of two worlds: templates are designed to enforce a separation between model and view [25].

This separation is achieved by introducing a separate file to describe a part of the user interface; the template definition. Templates are HTML based files that embed escapes which contain constructions to print primitive variables, and loop- and conditional statements to control the output. The server script instantiates a template and provides a structure that contains all values used by the templates.

4.3.1 *Smarty Template Engine*

Smarty [53] is a commonly used template engine for PHP. Besides loops and primitive variables, it can evaluate function calls and more complex PHP expressions. Although Smarty provides complex functionality like file-inclusion, scoping and methods to render HTML options, all functionality is still focused on presentation. Listing 4.3 displays a listing to initialize the template engine and assigns some values. Note the usage of PHP functions *capitalize* and *escape* which are applied to the input, and the smarty function *now*.

```
<html>
 <head>
  <title>User Info</title>
 </head>
 <body>
  User Information:<p>
  Name: {$name|capitalize}<br>
  Addr: {$address|escape}<br>
  Date: {$smarty.now|
   date_format:"%Y-%m-%d"}<br>
 </body>
</html>
```

```
include('Smarty.class.php');
// create object
$smarty = new Smarty;
$smarty->assign('name',
 'george smith');
$smarty->assign('address',
 '45th & Harris');
// display it
$smarty->display('index.tpl');
```

Listing 4.3: An example of a Smarty template (left) and its invocation (right)

4.3.2 *Drawbacks*

Terence Parr (author of the template engine *StringTemplate*) argues that most modern template engines do not strictly separate views from the model and controller (MVC is explained in chapter 6), as they often have side effects, calculate values, have assumptions about the meaning or type of objects etcetera. For such reasons designers still need to know about the implementation details of the application, and the application cannot change without affecting the model. Furthermore, he argues that the MVC lacks a rendering aspect, as converting objects to the proper string should not be part of the view nor the controller. Still, template engines are popular, and useful in the context of AJAX applications.

4.4 Ajax and templates

Ajax libraries usually take a HTML page (or template) and add dynamic behavior to specific parts of the page, using so called *placeholders*. A placeholder usually is an empty HTML *div* with a predefined identifier. Backbase [54] uses the XML features of XHTML to embed XML objects which describe widget configuration. Those objects are interpreted by the JavaScript engine. Dojo [55] uses the traditional HTML elements but adds runtime behavior by analyzing the *dojoType* attribute. This has the nice side-effect that many pages will still work without JavaScript, although the layout will be messed up (note that disabling JavaScript in most cases also changes the behavior of the application). This is demonstrated using Listing 4.4. Figure 4.2 displays its result with JavaScript disabled, and figure Figure 4.3 displays the intended result.

```
<div dojoType="dijit.Dialog" id="dialog1" title="First Dialog"
  execute="alert('submitted!');">
  <table>
    <tr>
      <td><label for="name">Name: </label></td>
      <td><input dojoType="dijit.form.TextBox"
        type="text" name="name" id="name"></td>
    </tr>
    <!-- some more inputs -->
    <tr><td colspan="2" align="center">
      <button dojoType="dijit.form.Button" type="submit">OK</button>
    </td></tr>
  </table>
</div>
```

Listing 4.4: A small dialog modeled using the Dojo toolkit

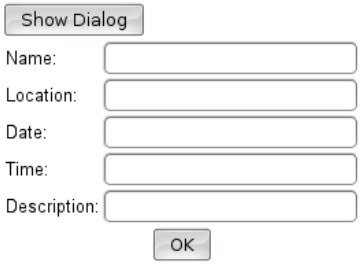

Figure 4.2: Dojo without JavaScript

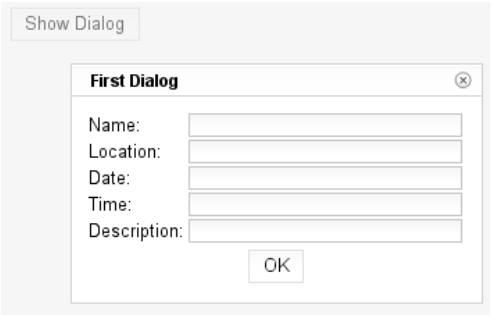

Figure 4.3: Dojo with JavaScript

4.5 XML and DSLs in the Web Domain

In this chapter we investigated some approaches to text based models of graphical user interfaces. Those approaches usually are less verbose and more easy to manage than user interfaces that are programmed using a general-purpose language.

XML is used frequently for developing user interfaces for web applications. For example by BackBase. Furthermore, defining templates is a commonly used technique to avoid the need for programmatic GUI specification. Although domain specific languages are useful to define user interfaces, they are hardly used to define Ajax based user interfaces. The next chapter introduces another approach to define user interfaces; the interface builder.

chapter 5 Interface builders

In the eighties, defining user interfaces became a hot topic in research [32], which is clearly demonstrated in the overview of Da Silva [3]. Two approaches evolved from this research: The *interface builders* (or: *visual designers*) and the *interface generators*. Interface builders provide a visual approach to designing interfaces. Rather than writing out the composition of a window, the composition is drawn. The idea behind interface builders was developed in the early eighties, resulting in, for instance, the Trillium [12] system being developed in 1981. Since that day interface builders have strongly influenced the development of graphical applications, since interfaces can be developed more rapidly and are easier to maintain [21; 23]

5.1 User Interface Management Systems

User Interface Management Systems (UIMS) evolved in parallel to the interface builders (next section) systems, and provides a non-programmatic approach to designing interfaces. The central idea behind UIMS states that interfaces should be generated by using a model that describes the tasks a user can execute using the program [16]. For instance, one of the most promising User Interface Management Systems (UIMS) was *Mastermind [30]*. Mastermind provides different layers of abstraction using several models. By combining these models the interface can be generated.

The application model defines the domain objects of the application. The domain objects are quite similar to OO-classes: they publish properties and methods. The task model provides a workflow model. Tasks consist of steps, preconditions and goals. Tasks can be fulfilled by the user, interface or application. Tasks might influence or activate each other. To express goals and conditions Mastermind provides a small expression language. Those expressions are constraint based (section 6.1); they are re-evaluated when one of the inputs changes. The Presentation Model describes the composition of the interface. Such models offer more complex behavior than most interface builders can offer. For instance, layout guides and magnitudes can be specified and complex expressions, in terms of the domain objects, that influence the interface can be modeled. Interface builders do not offer this kind of abstractions since they have no knowledge about the domain model.

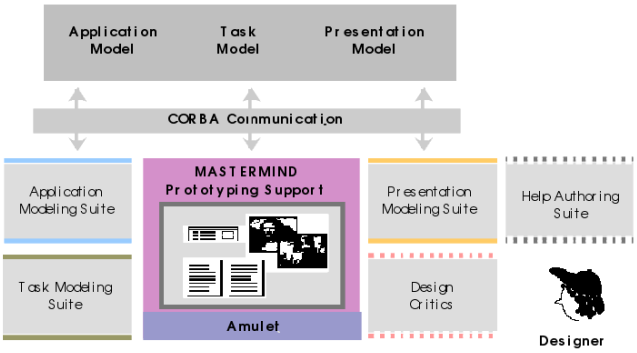

Figure 5.2: Architecture of the MASTERMIND UIMS (image taken from [30])

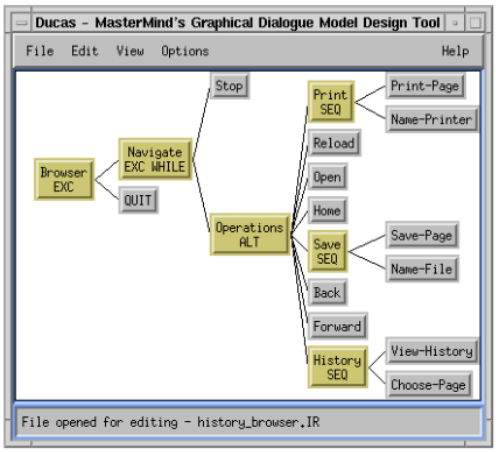

Figure 5.1: MASTERMINDS's dialog modeling tool DUKAS, which defines several kinds of relations between dialogs.

5.1.1 *Device independent models*

UIMS systems are able to abstract over different interface styles and different kinds of interfaces. Those abstractions however were irrelevant for a long time, as user interfaces for desktop applications became more and more standardized [22]. In the short future however, it is possible that we will encounter a comeback of these abstractions, as PDAs and netbooks will become popular targets of (web) applications. UIMS abstract naturally over different devices, since the interface is not specified by the developer but derived from the task model.

5.1.2 *Lack of control*

Some UIMSs omit the presentation model, and generate the user interface completely based on the application- (or domain-) and task- (or dialog-) model [1; 24] . As a result of this approach, a developer is hardly able to influence the final generated interface; fine-grained control is lost by using generators. Interface builders do a much better job when it comes to fine tuning an interface [20]. UIMS tend to abstract over the user interface; but most interface builders provide no real abstraction; they just provide a graphical interface to approximately the same code as one would write by hand. As a result interface builders introduce no 'magic'; no implicit behavior that was generated by analyzing a model. On the other hand, since UIMS introduce new languages and complex abstract concepts, they are often hard to master [3].

Many attempts have been made in order to generate user interfaces. Common acronyms that might use the generator approach are *CASE tools* (Computer Aided Software Engineering) with *code generating* capabilities and *4GL* (Fourth Generation Language) tools. However, none of the tools in this category[2] has gained wide acceptance when it comes to generating user interfaces based on higher level abstractions -models that abstract over the widgets and their composition by just describing tasks or domain objects- for reasons stated above. Nonetheless, there are promising products, which combine interface builders with domain models, such as the Mendix Business Modeler [56].

2 note that the definition of these acronyms is often not very strict, so it is disputable which tools fall into a certain category

5.2 Interface builders

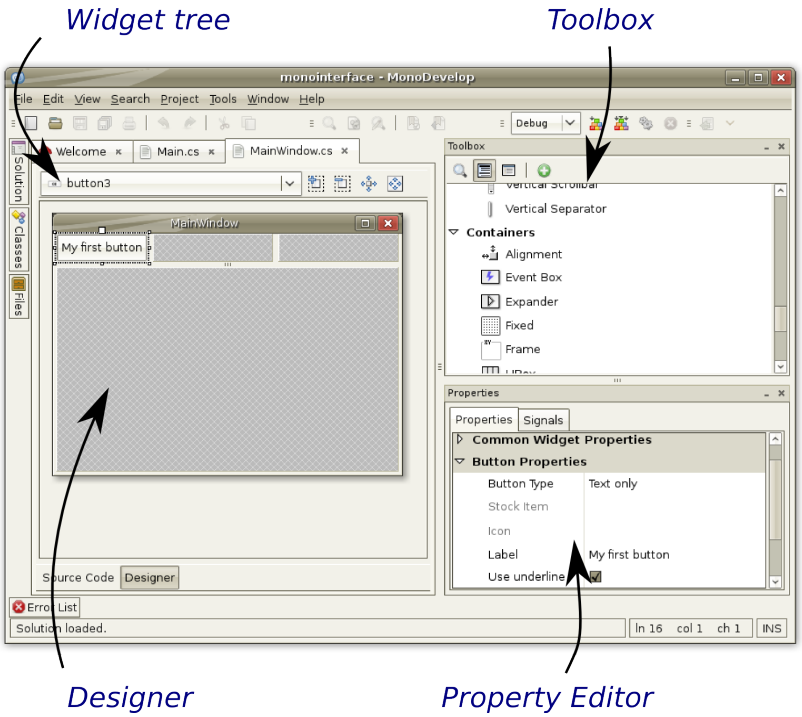

Figure 5.3: The interface builder of the MonoDevelop IDE

An *interface builder* is a WYSIWYG (What You See Is What You Get) editor that allows to interactively manipulate the layout and composition of widgets using drag and drop. This includes moving widgets, resizing using grips and specifying size constraints. Interface builders follow the steps of the programmatic approach closely, but allow the object instantiation, composition and layout to be done visually rather than writing out the required statements. A typical interface builder consists of the four parts shown in Figure 5.3.

The *toolbox* lists all available widgets. Dragging a widget to the *designer* results in the generation of the necessary statements to create a new object and initialize its parent and position properties. The available widgets depend on the libraries the interface builder is designed for. Most widget libraries provide roughly the same set of common widgets (Figure 5.4).

Figure 5.4: Typical widgets provided by most widget libraries (this one: GTK2)

Properties of a widget can be manipulated using the *property editor*. This holds for both the properties that can be edited using the designer and all other (not design related) properties. Examples of such properties are size constraints, captions, help text and behavior specifying properties. The property editor also lists the available events. An event can be coupled to a

procedure using this editor. Most editors are capable to generate procedure stubs if necessary. In such environments, all widgets share a generalized interface, which enables automatic integration in the designer and enables quick adoption of third party libraries in the designer. This generalized interface typically allows for three kinds of properties to be defined:

- Simple properties such as caption or default value
- Events such as *onclick* and *onresize*
- Child elements in composite widgets such as the panels in Figure 5.3

Finally the *object tree* provides an abstract view of the components in a form. This allows to quickly inspect or adjust parent-child relations and to select objects that are below others. Most visual designers store the model using text-based files. As a result, the visual designer is not the only interface to the contents of the model. This has some big advantages: developers can manually adjust or fix broken files, version support can be provided by tools like SVN and models are more open to other (third-party) tools.

5.2.1 *Keys to the success of interface builders*

Interface builders are still very popular today for developing graphical desktop applications, and are part of most IDEs. By using an interface builder, the number of lines of code needed to define the interface is decreased significantly. In this section we will identify some keys to the the success.

Platform Stability Desktop applications are developed for a very stable platform, since the nineties a desktop computer consists of a big color-screen, a keyboard and a mouse with an arbitrary number of buttons. Due to this stability, tools such as interface builders are able to catch up when occasionally a new technique emerges [22].

Uniformity Most interface builders and widget libraries provide approximately the same widget set. For this reason developers are able to quickly adopt new tools, libraries or even operating systems. Due to this uniformity widgets closely resembles the concepts a developer wants to use to design the interface.

Extensibility Widget classes can be sub-classed. Interface builders provide the functionality to pack those widgets into a library and integrate them seamlessly in the WYSIWYG editor. This enables the developer to quickly adopt a widget-set provided by third parties. \item[Graphical approach] Approaching a graphical problem with a graphical tool enables a developer to more easily translate his ideas into a concrete application~\cite{myers00}.

Event-Driven Visual Designers support the event driven model (section 6.2). This aids the developer in connecting his GUI with the back-end code. Some interface builders omit this feature, focusing solely on the composition of the interface.

5.3 HTML editors

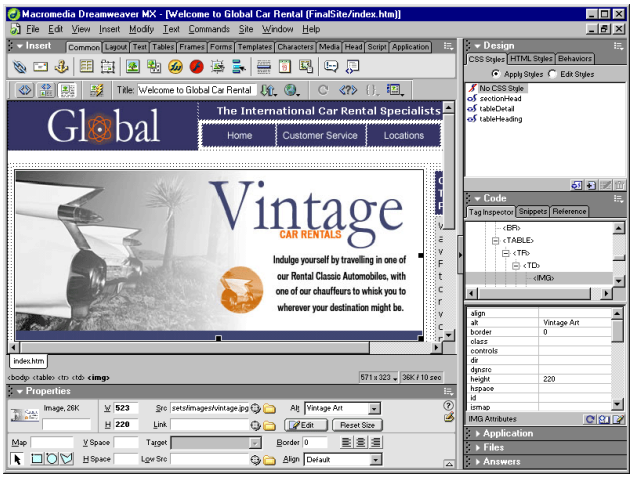

Figure 5.5: Macromedia Dreamweaver WYSIWYG HTML editor

To ease the development of HTML pages, a wide range of WYSIWYG (What You See Is What You Get) HTML editors have been developed. This enables web-developers to define a user interface by graphical means. Such editors disclose the creation of HTML documents to a very big audience. The average HTML editor does not render the contents very accurately. The most important reason is the complexity of the HTML and CSS formats. Because different browsers provide different interpretations of the formats, it is unclear to which standard an editor should conform. Furthermore, most editors render extra information, to identify the composition of the page. For example Macromedia's Dreamweaver (Figure 5.5) renders the dimensions of a table above the header, and grips to drag and drop or resize elements.

Since the rendering of the editor is not exact, most editors provides a preview function, which uses one of the browsers available at the system. To be able to adjust the HTML without the limitations of the graphical editor, editors provide a source view on the HTML. The structure of WYSIWYG HTML editors is usually very similar to the structure of interface builders; they provide, besides the designer itself, a toolbox with the most common HTML elements and a tree view reflecting the structure of the document.

5.3.1 *Ajax complications*

The introduction of delta updates not only radically changes the state management of web-applications, it also complicates design. The traditional WYSIWYG HTML editors, become partially useless since pages are build using tiny HTML parts, and editors cannot predict the composition of pages design-time. Moreover, many frameworks interpret user interface definitions in XML or JSON format at runtime which renders HTML based editors virtually useless. Although templates (Section 4.3), and the declarative approach taken by some Ajax frameworks (Section 4.4) allows the WYSIWYG HTML editors again to be used in the design process, this is rarely applied since the templates will be small all and less complicated then complete pages, reducing the added value of such editors.

chapter 6 Interaction specification

In the previous chapters several approaches to define user interfaces (views) are described. This chapter will handle the coupling between the user interface and the behavior of the application (back-end). The central problem in this chapter is how to synchronize the state of the interface with the state of the model.

Smalltalk [8] introduced the model-view-controller (MVC) pattern in 1980. This pattern describes how changes should be communicated to the program or its user. Although this MVC pattern is hardly used any more in its originally intended form, it introduces three parts we still can distinguish in a well designed graphical application.

The domain data being manipulated using the application is called the *model.* The *View* presents the model to the user; the user interface. And finally the behavior of the application; the connection between the view and the model is the responsibility of the *controller*. Several approaches exist to connect the view to the controller. In this chapter we elaborate on the event-driven approach and the constraint based approach.

6.1 Constraint based interaction

The phrase model-view-controller was introduced by the authors of Smalltalk [17]. But the current widely recognized concept of the Model View Controller architectural pattern differs from its original intentions. The central idea behind MVC is that a clear separation between the domain (model) and the presentation (view) should exist. Domain objects should be completely ignorant of the interface [5]. In classical MVC, the presentation part of an application is divided into two elements: The view and the controller. Every component (widgets or form) is divided into a view and a controller part. The controller is expected to react to user input and is notified on model

changes. It defines the relation between input devices and the model. Both controller and view observe the model. Input is propagated to the view by updating the model.

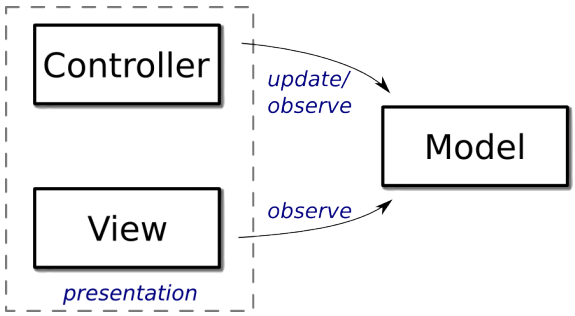

Figure 6.1: The essential relations between model view and controller in Smalltalk

The controller, however, is not event based (in the sense that the developer is required to define events) but observes the domain model and sends updates on user input. The view widgets receive a link to observe a value in the model (Figure 6.1). When the screen is initialized, widgets read their values from the model. When input is entered into a controller, the controller updates the model. Since all controllers and views observe the model, changes are propagated to the view automatically.

Using observers has the nice advantage that no code needs to be written to update the parts of the interface that should update when the model changes. This has nice side effects, for instance when multiple forms are open and inspect the same domain object, using observers all forms will be updated simultaneously when something changes. Without observers, this behavior would be hard to express, as a programmer can often not predict which forms will be open simultaneously. On the other hand, application logic that does not purely belong to the application's domain nor to the user interface, is hard to express in this MVC pattern. For example, low stock values of a product might be rendered in a red color, but this is not purely domain logic (as the domain has no notion of 'rendering' or 'red'), nor is it interface logic, as the interface has no notion of 'a low stock value'. As a solution to this problem an intermediate layer of abstraction can be introduced: the *presentation model*. But, in general, it is very hard in the MVC model to define behavior that does not directly influence or depend on the model.

6.1.1 *Constraint Systems*

Since observers can avoid the micromanagement to update the state of the screen when necessary, much research has been done on automatically propagating changes from the view to the model and vice versa. The technique to automate the propagation of changes using observers is often called *property models* or *constraint systems*.

While Smalltalk made a distinction between views and controllers, most constraint systems consist of a model and a view. A variation on this is the Abstraction-Link-View paradigm (ALV) [13] a layer is added between the model and the view, the link. The link describes consistency constraints and optional data transformation. For example the transformation of a stock-value to a string value plus color, fit for a specific widget.

6.1.2 *The mutual dependency problem*

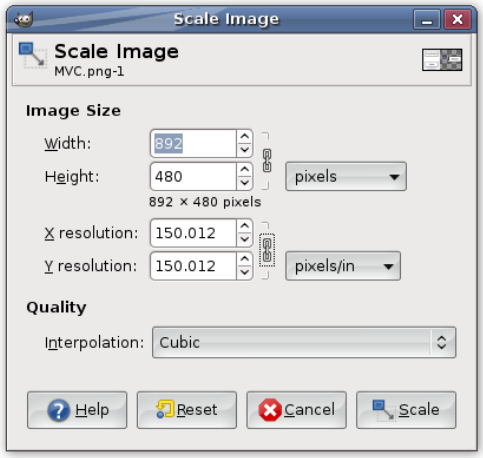

Figure 6.2: Resize image dialog. The widgets have complex constraints caused by the 'maintain aspect ration' chain

The behavior of an observer based system is very similar to the working of spreadsheets; when a user changes the contents of a cell, all depending cells will be recalculated. The most common problem in this approach are circular references; two values might depend on each other. This is not supported in spreadsheets, but this kind of behavior is often encountered in desktop applications. For example in figure 6.2 the height and weight properties depend on each other when the 'aspect ratio' is enabled. Observer based systems need to be able to handle the kind of situations were multiple values depend on each other.

Jarvi et al. [15] solve this problem by adding directions to constraint relations. They provide a declarative model to specify this kind of interactions, and use a constraint solver [34] to maintain consistency. Capturing the same behavior in an event based system is not trivial, as every event needs to deal with almost all input variables. For that reason using a declarative constraint based approach to be able to model complex dialogs is not a luxury. It promises an eight to ten fold reduce in the number of statements. The downside: there are complex constraints which still cannot be expressed in the proposed model.

6.1.3 *(Dis)advantages*

Despite the nice promises, constraint based interfaces are rarely used in practice. There are a few pointers which might indicate why [22]: First of all, outside the scientific world this approach is hardly used. The learning curve of constraint based systems is quite high, as it is a tedious work; errors in the model easily leads to errors which are very hard to debug. The behavior of constraint based systems is hard to predict, as most callbacks will be called implicitly by the system, and it might be unclear why. Last but not least, the constraint solver might come up with multiple solutions to maintain the constraints. This might result in apparently random behavior.

Nevertheless, the idea behind constraints -automatic propagation of changes- is used in many GUI libraries to maintain layout [26] . Widget properties like *docking* or *anchoring* use predefined, manually encoded, observers to determine the float of widgets. For instance to keep a status bar at the bottom of a window regardless the size of the window.

6.2 Event based interaction

In modern window based applications events are the most common approach to define interaction between the user interface and the back-end. Events enable a *direct manipulation* [28] or *mode-free* [21] style of interaction; at any time the user can choose between many actions to undertake, and he will receive instantaneous feedback from the application.

An *event-handler* links to a method which is invoked based on user input (such as pressing a keyboard or mouse button) or at specific state changes in the system. The cause of an event is often called the *action.* The utilities provided by the operating system to enable an event-based system are described in section 3.1. At first sight events seem to be very similar to callbacks in the MVC model, but callbacks in MVC are invoked by the model, while events usually are invoked by the interface, due to receiving a message from the operating system (Figure 3.2).

Usually events are defined using a GPL. Events can be used to manipulate both the model and the interface, and can be invoked by both. To be able to manipulate the user interface it is essential that it can be approached gracefully using the GPL: Names should not be confusing and the API to access and manipulate widgets should be uniform. Since events have access to both model and view, one of the dangers of using events is that the model, view and controller become tangled, since manipulating the interface and the model is often done in the same controller [15]. Therefore, it is good practice to keep event-handlers small. A event handler should just invoke a method in the back-end and invalidate or update some widgets in the GUI. Many visual designers support attaching methods to events. In several languages multiple methods can be attach to a single event.

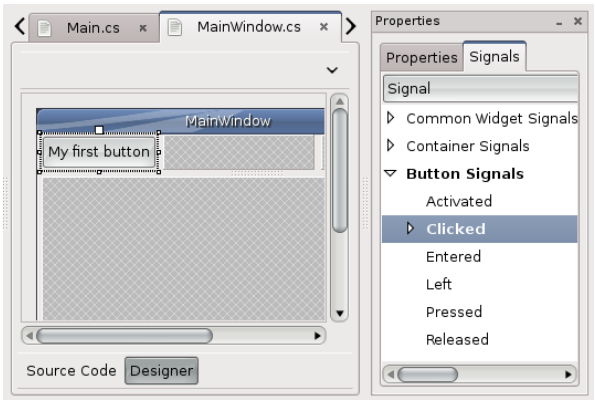

Figure 6.3: Event integration in the MonoDevelop interface builder. Clicking on the event name will generate a method stub and show the source editor.

By just using events the mutual dependency problem is not solved, and observer patterns still need to be encoded by hand. However, not solving those issues actually might have helped to gain the wide popularity events currently have: although more verbose, events are conceptually more simple than constraint based systems. Furthermore, events are very powerful since the full expressive power of the GPL can be used. And last but not least, they are well integrated in interface builders (Figure 6.3), partially due to the fact that events are easy to separate from the layout since they can be specified in separate files.

6.3 Interaction in internet applications

The essence of the MVC pattern still used today is dividing the presentation (view) from the model. It forms the rationale behind many (Ajax) Frameworks. However, coupling the applications back-end to the view is far more complex in web-applications, since the user interface and the model are a (physically) separate locations. The fact that a web-application exists in at least two places is the most fundamental difference between desktop and web applications.

For reasons of performance -a single server usually serves a vast amount of requests- it is undesirable for the server to store the stack (representing the current state of the program) between every request for every user. This causes state-management (e.g. storing information such as which user is logged in, which object is being manipulated, which sub-menu is currently selected) in web applications to differ drastically from state-management in local running applications. Since no state is stored on the stack, requests (a result of an action of a user) need to provide information about the state of the program. Therefore solving the interaction problem is directly affected by the state management problem.

6.3.1 *REpresentational State Transfer*

REST [4] has been recognized widely as the most appropriate interaction style for web applications. REST enables building very scalable web-applications since the server serves requests stateless. For this reason any server can handle any request. Inserting caches between different tiers forms no problem. The statelessness of the server is achieved by storing the state in the pages. This causes that a web application can be seen as a state-machine were the page indicates the current state. On every request the server moves to the next state, which is solely based on the data available in the request. This requires all state information to be encoded in the request (in contrast to using sessions). The key principle behind REST is that a server publishes resources, and that those resource should identifiable with an (unchanging) URI. For example, when submitting some changing to a record, the server should not remember which record the user currently was editing. Instead, such information needs to be encoded by the client in the URI. The constant '1843' identifies the current customer in the next example:

```
PUT  http://example.com/customer/1843/edit/name='Fielding'
```

It is important to notice that RESTful applications are not stateless, but rather that state is published as an resource, which can be retrieved using URIs. When using REST, one might pay extra attention to security issues. For example by changing '1843' in the URL to another constant, the user might be editing another customer, to which he should not have access. Since the server will not store session information a solution might be to send the credentials of the user along with every request to avoid *insecure direct object referencing [43].*

6.3.2 *Conversations*

Another very common approach to store state in web applications is by using conversations or server side sessions. Conversations are usually implemented as a key value table, in which variables can stored. Each client uses one such table. Since such a table receives an unique identifier, only the ID of the session needs to be send from server to client and vice versa to reproduce the state.

An advantage of this approach is that the state itself is not send along the wire, which improves the security, although one might want to build in extra check to prevent session hijacking. Sessions can be stored anywhere on the server, in memory, in a database or by using the file system. Since the state is stored at the server, the usage of sessions negates some of the scalability properties provided by REST. Furthermore the server needs a policy for life time management for sessions, since it might be unclear to the server whether a user is in-active or stopped browsing. An example request might be:

```
PUT  http://example.com/submit/name='Fielding'&session=h312bv2
```

Note that in contrast to the the previous example, the customer being edited is omitted, since this information might be stored in the state by the server. The state-table (session) to be used is indicated by the session variable.

6.3.3 *Advanced interaction enabled by JavaScript*

The interaction patterns offered by HTML are very limited. Originally, in fact only two events were implicitly supported: Clicking an *anchor* and submitting a *form.* To change the state of a page, even for small changes like adjusting a color, a new page needs to be fetched from the web-server. This introduces an unnecessary burden for both the web-server and the browser, as for every change a complete page needs to be transmitted over the (probably not too fast) internet connection.

To overcome this problem client side scripting was introduced, enabling dynamic behavior in the browser. Since 1995 Netscape Navigator ships with JavaScript. Like CSS, multiple web-browser vendors provided different implementations of JavaScript, resulting in the existence of JavaScript, ECMAScript [57], JScript and several less common dialects. The differences are not as severe as in CSS and create no limitations in practice .

JavaScript is used as scripting language to manipulate documents, such as hiding or showing objects, enabling time-dependent behavior, submit forms or change styles dynamically. JavaScript is designed to be easy to learn by inexperienced programmers. The introduction of JavaScript enables the web developer to create pages with a *rich user-experience*.

JavaScript/ HTML 4 introduced the possibility to define events in HTML pages. The way to define event-handlers in JavaScript/ HTML is very similar to the approach taken by general-purpose languages. HTML Elements publish the events displayed in listing 6.1, to which JavaScript routines can be attached [58]. Those events are a subset of the events supported by most widget libraries.

```
onclick
ondblclick
onmousedown
onmousemove
onmouseout
onmouseover
onmouseup
```

```
onload
onunload
onabort
onkeydown
onkeypress
onkeyup
onblur
```

```
onchange
onfocus
onreset
onselect
onsubmit
```

Listing 6.1: Events supported by JavaScript

Until the introduction of AJAX, JavaScript was only used in small proportions. Since the introduction Ajax (and faster JavaScript engines), JavaScript has become the backbone of many Ajax frameworks. Some of the hard-coded constraints used in many widget libraries (such as alignment) are already supported by CSS. Other constraints can be expressed using JavaScript,

although this is not trivial since reliable coordinate based positioning is hard to achieve in HTML documents.

Other means of introducing client side logic include ActiveX Objects, Java Applets and Flash/ Shockwave objects. Those act as an applications inside a web page and provide minor interaction with other parts of the page (in contrast to JavaScript) and are therefore left outside this research. Last but not least, those solutions are not general applicable in the sense that they require browser plug-ins.

6.4 What is next

This chapter introduced the problem of synchronizing the state between the model and the interface, and two common approaches to solve this problem. We determined that solving this problem in web applications is harder than solving this problem for common desktop applications, since at least two tiers are involved and it is undesirable that the server stores the complete state in memory. State management in Ajax applications is even more complex, as stated in chapter 11.

Before the introduction of JavaScript, HTML offered only very primitive support for interaction. JavaScript allows to be define events in a similar way as in general-purpose languages, although the domain lacks interface builders which ease the specification by generating and attaching stubs.

chapter 7 UI development recommendations

When developing new technologies, libraries or languages that aid the development of Ajax applications, one can learn from experiences in history. This chapter represents a list of recommendations based on results that has proven, and not proven to work in work accomplished in the last decades and as described in the previous chapters. The recommendations are categorized. Each recommendation is accompanied with references to chapters which demonstrate the statement. Applying those recommendations should result in a more rapid development process and better maintainability of (desktop like) web applications. Those recommendations will guide the design of the WebDSLx abstractions described in the next chapters.

7.1 General recommendations

Provide clear abstractions. This recommendation sounds almost trivial, but it is important that a developer is able to understand the transformations a tool is performing. Violating this constraint was one of the reasons UIMS did not work out. [5.1.2]

Interact at different levels of abstraction. When high level abstractions are provided, it is important that there is an interface by which the developer can influence the results of a transformation. For example when complete CRUD-pages are generated, it is important that those pages can be tuned to improve the experience of the end user. This tuning should not be done inside the generated code. User interface management systems neglected this interface, in contrast to XUL which uses the XBL standard. [Error: Reference source not found, 5.1.2]

Interface to HTML. Allowing the developer to specify tiny parts of HTML avoids the framework being required to support uncommon HTML constructions, and allows the developer, for

example, to embed a YouTube movie. Furthermore WYSIWYG HTML editors can be used to design those tiny parts. [3.4, 4.3, 5.3]

Book-markable URLs. Consistent, book-markable URLs are important, as users are able to send references of certain pages to each other. Especially when programming single page interfaces, this aspect is quickly forgotten. [12.3]

Platform independence. Being able to deploy to many different platforms, allows the tools to be disclosed to a bigger audience. [5.2.1]

Abstract over request details. A developer should have no need to specify how requests should be sent, how parameters are serialized etcetera. Such low-level implementation details should be handled automatically. [6.3]

7.2 Concerning interface composition

Hierarchical composition. Hierarchical structured definitions to represent interfaces have proven to work, as interface have a hierarchical composition. [4, 5.2.1]

No GPL interfaces. Specifying an interface using a bunch of statements written in a general-purpose language is a bad idea; it complicates analyses, refactoring and portability. Furthermore, it lacks the natural hierarchical structure of the interface one tries to implement. [3.2.4, Error: Reference source not found]

Allow textual interface specification. Interface definitions should be editable by plain text editors, just as any programming language. This allows to use sophisticated tools already available, such as versioning systems. [5.2]

Semantic division in the user interface. Ajax requires to model the interface in small parts. Being able to separate parts of the interface based on the semantic context leads to a natural reuse of parts in multiple pages. For example: a small interface which displays a user (using a thumbnail, his name, title and a link to its profile) might be reused in multiple pages. Templates can be used to achieve this. [2.5, 4.3].

Provide an interface builder. Interface builders aimed at Ajax applications are rare (some exceptions: Atlas [59], Backbase [60] and Mendix [56]. Interface builders have proven to be very successful as it eases the development of interfaces due to their WYSIWYG nature. In the near feature we will definitely encounter more (online) visual designers to build Ajax applications. [5.2]

Support preview functionality. Enabling a developer to preview the pages without the need to compile and redeploy allows for rapid interface development. [5.2, 5.3]

7.3 On interaction

Separate interaction and composition. To achieve the recommendations concerning composition, interaction should not interfere with the composition of the user interface. By separating

composition and interaction preview functionality and interface builders are easier to achieve. [6]

The behavior of an application should be event driven. It provides a simple and intuitive model which has proven to work for desktop applications. [6.2]

Use simple constraints. Do not provide a complex constraint systems, it introduces too much 'magic' and is to hard to debug. Providing (really) simple size- or data-binding constraints on the other hand might really work well. [5.1.2, 6.1]

7.4 Regarding widgets

Support the universal widgets. Support the uniform set of widgets available in most interface specification tools, developers will be disappointed when omitting some of them. [5.2, Error: Reference source not found]

Widgets should have a uniform interface It should be clear how widgets can be configured or manipulated using a textual, visual or programmatic approach. Furthermore, this allows for third-party widgets libraries to be plugged-in easily. [3.2]

Sub classing. Provide a sub classing system, by which developers can define and reuse common widget configurations. For example a button which has a 'close' icon and always hides its parent window. [3.1]

Combine widgets. Provide abstractions to easily create new widgets by composing existing ones, this encourages the reuse of reoccuring widget compositions, such as a dropdown-box and a button inside a frame. [3.1, 4.3]

7.5 To ease styling

Styling is an orthogonal issue. Separate styling from interface and interaction definitions. Styling should be applied globally. This enables style to be changed without the need to reinspect every interface definition. [2.4]

Abstract over browser differences. Designers put too much effort in creating stylesheets compatible with different mainstream browsers, which in fact results in writing the same styling multiple times. [5.3]

Customizable widgets. The style of widgets should be customizable, to be able to attach more eye-candy than the original developers of the widget intended. [12.3]

7.6 To improve performance

Detect which code should be executed where. Automatically detect which code should be executed at the client or at the server. At least provide a construction for the developer to influence this. [12.3]

Avoid server state. Avoid in memory state management (conversations) as much as possible. This has a positive effect on performance and allows for scalability. Reduced response times also improve the user experience. Furthermore, server state requires session management and garbage collection of sessions that were closed without informing the server. This often results in annoying time-outs when the user has performed no activity for a while. [6.3]

chapter 8 WebDSL

WebDSL is a Domain Specific Language initiated by Eelco Visser [37] and designed to develop dynamic web applications. WebDSL abstracts from many of the details web developers usually have to deal with, such as state management, data validation and database management. Application models are compiled into complete web applications, which has no stubs that need to be hand written. Since WebDSL generates ready-to-use web applications, there should be no strict need for developers to learn any of the complex languages usually used to develop web applications, such as XHTML, CSS, JavaScript and a server-side general purpose language.

Languages used to develop web applications are often loosely coupled, for which reason the validity of applications cannot be checked statically. This results is many errors which cannot be detected until such application is deployed and tested. WebDSL however provides a static type checker, which detects errors compile time. Therefore one can claim that WebDSL applications should never fail as a result of errors in the generated code after passing the type checker. A model that does not contain errors should result in an application that does not show erroneous behavior.

The WebDSL compiler is build using the *syntax definition formalism* (SDF) [36] and the *Stratego/XT* toolset [35]. The compiler parses the model using a SGLR (Scannerless Generalized LR) parser. This results in an *abstract syntax tree* (AST) which is transformed into an AST in the intended target language (Java in the case of WebDSL) and then pretty-printed to Java source files.

Since WebDSL is based on the syntax definition formalism, it is able to intertwine multiple languages in one model. WebDSL consists of multiple sub-languages, such as a language to define web pages, data models, queries (based on HQL), workflows etcetera. The compiler verifies that statements and expressions defined in different languages are embedded correctly in each other.

8.1 The structure of a WebDSL application

WebDSL applications are organized in **.app* files. An application is organized in different *modules.* In this way the application can be distributed over several files, and can modules be reused. A module consists of a set of definitions. A definition defines either a page, template, function or entity. Those definitions will be examined in detail in the next sections. A very simple application is shown in listing 8.1.

```
/* Helloworld.app */
application HelloWorld
imports MyFirstImport
section pages
define page home () {
  "hello " world()
}
```

```
/* MyFirstImport.app */
module MyFirstImport
section templates
define world() {
  "world"
}
```

Listing 8.1: Simple WebDSL Hello World application consisting of two modules

As mentioned WebDSL models consist of several sub-languages. This chapter only introduces the user interface language, since this is the only one that is directly influenced by the Ajax abstractions proposed in this thesis.

8.2 Page and template definitions

This section elaborates on *page* and *template* definitions. Page and template definitions define the user interface and the interaction patterns of the application. Both have the same structure. The semantics however are slightly different. First of all, pages can be referred to by using URLs. Secondly, a user can only see a single page at a time. In WebDSL, interaction with the user is achieved by navigating from one page to another. A page can embed several templates, and templates can recursively embed templates as well. A templates functions as a container that groups user interface *elements.* Templates enable the reuse of interface definitions in WebDSL applications. Both pages and templates can take a list of *arguments.* Overloading is allowed for templates.

It is allowed to redefine a certain template inside a page. This allows to reuse a template while redefining a small subpart of it. This mechanic is shown in listing 8.2. In this example both the *home* and the *publication* page use the *main* template. This main template invokes a *body* template, which might be overridden. Note that this construction introduces dynamic scope.

```
define page home() {
  main()
  define body() {
      "this is the home page"
} }

define page publication(p : Publication) {
  main()
  define body() {
      "Publication: " output(p)
```

```
  }
}

define main() {
  sidebar() body() manageMenu()
}

define body() { // defaults to empty }
define manageMenu() { "menu" }
define sidebar() {"sidebar"}
```

Listing 8.2: Local redefines in WebDSL

8.3 User interface elements

The contents of templates is defined by calling user interface elements. Roughly four different kinds of elements exist. *Control-flow* elements (such as loops and variable declarations), *user interface* elements, *actions* and *local template definitions*. User interface elements are either built-in or defined in the model as templates. The built-in elements (which are in fact templates provided by the compiler) closely resemble to the user interface tags of HTML. In general an interface element (or *template call*) consists of three parts. An identifier, a set of parameters and a set of children which will be wrapped by the element referred to. There are many built-in user interface elements available in WebDSL. The next section demonstrates a few.

8.3.1 Common interface elements

Text. To output text, quoting it between "" is enough. The common escaping rules apply.

To display the value of an expression one can use the generic *output(Expression)* element.

Header generates a text header. The children of the header are used as the caption.

Image(string) displays an image.

Table defines a tabular structure. Tables are useful to display a dataset or to achieve a column-based layout. Any child element of the table that is not a row will automatically be wrapped by one; wrapping arbitrary elements by a table will order them below each other.

Section result in a HTML *span* element. Sections can be used to group multiple items inside a single span. This might be useful to assign styling.

List and *listitem* are used to display lists. By default each item displays a bullet and all items are displayed below each other.

Navigate(target) creates a link (<*a href*>). The argument can either be a string containing an URL or a call to an existing page, for example:

```
navigate(home()){”click me!”}.
```

8.3.2 Forms

Form are used to group user inputs that should be submitted simultaneously to the server. The submit of a form can be invoked using the *action* element. A simple form is shown in listing 8.3.

Inputs for an arbitrary variable or field can be created using the *input* element. WebDSL rewrites every *input* and *output* element to a more specialized input or output element, based on the type of the provided expression. A developer can override the *output* of a certain type. *Action(string, actioncall)* shows a button which invokes an action (yes, WebDSL has two different notions of *action)* defined in the same page or template.

```
define editFolder(f: Folder) {
  form { group("editing folder "+f.name) {
      groupitem {
        label("name ") {input(f.name) }
      }
      groupitem {
        label("description") {input(f.description) }
      }
      groupitem {
        action("remove", remove())
        action("save", save())
  } } }
  action save() {
    f.save();
    return showfoldercontents(f);
  }
  action remove() {
    f.delete();
    return showfolderlist();
} }
```

Listing 8.3: Simple form using actions

8.3.3 Control flow elements

WebDSL has a small number of elements that influence the rendering order of a page. Control flow elements can be used to include or skip a part of the user interface, or to iterate over a collection. Control flow elements can be used inside action definitions as well. Variable declarations can be used inside a page to store certain information. If an action is declared inside the scope were the variable was declared, it is not necessary to pass the variable along as parameter. Since actions are defined inside a page or template definition, there is no need to explicitly pass the page or template arguments.

WebDSL provides an *if-then-else* construction very similar to the ones provided by general purpose languages such as Java. The *for* statement is a powerful construction in WebDSL. It is able to fetch objects from the data store automatically, and iterate over them. It supports filters, ordering and separators. Listing 8.4 defines the complete syntax of the *for* statement.

```
For := "for" "(" Id ":" Sort ("in" Exp)? Filter? ")"
  "{" element* "}" ( "separated-by" "{" element* "}" )?
Filter   := ("where" Exp) ? ("order" "by" OrderExp)? (Limit | Offset)?
OrderExp := Exp ("asc" | "desc")?
Limit    := "limit" Exp Offset?
Offset   := "offset" Exp?
```

Listing 8.4: Syntax of the WebDSL for construction

8.4 Actions

Beside elements, WebDSL pages and templates can also contain *actions.* The purpose of an action is to calculate the next state of the application. In order to do so, it might process a number of statements with the purpose to determine which page is the next one to display to the user. Actions are a bit similar to functions in general purpose languages. Actions are allowed to manipulate data in the database, and can call global or entity functions. Furthermore the most common control-flow structures are supported, such as if-then-else, the for loop and the while loop. In the end, an action is required to return a page call using the *return* keyword. Omitting the return results in a page refresh. Listing 8.5 shows a simple save action.

```
action savenewdocument(name: String, description: String) {
  var d: Document := Document{
    name := name,
    description := description,
  };
  d.save();
  return outliner(d) ;
}
```

Listing 8.5: Simple save action

8.5 An evaluation of WebDSL

WebDSL has no built-in support for Ajax based applications. However when looking into the recommendations stated in chapter 7, one can argue that this language is a good place to start. First of all, WebDSL has been designed from a developers point of view. Both the abstractions and syntax feel naturally and are user friendly. WebDSL allows the developer to avoid writing much of the boilerplate code, which is usually required in other programming environments when writing web applications. One can argue that WebDSL does *provide clear abstractions*.

Furthermore the language allows to define applications using different levels of abstractions. For example, for a quick start, the language can generate CRUD pages based on the data model. Those pages are defined in the (core) language itself, for which reason the developer can easily override the generated pages by ones self defined. The same principle holds for the Access Control [9] and WebWorkFlow [10] sub languages; they translate into core WebDSL definitions. Therefore the developer is able to *interact at different level of abstractions*. An syntactic *embedding of HTML* has also been added to the WebDSL compiler.

The WebDSL compiler translates *page calls* into nice *book-markable URLs*, for example *http://webdsl.org/webdslorg/selectpage/Manual/ActionCode*. Furthermore the compiler is designed to support compilation to *multiple platforms.* Although at the time of writing only the Java back-end supports all features of the language, the Platform Independent Language [11] is under investigation to solve this issue. In addition, as one can see in the examples throughout this chapter, WebDSL *abstracts over request details*, such as serialization, character encodings, escaping, state management, security etcetera. For this reason much of the boilerplate code a developer usually has to write can be omitted.

WebDSL does allow the user interface to be *composed hierarchically*. Those interface definitions are *accessible by any text editor*, and can be managed using third party tools such as subversion. The language has a notion of templates, which allows to divide the user interface in separate parts, and reuse those parts in multiple pages. Those parts can be parameterized using objects from the domain model. This allows for *semantic division* of the user interface. The compiler allows to embed Java code for native function calls. So *it has a GPL interface*, although the interface wrappers are separated from the other definitions.

Interaction and composition are separated inside the page definitions. Actions are definitions inside a page, to which certain elements can refer. WebDSL applications are *event driven* to a limited extend, as is the case with HTML (section 6.3.3). Clicking a link or submitting a form fires an event, causing the application to determine the next state. WebDSL does provide *simple constraints*, both in the user interface as on the data model. In the data model constraints such as inverse relations can be specified, and in the user interface input elements are bound automatically to the variables they represent. For example defining an input for an object relation, results in the relations (and its inverse) being modified automatically when the corresponding form is submitted, without writing any code. Currently WebDSL provides no visual *interface builder* and it has no *page preview functionality*.

Since WebDSL has no widget system, it does not support any of the related recommendations. It has however some styling support, for which reason *styling can be defined separately* from the user interface definitions. The styling extension of WebDSL is described in [14]. Finally, WebDSL *avoids server state* in the form of conversations. For this reason WebDSL applications suffer neither from expired session nor from an increased memory footprint when more users connect to the server.

8.6 What is next

The WebDSL language described in this chapter is the starting point of the WebDSLx extension. Although WebDSL has no support for Ajax or delta updates, it provides nice page and template models, to which Ajax support can be added. WebDSL implements many of the recommendations mentioned in chapter 7, but not all of them. It lacks widget abstractions and some of the recommendations need to be reevaluated after implementing a new state model. This will be done in chapter 16.

chapter 9 Desired abstractions in asynchronous interaction

Ajax based applications provide a significant better user experience compared to classic web applications. Faster responses and more advanced interaction patterns provide a *rich* user experience, often at the same level as classical desktop applications. In section 2.5 we explained how delta updates enable to transfer data in the background and modify pages on the fly. WebDSL does not support this interaction model since it is based on the single state paradigm where the user is sent from one page to the other (Section 6.3).

In this section we investigate how we can embed the new interaction pattern into the WebDSL language, with a minimum of changes in the WebDSL abstractions. All the nice advantages of WebDSL (Section 8.5) should remain, such as hiding state management issues etcetera. This chapter investigates which primitive operations need to be added to the WebDSL model in order to achieve an asynchronous user interface.

9.1 A note on asynchronous interaction

In the Ajax acronym, *asynchronous* refers to the fact that request are send to the server in the background, and that those requests are executed asynchronously; the user stays on the same page and can continue browsing and will experience no click-and-wait pattern. Furthermore responses of the server can be received in different order than that the requests were sent. *Asynchronous interaction* however, does not purely refer to the request mechanism itself, but rather to the new state model introduced by this mechanism; the actual state of the application is no longer represented by the page the user sees (as described in section 11.2), since pages are not submitted anymore as a whole, and might have been modified by delta updates. Asynchronous interaction refers to the interaction model of asynchronously updating the page using delta updates. The complications concerning state management caused by this pattern are described in chapter 11.

9.2 What delta updates really are

Delta updates form one of the important advantages of Ajax. Delta updates can be used to replace an arbitrary part of the user interface. When inspecting a HTML page, for example by using a debugger (figure 9.1), it becomes clear that plain HTML documents are parsed into tree structures. For this reason there is no need to modify the plain HTML source when using delta updates, rather, manipulation the document tree (called the Document Object Model or DOM thee) will suffice. For this reason, to be able to perform delta updates, arbitrary nodes of the DOM tree need to be manipulatable. In general, a limited set of transformations can be applied to a tree node:

- Insert a new node
- Remove a node
- Modify a node or one of its attributes
- Move a node
- Replace a node

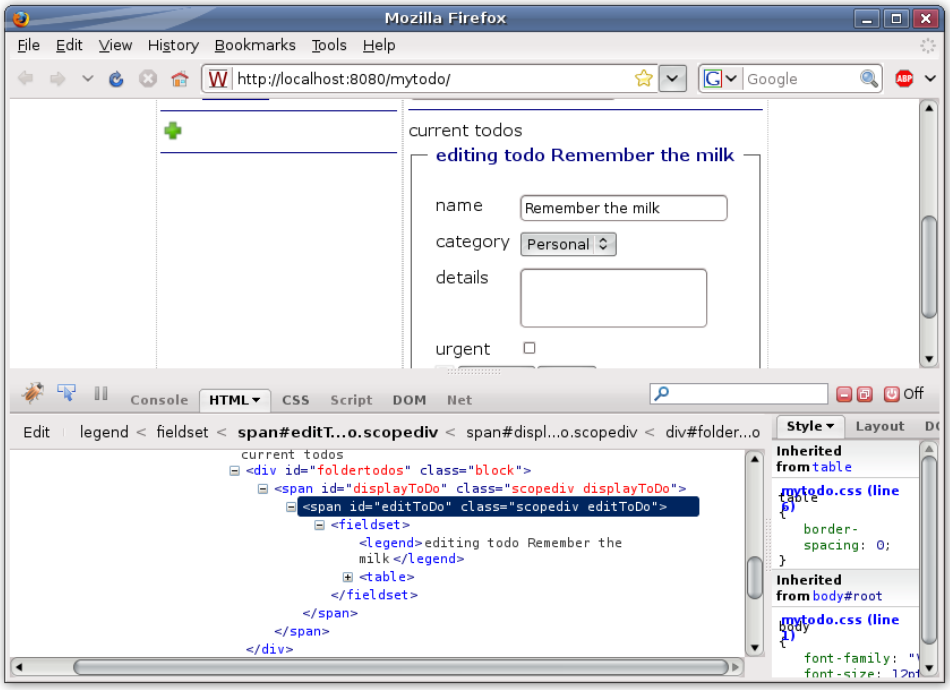

Figure 9.1: Inspecting the DOM tree (using Firebug) of the MyToDo application (chapter 14). The contents of the template displayToDo node is replaced by the editToDo template.

Those transformations are the real primitives of manipulating HTML documents, and delta updates will result in such transformations. In general a delta update needs to specify two things, a transformation (with some arguments) and a node to apply the transformation to. The latter one is called the *location* or *target* of the transformation. Transformation like inserting or replacing a node require a value to specify the new look of a node. The mentioned tree transformations are at a low level of abstraction, and far more powerful than most applications require. When investigating the structure of desktop applications, one can discover that the user interface is often manipulated at another level of abstractions.

9.3 Widget grouping

Upon user input most applications will show a new tab page, popup dialog or form, or enable or disable a part of the GUI. Generally speaking, interface manipulation happens at the level of so called *panels*[3]; a collection of (input) widgets that are related to the same subject.

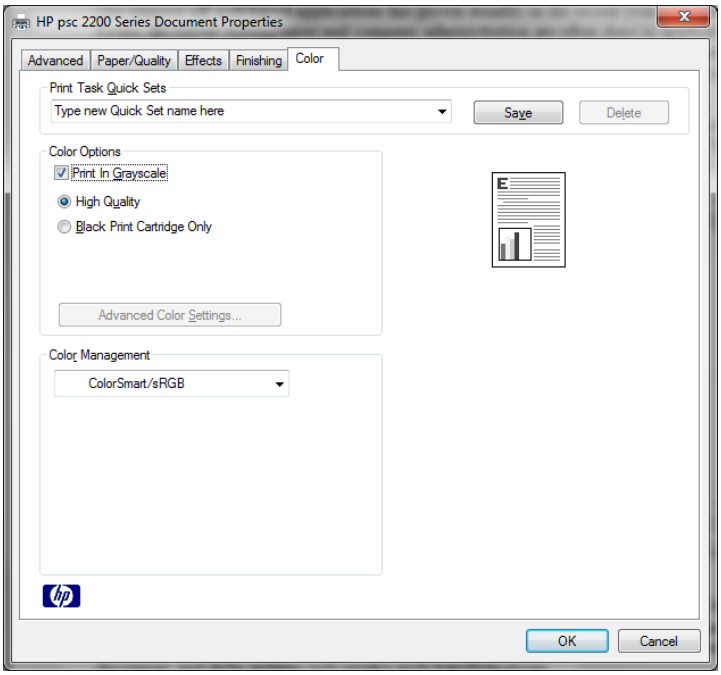

Figure 9.2: Example of a dialog with semantically grouped widgets

For example, in a print dialog (Figure 9.2) pressing the *Color* tab will result in a screen (1) with multiple widgets related to the color settings. Checking *Print in Grayscale* will enable the radio buttons (2) and disable the *Advanced Color Settings* (3) button. This adds up to a total of three interface transformations, but those transformations resulted in many more widgets to be manipulated. So by grouping *widgets* related to an certain *subject* we can reduce the number of user interface manipulations required to achieve the desired transformation, since, for example, it allows to hide multiple widgets simultaneously. By grouping objects we can reach a higher and more natural level of abstraction. The semantic groups are clearly communicated to the user in the example by drawing borders around the groups. Last but not least, by allowing widget groups to nest, trees of widgets can be achieved, which matches the natural structure of interfaces (Section 4.1). The division of labor applied in user interfaces by using panels can also applied in static web applications, were template fulfill this role (Section 4.3). Templates are responsible for rendering a specific part of a page, and usually relate to a specific domain object.

For those reasons, templates are a natural subject for Ajax operations. Instead of manipulating a DOM node, manipulating templates as a whole provides a better level of abstraction. WebDSL supports templates, which are parameterizable reusable interface definitions. As such, *template calls* (an invocation of a template with actual parameters) forms the *value* of most Ajax operations provided by WebDSLx.

3 We consider tabpages, forms, dialogs etcetera as special kind of panel

9.4 Placeholders

To identify on which node an Ajax operation should be applied, XPath expressions [61] could be used. But although XPath expressions are very powerful, they should not be used inside the WebDSL model. When writing an XPath expression, the developer needs to know the internal structure of pages generated by the web server, which is exactly the kind of details the models abstract from. A developer should not need to know the internal structure of the generated interface. For that reason WebDSLx provides another approach by using placeholders. Placeholders are used by many Ajax frameworks (Section 4.4), to identify the location in the DOM tree to which certain behavior should be added. It is merely an global identifier to which an Ajax action can refer.

9.5 Events

Event based interaction is the most applied and most successful technique to define interaction in graphical user interfaces (Section 6.2). HTML browser are event based, but the events itself are hidden in the HTML standard. Generally speaking, two events can be defined on a HTML page without JavaScript: clicking an anchor (*a href)* or submitting a form. For both action, one cannot define a callback, but only a new location to navigate to. Since the introduction of JavaScript, events can be specified in HTML, by assigning a code snippet to an event attribute of an interface element.

By defining events on HTML elements, delta updates can be requested from the server, even when the user does not navigate to another page or submits a form. This mechanism allows to manipulate the user interface without replacing the whole page by a new one, as is done in classic web applications. For this reason the WebDSLx extension needs to introduce event attributes, which can be used to invoke server-side actions without the need to submit the current page or click a *navigate* element.

9.6 Actions

An event based approach requires some changes in the semantics of action definitions. The purpose of an action is no longer solely to compute the new state and next page the user should see. Rather, an action might decide to send some delta updates back to the client. So the set of statements supported in actions should be extended by statements which define delta updates. Those statements should be based on the earlier mentioned tree transformations. The next chapter provides a concrete model for the in this chapter suggested abstractions.

chapter 10 Overview of the WebDSLx primitives

The previous chapter describes which abstractions are required to support the asynchronous interaction pattern which is typical for Ajax applications. This chapter describes how those abstractions are implemented by the WebDSLx extension. Listing 10.1 shows the source of a small template which uses some of the WebDSLx operations explained in this chapter. It defines a template of the MyTodo task management application (described in chapter 14). The responsibility of this template is to show a *ToDo* item to the user. When the ToDo item is not finished yet, a checkbox and link will be shown (figure 10.1). Clicking either of them will set the ToDo item to finished, and re-renders the template in place. Furthermore there is an *edit* button, which will show the *editNote* template in place.

```
define displayTodo(n: Todo) {
  form {
    group(n.name) {
      if(n.urgent) {
        image("/images/urgent.png")
      }
      if (n.finished) {
        image("/images/finished.png")
      }
      output(n.details)
      if (n.finished == false)  {
        var b: Bool := false
        input(b)[onclick := finish()]
        navigate[onclick:= finish()] {"finished"}
      }
      image("/images/edit.png")[onclick:=
        action { replace (displayTodo , editTodo(n));}
      ]
  } }
```

```
  action finish() {
    n.finished := true;
    n.save();
    replace (this , displayTodo(n));
} }
```

Listing 10.1: Small example of a WebDSLx based template

Figure 10.1: Example instances of the displayToDo *template*

10.1 WebDSLx functions for delta updates

```
    replace(target, templatecall)
    append (target, templatecall)
    clear  (target)

    restyle (target, String)
    relocate(pagecall)

    visibility(target, visibilitymodifier)
    refresh()
```

Listing 10.2: Action statements provided by WebDSLx

Section 9.2 describes that delta updates require the possibility to manipulate the DOM Tree of a page. WebDSLx provides several operations to modify the structure of the DOM Tree. All WebDSLx operations can be invoked from any action in a WebDSL model.

The *replace* operation replaces the contents of a placeholder by loading another template. Replace takes two arguments, a target location and the template to load. The latter might also be a template call with formal arguments. An example invocation of the *replace* action is shown in listing 10.1. The *append* operation has the same signature and behavior as the *replace* action, except that it does not remove the current contents of the placeholder, but rather appends the new template at the end of the placeholder, which is useful for prolonging lists. The *clear* operation only requires a target location, and removes all content from the placeholder.

Relocate can be used to navigate the browser to new location, and its behavior is pretty comparable to the native WebDSL *return* statement. *Refresh* performs almost the same action, it refreshes the page currently displayed. The *restyle* operation can be used to change the style of an object. Like the other operations it requires a target and as second parameter it takes a string value which should be the classname of a style. Styling in WebDSL is beyond the scope of this thesis but more information can be found in the master thesis of J. Holwerda [14]. Finally *visibility* changes the visibility of its target. The second argument is an identifier from the set *show*, *hide*, *toggle*. *Hide* does not clear a placeholder, so the original elements can be shown again (using *show*) without requiring the server to re-transfer the contents. *Toggle* shows the target if hidden and vice versa.

Note that although there are only a few operations, this is not the minimal set of operations required. For example the *replace* operation could be expressed as a combination of *clear* and *append*. However, *replace* is a more appealing abstraction for the developer.

In order to reduce the number operations required to achieve the desired user interface transformation, elements need to be grouped into templates (Section 9.3). WebDSL provides the *template* notion to achieve this. WebDSLx extends this notion with more advanced constructions to structure the interface, as described in chapter 13. The delta operations provided do not allow to apply arbitrary changes to arbitrary nodes. In practice (chapters 14 and 15) there seems to be no need for such low level operations. Furthermore the provided operations abstract from the fact that one is manipulating a DOM tree and therefore will suffice for any (asynchronous) user interface interaction system.

A single WebDSLx action might result in multiple delta updates. Those updates are collected and encoded as a JSON [40] array by the server. An example of such a response is shown in listing 10.3. Note that for the *replace* operation, the template call is immediately evaluated an the resulting HTML is send to the client. This saves another request from client to server.

```
[{
   action: "replace",
   id: "this",
   value : "<span id=\"editToDo\" class=\"scopediv editToDo\">
     <fieldset><legend>editing note Fetch a meal</legend>
       <!-- bunch of HTML -->
     </fieldset></span>"
 },
 {
   action: "visibility",
   id : "sidebar",
   value : "hide"
 }]
```

Listing 10.3: Example of some delta updates generated by the server

10.2 Locations

Most delta operations require a target location to be applied to. WebDSLx provide four ways to identify a location in the page. First, there is the *placeholder* construction. A placeholder is a set of elements with an identifier. The elements act as the initial contents of the placeholder (Listing 10.4).

```
placeholder leftbar { /* elements go in here */ }
table[id := maintable] { /* ... */ }
```

Listing 10.4: Example of two placeholders

Furthermore WebDSLx accepts *this* as value for a location, which always refers to the template which contains the object that fired the *event* (see next section) responsible for the current action. This is a useful feature since many interface manipulations need to be applied *in-place.* Similar to the *this* functionality is using the formal name of a template as target, which replaces the closest occurrence of the template. For this reason every template generates a placeholder with the same name. Those constructions are demonstrated in listing 10.1.

Finally, almost any user interface object can be turned into an placeholder by setting the *id* attribute. This is especially useful when using the *restyle* or *visibility* operations, which might be useful for a single object. For example drop-down boxes can be extended dynamically by assigning an id attribute and using the *append* operation.

Identifiers can be used multiple times throughout the application definition. In order to apply a transformation to the proper target, a small algorithm is used. This algorithm allows templates to be recursively nested, while target identifiers still refer to the naturally intended subject. The target of a delta update is found by applying the following algorithm in the browser.

1. Starting at the element that fired the event, the DOM tree is searched upwards until an element with the proper *id* is found, or until a *scope boundary* (template) is encountered. In other words, initially the search will start inside the template that contains the fired action.
2. When the desired target is not found while searching upwards, an top down breadth first search, origination at the scope boundary, is applied to find the proper element (The reason for the breadth first is shown in figure 10.2).
3. When still nothing is found, the whole tree is searched upwards, originating at the current template. This is useful in order to be able to replace the owning template of the current template.
4. As a last resort, search the whole DOM Tree. Apparently, a section which is not structurally related to the current item needs to be replaced; such as a sidebar or status bar.

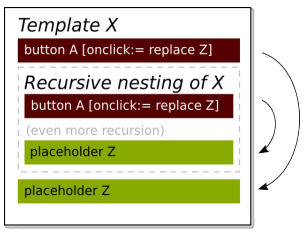

Figure 10.2: When using recursion, target identifiers should prefer placeholders generated by the same template instance

10.3 Events

To invoke an action when a HTML event is fired, for example when pressing a key, event attributes can be defined on elements. Attributes can be used to define domain-unrelated, optional arguments. In this chapter attributes are used to assign id's and events, but as we will see in chapter 13, attributes can be used for many more purposes. Attributes can be specified on calls to both user defined templates and built-in elements.

Listing 10.5 shows how actions can be hooked to events using attributes. All events published by HTML (Listing 6.1) can be used. The coupling is achieved by assigning the call to an action to the attribute with the name of the desired event. Besides assigning actioncalls, WebDSLx provides the possibility to inline anonymous *actions* and *templates* to reduce the amount of code required to execute simple events. By using in-lining the programmer is allowed to define a template anywhere where a template call is required, and an action anywhere where an actioncall is required. Inline actions and templates can be intertwined (listing 10.5). Inline actions and templates will be lifted by the compiler to (global) definitions with a generated unique name. Variables provided by the closure of the action (or template) will be passed to the new definition as argument (listing 10.6).

```
template hoverLink(f: Folder){
  actionLink(f.name)[
    onmouseover := action { replace(folderdetails,
      template { block {output(f.description) }}
    );}
  ]
}
```

Listing 10.5: Examples of event attributes and in-lining

```
template hoverLink(f: Folder) {
  actionLink(f.name)[
    onmouseover := inline_action1()
  ]
  action inline_action1() {
    replace(folderdetails, inline_template1(f));
  }
}

template inline_template1(f1: Folder) {
  block {output(f1.description) }
}
```

Listing 10.6: Result of applying lifting to listing 10.5

10.4 What is next

In the next chapter we will discuss the complications introduced by the delta operations. The next chapter will explain how Ajax influences the state of web applications in general, and how WebDSLx deals with those issues.

chapter 11 State management in WebDSL(x)

In most web applications the semantic behavior of an application is the responsibility of the server. The server validates and stores input and presents domain data to the user. Since a server needs to be able to serve numerous users, storing the program stack (and thus complete state) between two requests is undesirable. For this reason variables set by one request (for example the current user) are not automatically available while serving the next request from the same client. To overcome this problem, the information that is required to serve the next request needs to be encoded in the current response and send back by the client during the next request, or it needs to be remembered somehow by the server. As a result of this, the developer of a web application needs to take care of many details and enforced to think about:

- What data eventually might be required by subsequent requests.
- Where to store data; in the database, the web-server's memory or at the client.
- How to store the data. This is another complex task since a developer needs to deal with limitations in memory usage, bandwidth and charset encodings.
- Furthermore security might complicate the task, since confidential data should not be send to the client, where it is sensitive to tampering.

In general there are two models for state management in static web applications, as described in section 6.3. The next section describes the WebDSL approach to state management, while the remainder of this chapter describes how Ajax complicates the issue, and how those complications are dealt with.

11.1 The WebDSL approach to state management

The WebDSL web server combines the strong points of REST and conversation based approaches. It simulates conversations while acting on RESTful requests, without keeping the state in memory between request. Instead of storing the state, it reproduces the state generated by the previous page. This achieved encoding in the page the state that was used to render a certain part of the page. For example when generating the HTML source for an input form, the actual arguments of the current page are encoded in hidden fields. Furthermore every object which might influence the state of the page receives an unique identifier, which is basically an integer with an incremented value for each next object. This identifier is modified by every control flow structure and every input object. Furthermore the identifier is hashed, to prevent tampering.

When the server serves an action, it parses all parameters from the request data and delegates the request to the proper page, which restores the actual arguments by which this page was called in the previous request. The server then traverses the page structure, executing all logic of the page, just as would be done in a page-render (except that this time, the elements are not rendered, although they will be re-instantiated). When a restored identifier matches the from the request received one, the server knows that the context in memory matches the state in which this page was originally rendered. When identifiers in the request do not match a rebuilt identifier, the server recognizes this as an attempt to hijack an identifier and ignores the request.

This is the essence of storing and recovering state in WebDSL. The real implementation is a bit more sophisticated, since the server can deal with cases where for example collections have been modified. In listing 11.1 the proper user is deleted from the database when, for example the second, delete button is clicked, without transferring the identifier of the selected user across the wire. The server does not need this identifier because, during the render phase, the generated name of each submit button is influenced by the actual value of the iterator.

```
page deleteUserFromGroup(group : Group) {
  form {
    'Press delete to delete a user'
    foreach(User u in group.users) {
      output (u.name)
      submit('delete', deleteuser)
      'break'
  } }
  action deleteuser() {
    group.users.remove(u);
    return deleteUserFromGroup(group);
} }
```

Listing 11.1: The variables group *and* u *will be restored automatically before executing the action* deleteuser.

The great advantage of WebDSL is that state management is handled transparently. All state management code required for listing 11.1 is generated by the compiler, proving that one can abstract over those issues, which results in a more natural application model.

11.2 State management in Ajax applications

State recording in classical web applications is fairly straightforward; since every page is completely rendered by a single request all data that influences the state is available during the rendering. For this reason state can be encoded by means of (for example) hidden fields. Security can complicate the issue, for which reason sensitive information is often stored in so called session-objects. Static web applications can be seen as state machines, where the current page indicates the current state of the application. The user can either click a link or submit a form, which invokes a state transition. Upon receiving a request the server calculates the new state, that is, a new page which will be presented to the user. As a result, the application is in a continuously in a consistent state; since the whole page is rendered using the same server state it does not display internal inconsistencies.

Ajax complicates the state-management problem since a page is built using many tiny requests. Delta updates are based on the most actual state of the server, while parts that are not affected by the updates, are in fact cached at the client. Caching always introduces additional management issues and as a result of caching it is possible for the user to perceive state inconsistencies, such as name changes which are not propagated through the whole page. In addition to the caching, by using Ajax request, the page is not submitted as whole; not all data is send back automatically to the server. Still the server needs to determine somehow the (relevant parts of the) current state while handling requests [18].

Automatically updating all parts of a page that depend on the same model changes is an instance of the constraint maintaining problem which also occur in desktop user interfaces (Section 6.1.2). Although a property models can solve this problem and thus seems to be a welcome addition to Ajax based applications, in order to keep pages consistent, implementing such a system in an Ajax environment is complicated: Events originate at the client, but the model -which should be inspected to detect violations of the constraint- resides at the server. For this reason it might be better to just carefully encode under which conditions certain parts need to be updated; too many server requests neutralize the performance advantages of Ajax. Update conditions and invocations often need to be encoded using low level constructs, although several Ajax-frameworks provide some abstractions for this kind of constraints [55].

11.2.1 Problems in reconstructing the state

The possibility to replace, remove or append an arbitrary part of a page causes an explosion of the possible states the application can be in. Where a static application can produce a fixed number of different pages (not considering differences in the actual domain objects being displayed) an Ajax application can produce an unlimited number of different pages. For this the current state cannot be identified just by using an identifier, such as the name of the page. Rather, we need to keep track of all the updates send to the client in order to be able to reconstruct the current state of the page the user sees, and perform complex calculations to determine which updates completely hide previous ones.

There are two solutions to this problem. The first one is to move the responsibility for state management from the server to the client, as described in the next section. The second solutions is to partially ignore the problem and use state localization. This approach however forces to define update constraints manually.

11.2.2 Client-side state

Besides complicating the state management problem, Ajax also enables a new state management model. In the *client side state* model the server no longer determines the state of the application, rather, JavaScript scripts in the browser execute the application flow, determining the next state, while the web server just acts as a data or RPC server. Since the application state is managed in the browser, delta updates are not received from the server, but generation by the client itself, for which reason it is more easy to keep the interface consistent. This approach is able to reduce the amount of date send to- and received from the server, since, after loading the application, only domain data needs to be send across the wire. The Google Web Toolkit is an example of this design philosophy [62].

This approach however has some downsides. Loading the application might be quite slow since the core of the application need to be transferred to the client first. Secondly the application will not work in browsers which do not support JavaScript, while server side applications can provide workarounds. Furthermore the applications depend on a browser with a fast and memory safe JavaScript interpreter, which only holds for the more recent browser generations. So the number of potential users might be restricted a little by those technical constraints. Nice URLs and indexing are other complications in this model, although they can be solved.

11.3 The WebDSLx approach

The solution for state management applied in WebDSL is not directly applicable in Ajax environments. There are two reasons for this. First, by using Ajax, only a part of the page is submitted to the server. The client-side Ajax library avoids that the whole page is being submitted, so that the user can interact with the other parts of the page while awaiting the response. As a result not all data will be submitted automatically to the server. Furthermore, the identifiers generated in the page to reproduce the server state will become invalid, since the structure of the page might have been changed by delta updates. In other words, a single identifier cannot be used to determine the current state of the application, since the current page is not produced by just a single state transition. However, WebDSLx provides a solution to those issues, one which still avoids the server to remember the state but still prevents some security issues such as object identifier hijacking.

11.3.1 WebDSLx state localization

In chapter 9 templates are introduced as the lowest level of granularity for delta updates, whereas the level of granularity in static applications is the whole page. The problems described in the previous sections are solved in WebDSLx by localizing the state management to templates. That is, the server no longer tries to reproduce the state for the whole page, but only for the template.

A first solution is to localize state management to the template that owns the invoked action. The client needs to encode with what parameters this template instance was called, just as is done for pages in WebDSL. All the techniques used to reproduce the state of a page in WebDSL are now used to reproduce the template that send the request. Templates become in fact a special kind of pages, as they can now serve requests on their own. In first instance this is an viable solution for two reasons: First, the contents of a template solely depends on the actual arguments the template is called by (the same holds for pages). Furthermore *form* boundaries cannot cross the boundaries of a template, that is, server request always originate from a single identifiable template (or page).

Therefore WebDSLx forms submit to the owning template of the form. The form encodes in hidden fields the actual arguments which were used to call this template. So basically by limiting the state management to the level of templates, the WebDSL server is able to restore any state information needed.

11.3.2 WebDSLx state localization 2.0

However, problems arise when the abstractions for advanced widgets (which will be explored in chapter 13) are used, since it invalidates the first reason why state localization works; the contents of templates no longer solely depends on the actual arguments when the *with/ require* syntax is used. First, a template was closed over its arguments. However, using the widget abstractions, elements from outside a template can be embedded inside a template and those elements might refer to variables or templates, defined outside the embedding template, as shown in listing 11.2.

```
define template B() requires C() {
  submit("Twice", twice())
  C()
  action twice() {
    append(this, C());
} }

template A( ) {
  B() with {
    C() {
      "hello world"
} } }
```

Listing 11.2: Problems introduced by using with/ require: The twice action cannot be submitted to template B any longer, since it requires template C which is provided by template A.

This construction invalidates the hash reconstruction used before, since the inserted elements themselves might be subject to delta updates, or provide forms based on a hashes generated outside the current template. So we have to make sure those closures are reconstructed when reconstructing the state of the current template. In listing 11.2 for instance, not only template *B,* but also template *A* needs to be reconstructed in order to server action *twice*. State reconstruction needs to start at a place where there is no danger of possible injection of other elements and their closures. On the other hand, we have to make sure that identifiers are not influenced by any later delta updates. For this reason WebDSLx starts reconstruction the state exactly at the invocation of the (delta update) operation that is responsible for the existence of the current action.

Given the delta operation "*replace (this, A(n))",* any action or form inside *A,* or any of the templates it invokes, will submit to A*(n). N* is the set of actual arguments, which are preserved. From this point on, the identifier hash can be safely reproduced using the approach mentioned earlier. Note that this approach demands that *A* has an empty closure; it should not require any templates nor depend on any variable outside its definition.

chapter 12 Ajax frameworks

To ease the development of Ajax applications, a large number of Ajax frameworks have been invented ([44] provides an overview). The Ajax technique creates the possibility to build rich interfaces, but does not provides the abstractions to do this in an easy and elegant way. Abstractions for automatic suggestions, drag and drop, updating, appending and replacing a part of the interface are not provided by the technique although they can be very well implemented. Useful JavaScript scripts that abstract over several of those issues have been combined into *Ajax frameworks*. Most frameworks leverage a broad set of functionalities to the developer, among others:

- Provide higher level functions to manipulate the DOM (Document Object Model), and abstract over implementation differences between browsers.
- Provide abstractions over request handling and state encoding and decoding.
- Provide a widget library. Those widgets usually have a close resemblance with the widgets used in desktop applications.
- Provide an event based user interface, with widget specific events.
- Abstractions for keyboard handling.
- Effects and animations.
- Hide the details of the more complex interaction model.
- Provide interoperability with third party services trough for example web services or JSONP requests.

12.1 Client and server driven approaches

Generally speaking, most frameworks are either client or server driven. Server driven applications generate the necessary HTML/ JavaScript code dynamically at the server. This often requires only a small client-side JavaScript library. Requests are interpreted by the server which sends delta updates back to the client, which are often directly injected in the DOM tree of the web page [19]. Client driven applications send and receive the raw domain data and user interface definitions. They are parsed and interpreted client-side. For instance the Google Web Toolkit uses this model [62]. This model has two advantages: The server is not required to keep track of the state and performance

might be better since less interaction with the server is required (section 11.2.2). As a downside, this approach might result in a high CPU load on the client since the demands on the JavaScript interpreter are higher. Client driven frameworks tend to provide a complete client-side runtime system build on top of JavaScript. Such libraries provide beside the usual widgets functionality for object-management, garbage collection, animation, state-management, partial loading etcetera. Examples of frameworks that provide such functionality are ExtJS [63], Dojo [55] and GWT [62].

12.2 Towards desktop-like interfaces

(X)HTML provides a very limited set of widgets to build complex interfaces. Ajax frameworks however seems to be fit to close the gap between the primitive widgets provided by HTML and the more complex widgets and abstractions provided by many interface builders. Frameworks have taken numerous widgets and abstractions provided by GUI builders and made them available to the web-developer. Many frameworks mimic layout abstractions which are not natively available in HTML; like docking, absolute coordinates, sliders, tree-views, drag and drop etcetera.

Although many new features and abstractions are enabled by these frameworks, they often do not implement their behavior as transparently as widget libraries, due to the more complex interaction model. Many interaction patterns provided by the OS or widget libraries need to be specified manually when using Ajax frameworks [2], although most frameworks try to make this as trivial as possible.

Ajax frameworks enable building more vivid web applications. They provide additional expressibility and functionality. However, they do not reduce the complexity of building Ajax applications significantly. Those framework still require the developer to understand many details of the HTML DOM, and the complex issues that arise when using CSS in different browsers.

12.3 Differences with desktop applications

As a result of the introduction of the Ajax technique and the frameworks build on top of it, the gap between web- and desktop applications has become smaller [7], as far as it concerns the user interface. In this section some of the problems and remaining differences between desktop- and web-applications are summed up. Issues not solved yet by most frameworks.

Eye Candy. Desktop applications have a consistent styling, which is provided by the window manager. Web applications however try to distinguish themselves among each other, by adding all kinds of eye-candy, such as animations and complex styles. Therefore the possibility to customize widgets is an important issue in the development of web applications. Most Ajax frameworks provide a default style and provide (CSS) hooks to change the style.

Local and remote functionality. The functional logic might, depending on the nature of the framework, reside partially at the client, and partially at the server. Detecting where to execute certain logic is an important issue. For example, security related functions should be executed at the server, while styling related functions (such as highlighting or drag and drop) should be executed by the browser without requiring a request to be send to the server. A developer needs to deal with this separation when building web applications.

Readable URLs. Since requests are sent in the background when using Ajax, readable URLs that indicate the current state are not or only partially available; Single Page Interfaces do not cause the browser to navigate to other pages. For this reason developers often use a hybrid page model, or write additional JavaScript to achieve nice, book-markable URLs.

Platform stability. Developing web applications requires more knowledge of the details of the target platform than developing desktop applications. In desktop applications, important platform constraints are the programming language, the operating system, the database-system and the window-manager. The choices for most parameters are limited, as mentioned in section 5.2.1 . Web applications are often targeted at more specific platforms: the operating system of the web server, the programming language, the database, the JavaScript framework and the browser all need to be taken into account. It might be quite hard to abstract over multiple values for those parameters. Furthermore, new techniques or languages are introduced on a regular basis.

Maturity. Due to the fact that many frameworks are developed as open source, built by volunteers, and the fact that the platform as a whole is subject to many changes, most frameworks lack proper tooling such as interactive debuggers. A popular debugger is FireBug [64] but this tool does not debug at the level of the provided abstractions, but at the level of the JavaScript/ HTML constructions those abstractions are translated to. Furthermore, there are hardly any visual designers and documentation is widely recognized as 'poor' for most frameworks.

12.4 What is next

In this chapter we introduced the phenomenon of Ajax frameworks. They provide the possibility to give web applications the look and feel of desktop applications. But still, differences exists, and developers often cannot use the abstractions developed for developing desktop applications. In the next chapter the widget abstractions of WebDSLx are proposed, which are useful to develop advanced widgets or wrap third party widgets from existing Ajax frameworks.

chapter 13 Widget abstractions in WebDSLx

The popularity of interface builders is a result of the possibility to easily develop graphical user interfaces using a consistent set of widgets which can be configured easily. Since the set of widgets available in many interface development systems is quite uniform, developers expect those widgets in any interface building system. HTML however provides a limited set of not very sophisticated widgets. For example creating a drop-down box with images inside is complicated. Until the introduction of Ajax frameworks, web interfaces were not able to provide a *rich* user experience, compared to most desktop applications[4]. This chapter describes which abstraction WebDSLx provides to ease the use and development of complex widgets.

13.1 WebDSL widget supporting features

During the development of the WebDSLx extension, the template mechanism of WebDSL was extended with a few useful features which support the creation of advanced reusable widgets. The first one is that template calls can be parameterized with a set of elements. This was already supported for built-in templates, but has been generalized. By using the reserved *elements()* call, a template definition can insert the provided elements. Using this constructions, many built-in templates can now be expressed as user defined templates as well.

Furthermore an XML embedding has been added to the WebDSL language, which allows to generate arbitrary XML (and thus XHTML) in templates. This allows the developer to have more fine grained control over the XHTML output, which might improve layout and styling. Furthermore composite structures like panels can be developed, as long as they only expect a single set of elements as children. For example the built-in template *par* (which wraps its contents in a HTML <*p*> element) can now be defined using listing 13.1.

4 Using Flash or Java can solve this issue, but it requires the user to install certain plugins in his browser

```
define par () {
  <p>
  elements()
  </p>
}
```

Listing 13.1: Defining the par *element in WebDSL*

The XML embedding can only be used to insert nodes, and not for arbitrary text or CDATA elements. In practice this is sufficient. One exception though is the usage of JavaScript. JavaScript cannot be added to a template using a WebDSL string, since it will be escaped automatically (another abstraction of WebDSL). For this reason the *<script>* tag is added to the language, which accepts any string, which will be streamed literally to the HTML output. This provides the possibility to generate JavaScript. In order to be able to use WebDSL expressions inside the JavaScript embedding, ~(tilde) escapes to a WebDSL expression, which will to be evaluated before streaming to result the client. Listing 13.2 provides an example.

```
define no-span template loadCSS(stylesheet: String) {
  <script>
    loadCSS("~(baseURL())"+"/stylesheets/"+"~stylesheet");
    if (typeof(console) != "undefined") {
      console.log("additional style sheet has been loaded: "+"~stylesheet");
    }
  </script>
}
```

Listing 13.2: Example of the <script> embedding provided by WebDSLx

13.2 Template attributes

Chapter 10 provided a quick glance of attributes which can be added to a call. Those attributes provide the possibility to override properties of a widget and thereby fulfill a role similar to the *object inspector* in many interface builders. Attributes are always optional. Most built-in templates support attributes. Those attributes will be translate to HTML attributes, and inserted in the outermost node of the HTML code of such template. For example

```
image("close.png")[width:= "20px", class:= [redborder, roundborder],
  alt:="not found", id:= myimage]
```

translates to :

```
<img src="close.png" width="20px" class="redborder roundborder"
  alt="not found"/>
```

Attribute values can be either identifiers, (inline) actioncalls, or expressions such as a string literal. In the template definition the value of an attribute can be retrieved using the *attribute(key: String, default: String)* function. If the property the key refers to is not provided, the default value will be used. All attributes return string value, even identifiers and events (which contain a JavaScript expression in a string, so that the value can be assigned to the HTML event attributes). The *rndButton* definition in listing 13.3 shows multiple invocations to the *attribute* function.

13.3 Events

Events are a specific kind of attributes, referring to an action (callback) defined in the scope of the call to the template. In a simple scenario, the widget is only responsible for invoking the callback. For example in listing 13.3 the *rndButton* template invokes the *onclick* event (note that the *onclick* attribute is actually passed to the *container,* which will set the HTML *onclick* attribute of the resulting <*div*> element.

```
define no-span toolbar(doc: Document) {
  rndButton("Remove", true)[onclick:=remac(),  class:= middlebtn]
  action remac() {
    replace(popup, delete_popup(doc));
  }
}

define no-span rndButton(kind: String, showCaption: Bool) {
  container[ class:=rndButton,
    onclick:= attribute("onclick","")
  ] {
    image("images/"+kind.toLowerCase()+".png")
  }
}
```

Listing 13.3: Example of an onclick event and callback

In some cases, it is desirable that the widget itself provides the actual arguments of the callback, for instance, when the users selects an item in a list, the callback expects the current selection as argument. The *event*(*handler, args)* construction takes care of this concept. The *handler* parameter defines which callback should be invoked and *args* provides a key-value map with arguments. The keys refer to the names of the arguments of the callback and value is a JavaScript expression which provides the actual value. Listing 13.4 displays how we can create list callbacks using this construction. Note that the callback is still passed as function call and not as function reference to the template (a default value for the argument is provided at the call site).

```
define page testpage(post: BlogPost) {
  "TAGS" spacer
  stringList(post.tags)[onclick := listclicked(null)]
  action listclicked(value : String) {
    append(this, template { "clicked: " output(value) });
  }
}

define stringList(items: List<String>) {
  table {
    for(item: String in Items) { row {
      column[onclick := event(onclick, [value := "this.innerHTML"])]
        { output(item) }
    } }
  }
}
```

Listing 13.4: Advanced callback using the event *construction*

13.4 Input widgets

WebDSL provides automatic binding of built-in input templates (section 8.3.2). When using custom widgets, this behavior is lost since the compiler has no knowledge about the fact that a template acts as an input template. To solve this issues WebDSLx introduces the *inputtemplate* template modifier. This modifier indicates that the first argument passed to this template returns a value which needs to be bound to the original expression when an action is executed. Furthermore the last formal argument should be of type string, and not be provided by the call, since this last argument will be provided by the compiler and indicates the name under which the data needs to be post in order to rebind the value automatically. Listing 13.5 demonstrates an input template. The string *s* will be bound to the inputted value automatically.

```
define inputtemplate myinput(val: String, name: String) {
  <input name=name type="text" value=val />
}

define page aninputtest(s: String) {
  "current value: " output(s)
  form {
    "enter new value: " myinput(s)
    action("go",save_s())
  }
  action save_s() {
    //s will be bound to the new value here
    return aninputtest(s);
} }
```

Listing 13.5: Example of an input widget

13.5 Complex compositional widgets

In the abstractions provided by WebDSL(x) thus far, reusable pre-defined composition of widgets is limited to wrapping elements. In interface development however, many re-occurring complex compositional patterns exist to create consistency in user interfaces. In figure 13.1 the two applications are globally divided in three parts, a *toolbar*, a content *outline* and a page *view*. Although the *subjects* of both applications have nothing in common, the *interface structure* is similar: Selecting an item in the tree results in the details of the object being shown in the content pane. This hints that one should be able to combine sets of widgets into reusable compositions.

A solution to this compositional problem might be found by promoting template definitions to first class citizens. By providing the possibility to parameterize templates with templates, and by by using *generics* and *interfaces* (in Java terms), one could create reusable compositional widgets. Listing 13.6 shows how using such constructions, a *masterdetail* widget, that draws a tree and a detail view could be constructed. In the example the masterdetail accepts a set of objects, which implement the *TreeItem* interface, and a set of templates that can display those objects.

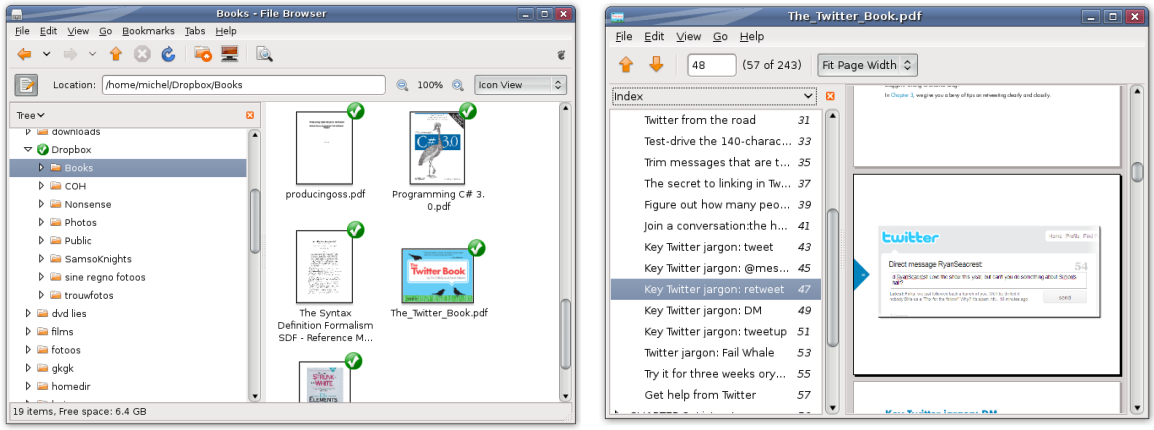

Figure 13.1: Example of different applications with a similar interface structure (toolbar, tree and content)

```
interface TreeItem {
  parent: TreeItem;
  children: List<TreeItem>
}

define template masterdetail<T extends TreeItem>
  (root: T, treeview: template(T), detailview: template(T)) {
  container[id:= tree] {
    treeNode<T>(root, treeview, detailview, false)
  }
  divider()
  container[id:= detailviewholder] {
      "(nothing selected)"
} }

define template treeNode<T extends TreeItem>
       (item: T, treeview: template(T), detailview: template(T),
            collapsed: boolean) {
  //draw current node
  /* code to draw a + / - sign should go here */
  navigate()[onclick := click] {
    treeview(item)
  }
  //draw children
  if(!collapsed) {
    foreach(child: TreeItem in item.children) {
      treeNode(child), treeview, detailview, true);
  } }
  action click() {
    replace(detailviewholder, detailview(item);
} }
```

Listing 13.6: Defining a tree using interfaces, generics and first-class templates

This proposal allows to define the *masterdetail* view quite elegant. However, it has a few drawbacks. The abstractions are at a low level of abstraction; it introduces several notions which turns our nice domain specific language almost into a general purpose language, by introducing a complex full blown type system. Notions such as *generics* might be unfamiliar to unexperienced developers. Furthermore this solutions requires changes that have deep impact in the WebDSL type

system, which is required to become almost as sophisticated as that of a general purpose language like Java.

For those reasons WebDSLx takes the road less traveled. Templates can be passed to other templates using a dedicated syntax construction, as demonstrated in the *masterdetail* listing shown below. A template can be provide or require templates using the *require* and *with* keywords. In listing 13.7, the *masterdetail* template states that it requires the parameterless templates; *detailview* and *masterview*. Furthermore it invokes the *twoColumns* widgets, which requires the templates *left* and *right*, which are provided using the *with* clause.

```
define template masterdetail() requires detailview(),masterview()  {
  twoColumns[
   width:= "100%",
   left := attribute("left",""),
   right:= "100%"
  ] with {
    left() {
      collapseLeft() {
        masterview()
    } }
    right() {
      detailview()
} } }
```

Listing 13.7: First example of the with/ requires syntax

Although it seems that templates now can be passed as first-class citizens, this is not really the case, since it is not possible to produce templates from functions or to store them in variables. The *with* clause in fact lifts its templates to uniquely named global templates, and puts references to them in the closure of the (in the example *twoColumns*) call. The *requires* clause signals the type checker that it expects a *with* clause with the proper templates to be in scope at every call to this definition.

The *with/require* syntax allows to define widgets which accept a limited set of named elements, which is useful for defining layouts containing multiple columns, headers, bodies etcetera. This is something most composite widgets in interface builders are also capable of. In addition to this functionality, the *with/require* syntax can be used to create iterating, recursive compositions, by using the fact that required templates can be parameterized and WebDSL entity types can be subtyped. For this reason the definition in listing 13.8 can be used to create a tree. Note that using a TreeItem *interface* would be more elegant, but WebDSL does not support interfaces yet. In the example the *treeView* template should define how an domain object should be displayed in the tree.

```
entity TreeItem {
  parent -> TreeItem
  children <> List<TreeItem>
}

define customTree(item: TreeItem) requires treeView(TreeItem) {
  container {
    /* here could be some +/- sign responsible for collapsing */
    treeView(item)
  }
  container[style:="padding-left: 16px"] {
    for(child: TreeItem in item.children) {
      customTree(child)
} } }
```

Listing 13.8: A tree defined using the with/require syntax

13.6 What is next

This chapter describes the abstractions provided by WebDSLx to create reusable widget libraries. It extends template calls with the notion of *attributes*, which can be used to influence the appearance of both built-in and in WebDSL defined templates. Furthermore those attributes can be used to define events or callbacks, especially in combination with the *event* construction. In addition, by using the *elements()* call or the *with/require* syntax complex composite widgets can be defined. Finally, widget that support automatic variable binding can be developed using the *inputtemplate* modifier.

The next chapters describe two case studies of web applications built using the WebDSLx extension. The second case study demonstrates the power of the widget abstractions by both defining new widgets and by wrapping existing widgets taken from an external widget library.

chapter 14 Case study: MyToDo

This chapter describes the first case study of an application built using WebDSLx. This application is used to test the abstractions introduced in chapter 10. The interface of the application will be constructed and updated by using delta updates.

14.1 MyToDo

The purpose of the *MyToDo* application is to manage a simple ToDo list. Those ToDo items can be organized in categories. The user can search for ToDo items, and mark them as important or finished. This results in the simple WebDSL data model shown in listing 14.1). Throughout this chapter the complete sources of this application will be discussed in detail.

```
entity ToDo {
  name        :: String
  details     :: Text
  finished    :: Bool
  urgent      :: Bool
  folder      -> Folder (inverse = Folder.todos)
}
entity Folder {
  name        :: String
  description :: String
  todos       <> List<ToDo>
}
```

Listing 14.1: Data model of the MyToDo application

14.2 User interface structure

The user interface of the MyToDo application (Figure 14.1) is designed as a Single Page Interface, in which, for instance, items can be edited in-place. The structure of the user interface is defined by the root page (Listing 14.2). This page defines the header, the tool bar (with on the left a button that collapses the folder list, and on the right a search bar function), some placeholders and the footer. The collapse functionality is written using an inline action. Furthermore different parts of the interface have a *class* attribute, which act as hooks for the CSS style used. The two placeholders act as target for the delta updates. The *folderlist* placeholder immediately loads the *folder* template.

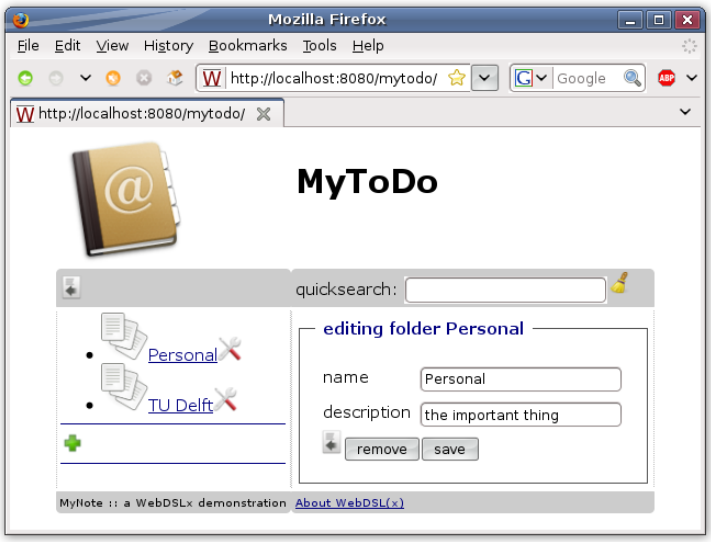

Figure 14.1: Interface of the MyToDo application

```
define page root() {
  block[class := mytodo] {
    table {
      row[class := headerrow]{
         navigate(root()) { image("/images/gohome.png") }
         header{"MyToDo"}
      }
      row[class := graybg] {
        image("/images/toggle.png")
          [onclick :=  action {visibility (folderlist , toggle); }]
        quicksearch()
      }
      row[class := middlerow] {
        placeholder folderlist {
          folders()
        }
        placeholder todolist {
          "(please select a folder)"
      } }
      row[class := footerrow] {
        text("MyToDo :: a WebDSLx demonstration")
        navigate(url("http://www.webdsl.org")) { "About WebDSL(x)" }
} } } }
```

Listing 14.2: Interface structure of the MyToDo application

14.3 Quick Search

The MyToDo application provides an interactive search function. Pressing a key the *quicksearch* box results in an immediate update of the right part of the interface, which will contain matching search results (Figure 14.2). Such behavior is typical for Ajax applications and should be easy to define. The *quicksearch* template is defined as shown in Listing 14.3. The *updatesearch* action is hooked to the HTML *onkeyup* event. The action searches all nodes (in a bit naive way) and adds all matching items to the *todolist* placeholder (defined in Listing 14.2). The *clear* action is an inline action, hooked to the *onclick* of the image and replaces the right part of the interface with a small inline template.

```
define quicksearch() {
  var search : String := "";
  form {
    "quicksearch: "
    input(search)[onkeyup := updatesearch(search)]
    image("/images/clear.png")[onclick := action {
      replace(todolist, template{ "cleared" } );
  } }]
  action updatesearch(val: String) {
    clear(todolist);
    append(todolist, template { "searched " output(val) spacer });
    for(n : ToDo order by n.name) {
      if (n.name.contains(val)) {
        append (todolist, displayToDo(n));
} } } }
```

Listing 14.3: Quick search definition

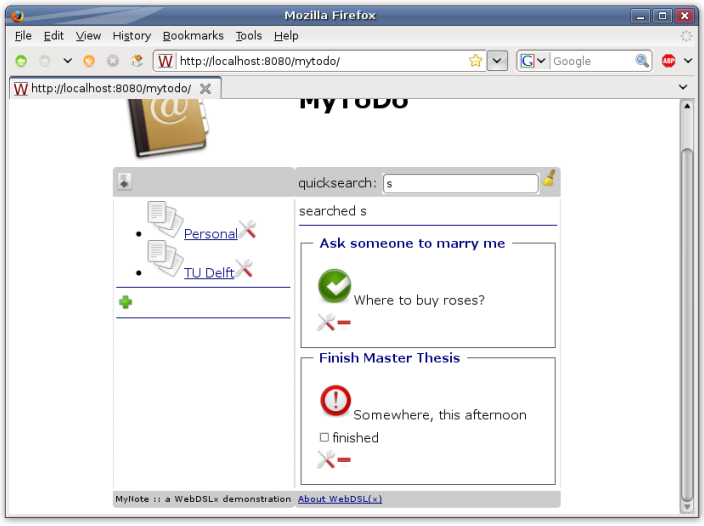

Figure 14.2: On-the-fly result of searching 's'

14.4 In place interaction

The MyToDo application has several applications of in-place editing and updating. For example when the user marks the *finished* box (Figure 14.3) the item is immediately marked as being finished, without reloading other parts of the page; for instance the search box still contains the same input string. Combining in-place editing with delta updates has the advantage that the user can edit multiple ToDo items simultaneously; when one form is submitted, the other form will remain in its current status (Figure 14.4). This kind of interaction is very hard to achieve in classic web applications.

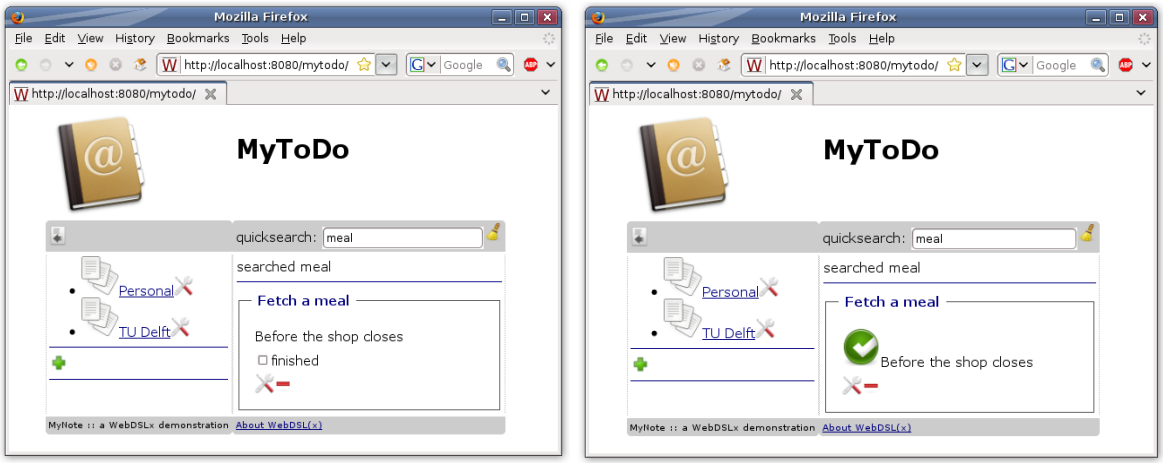

Figure 14.3: In-place update of a ToDo item by pressing the 'finished' checkbox (before and after)

Listing 14.4 defines three templates. *Foldercontents* displays ToDo items per folder and enables the user to can create new ToDo items by clicking the add button. *DisplayToDo* describes how a single ToDo item should be rendered. Furthermore it defines the action which will be executed when the user marks the *finished* checkbox. This *finish* action returns a delta update which replaces the template that fired the event with the new version of the same template. Furthermore the *remove* action just removes the template. Pressing the *edit* button replaces the displayToDo template with the *editToDo* template (Figure 14.4). When the user finishes the manipulation of the ToDo item, the editToDo template will be replaced back by the displayToDo template.

```
define foldercontents(f : Folder) {
  output(f.name)
  var newToDo : ToDo := ToDo{}
  form {
    input(newToDo.name)
    image("/images/add.png")[onclick := addtodo()]
  }
  action addtodo() {
    newToDo.folder := f;
    newToDo.save();
    append (foldertodos , displayToDo(newToDo));
  }
  spacer
  text ( "current todos" )
  block[id := foldertodos] {
    for(todo : ToDo in f.todos  order by todo.name) {
        displayToDo(todo)
} } }

define displayToDo(n: ToDo) {
  form { group(n.name) {
      row { column {
          if(n.urgent) {
            image("/images/urgent.png")
          }
          if (n.finished) {
            image("/images/finished.png")
          }
          output(n.details)
      } }
      if (n.finished == false) {
        row { column {
            var b: Bool := false
            input(b)[onclick := finish()]
            container[onclick:= finish()] {"finished"}
      } } }
      row { column {
          image("/images/edit.png")
            [onclick:= action { replace (this , editToDo(n));}]
          image("/images/remove.png")[onclick:= remove()]
  } } } }
  action finish() {
    n.finished := true;
    n.save();
    replace (this, displayToDo(n));
  }
  action remove() {
    var f: Folder := n.folder;
    f.todos.remove(n);
    n.delete();
    clear(this);
} }

define editToDo(n: ToDo) {
  group("editing todo "+n.name) { form { table {
        row{"name "    input(n.name)}
        row{"category" input(n.folder)}
        row{"details"  input(n.details)}
```

```
        row{"urgent"   input(n.urgent)}
      }
      navigate[onclick:= close()] { image("/images/toggle.png") }
      action("remove", remove())
      action("save", save())
    }
    action close() {
      replace(this, displayToDo(n));
    }
    action save() {
      n.save();
      replace(this, displayToDo(n));
    }
    action remove() {
      var f: Folder := n.folder;
      f.todos.remove(n);
      n.delete();
      clear(this);
} } }
```

Listing 14.4: Source of the templates foldercontens, displayToDo *and* EditToDo.

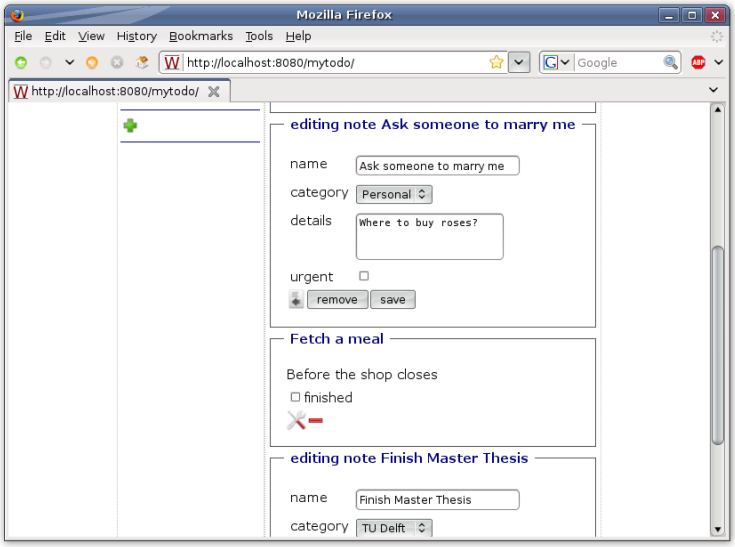

Figure 14.4: Simultaneous manipulation of multiple ToDo items

14.5 Manipulating folders

Editing folders (Figure 14.1) is quite similar to editing ToDo items, except that it does not apply in-place functionality. The source for folder management is shown in Listing 14.5. The *folders* template iterates over all available folders and displays their names. When the user hovers the mouse over the name, the description of the folder will be shown at the bottom. This behavior is defined using in-line actions and templates.

The *editFolder* template is responsible for the manipulation of folders. Most *editFolder* actions result in two delta updates. One for each side of the user interface. This is necessary because otherwise the manipulated folder would be properly saved, but the view on the left would not be updated. In a synchronous interface there would be no need to specify this update, since the whole page would be rendered. By using delta updates however, such constraints need to be enforced.

```
define folders() {
  action doEdit(f: Folder) {
    replace (todolist ,editFolder(f));
  }
  list {
    for(f : Folder  order by f.name) {
      listitem {
        image("/images/todos.png")
        navigate()[
          onmouseover := action {replace(folderdetails,
            template { block {output(f.description) }}
          );},
          onmouseout := action {replace(folderdetails,
            template { block { par { "" } }});} ,
          onclick :=
            action {replace(todolist, foldercontents(f));}
        ] { output(f.name) }
        image("/images/edit.png")[onclick := doEdit(f)]
  } } }
  spacer
  image("/images/add.png")[onclick:= createfolder()]
  action createfolder() {
    var newfolder: Folder := Folder{};
    newfolder.save();
    replace(todolist, editFolder(newfolder));
  }
  spacer
  placeholder folderdetails {}
}

define editFolder(f: Folder) {
  group("editing folder "+f.name) {
    form {
      table {
        row{"name " input(f.name)                    }
        row{"description" input(f.description) }
      }
      image("/images/toggle.png")[onclick:= close()]
      action("remove", remove())
      action("save", save())
    }
    action close() {
      replace (todolist, foldercontents(f));
    }
    action save() {
      replace (folderlist ,folders());
      replace (todolist, foldercontents(f));
    }
    action remove() {
      f.delete();
      replace (folderlist, folders());
      replace (todolist, template { block{ "select a folder.."}});
} } }
```

Listing 14.5: Folder management in the MyToDo application

14.6 Evaluation

The MyToDo demonstrates how rich interaction can be achieved by using a compact application definition (about 200 LOC) and some simple delta updates (clear, replace, append and visibility). Surprisingly, this seems to be enough to reach all kind of complex interaction patterns, such as in-place editing and updating, and search-while-you-type. The applications however also displays a leak in the abstractions; the developer needs specify which parts of the interface needs to be updated. But all in all, the WebDSLx primitives seems to work pretty well and can be used in a natural way. The delta updates provide smooth user interaction and save bandwidth.

chapter 15 Case study: Outliner

In this chapter we look into another case study, the *Outliner* application (Figure 15.1). This application features a more complex, recursive data model and will enhance the user-experience by building advanced widgets, using the abstractions provided in chapter 13. Those widgets include a popup dialog, trees, in-place editors, a lazy loading tab control, collapsing panels and drag and drop related widgets. The widgets built for this applications should be generic, that is, reusable in other applications and fit to put in a library (WebDSL has no notion of libraries yet).

The purpose of the Outliner application is to specify a hierarchical structure of small notes which could aid the development of applications, documents etcetera. The Outliner application supports three types of notes; headers, text-blocks and images. The data model is defined in Listing 15.1.

```
entity Document {
  name :: String
  description :: Text
  root <> HeaderNode
}
entity HeaderNode : TreeItem {
  caption :: String
  depth :: Int
}
entity TextNode : TreeItem {
  contents :: Text
}
entity ImageNode : TreeItem {
  image :: Image
}
```

Listing 15.1: Data model of the Outliner application

15.1 User interface of Outliner

Figure 15.1: Default view of the Outliner application.

1. Toolbar. The first three buttons will show a modal dialog where the user can further specify his action

2. Current open document. The name can be edited in-place. Hovering will show the description of the current document. Furthermore a close button to close the document

3,4. Grips to hide parts of the interface

5. Tab-pages to switch between the edit and print view

6. Outline of the current document. Clicking an item will limit the main view to the selected header

7. Breadcrumbs path to the current selection.

8. A header item. Clicking the title allows to edit the header. Pressing the arrow will collapse this section

9, 10. Example text and image item.

11. Item toolbar. Using the *hand* button, the item can be moved around using drag and drop. The *zoom* button hides all items outside that item. *Delete* can be used to remove the item. The small black *arrow* will bring up the *insert new item* toolbar (12)

12. Toolbar to insert new items. Header, text or image. Furthermore a hide button.

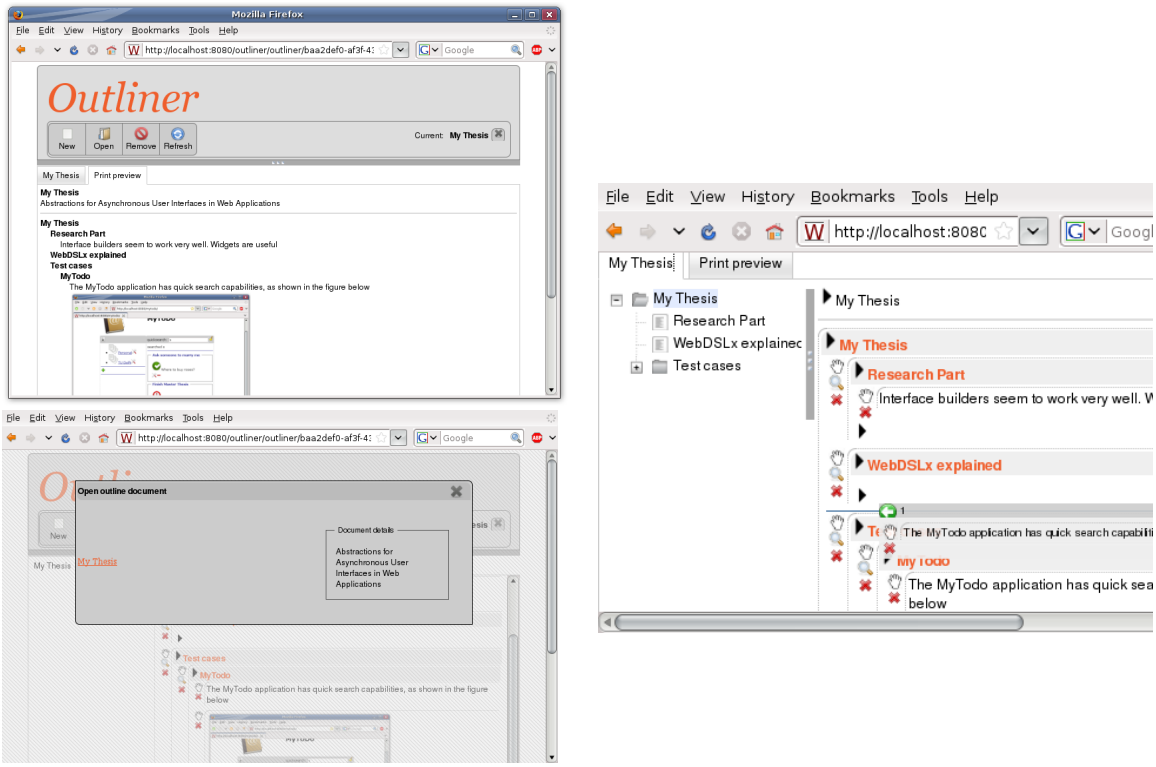

Figure 15.2: Outliner: print preview, drag & drop and popup dialog

15.2 In-place field editor

Before elaborating on the sources of the Outliner application, this section and the next ones will first describe how several of the required widgets have been developed. In figure 15.1, the name of the current document is displayed in the toolbar. By clicking on this name, the text transforms into an input field, which, when the user leaves the input field, immediately persists the new name in the database and transforms back into an ordinary label. Listing 15.2 shows the source of the in-place edit widget. The widget is defined as input widget, for which reason its value will be bound automatically when the form is submitted. In addition, the form will be submitted automatically when the user leaves the input field. The first example in listing 15.3 shows how one can make use of the automatic form submit and variable binding. Another approach is to supply an *onblur* callback. This action will be invoked whenever the user changes the value, without submitting a form. This is demonstrated by the second invocation in the listing.

The template itself contains HTML, JavaScript and CSS. Those standards are all have their own quirks, and expert knowledge is required to built such widgets. The WebDSLx abstractions do not change this. However, by building such widgets, the knowledge can be hidden in a widget library and accessed through a uniform interface. Note that, especially in the first invocation, the call cannot be distinguished from a call to an ordinary input. The semantics are identical. The possibility to box and hide knowledge which can be accessed using a uniform interface is what makes the WebDSLx widget abstractions useful. In the next section we will evaluate a more complex widget.

```
define inputtemplate inplaceFieldEdit(value: String, name: String) {
 <script>
  function showTA(t,a) {
   a.width=t.clientWidth;tclassName='hidden';a.className='visible';
  }
```

```
  function hideTA(t,a) {
   a.innerHTML=t.value;a.className='visible';t.className='hidden';
   if (t.form != null) { t.form.onsubmit(); }
  }
 </script>
 <span onclick="showTA(this,this.nextSibling);" class="visible">
    output(value)
 </span>
 <input onblur="hideTA(this, this.previousSibling);"+
      event(onblur,[value:="this.value"])
    name=name
    value=value
    style="width:100%" class="hidden"
 />
}
```

Listing 15.2: Source of an in-place editor widget

```
//form based invocation
form {
  inplaceFieldEdit(user.name)
  //a default submit action needs to be provided:
  action("save", save())[id:= submit, class:= "hidden"]
  action save() {
    /* username will be bound and saved automatically here */
    refresh();
} }

//event based invocation (no form required)
inplaceFieldEdit(user.name)[onblur := save(null)]
action save(value: String) {
  user.name := value;
  refresh();
}
```

Listing 15.3: Two examples of using the inplaceFieldEdit widget

15.3 Dojo widgets

As described in chapter 12, Ajax libraries provide interactive widgets which might save the developer the wearisome task of developing common widgets. Those widgets can often be configured by writing JavaScript statements, or declaratively by using XML (Listing 4.4, 15.4). Since WebDSLx provides abstractions to box a JavaScript or XHTML based widget in a WebDSL template definition, reusing existing Ajax widgets will save a lot of development time. This not only increases the productivity, but also the quality of the widgets, since some Ajax libraries have a large user base and are maintained properly.

Dojo [55] is one of the most popular and sophisticated Ajax frameworks [44]. Dojo has two advantages which makes it easy to embed the framework in WebDSL. First, Dojo widgets can be instantiated declarative using XHTML elements. Since arbitrary parts of a template can be specified by using XHTML in WebDSL, this allows for easy configuration of Dojo widgets. Furthermore, the Dojo sources can be loaded dynamically in a web page using a *Content Delivery Network,* which

allows us to avoid configuration issues at the cost of a speed penalty. In the Outliner application, the drag and drop, tab control and tree-outline widgets (Figure 15.1 items 5 and 6) are based on Dojo widgets.

15.3.1 *The tab control*

The definitions of the *tabs* and *tab* templates are fairly straightforward (Listing 15.4) and depend heavily one the WebDSL XHTML syntax, and the usage of the *elements*() construction. By setting the *dojoType* attribute, Dojo widgets can be instantiated. The *elements()* call is used to ensure that any child elements are placed inside the *TabContainer*. The definition of template *tab* is quite similar to the definition of template *tabs*. The *no-span* option of this template indicates that the template should not generate a placeholder, since this would change the structure of the DOM tree of the control, which would harm the Dojo TabContainer layout.

```
define template tabs() {
  <script>
    dojo.require("dijit.dijit");
    dojo.require("dijit.layout.TabContainer");
    dojo.require("dijit.layout.ContentPane");
  </script>
  <div id="mainTabContainer" dojoType="dijit.layout.TabContainer"
     class="tundra" style="width:100%;height:"+attribute("height","600px")>
     elements()
  </div>
}

define no-span template tab(title:String) {
    <div dojoType="dijit.layout.ContentPane" title=(title)>
      elements()
    </div>
}
```

Listing 15.4: The Dojo based Tabs and Tab widgets

In the Outliner application, the *print preview* tab loads lazy. The contents of the tab will not be fetched from the server until the user enters the tab. This has two major advantages. First; the user will always see the most recent version of the document and secondly the main view loads a bit faster since it is not required to render the *print preview*. It should be possible to express this notion of lazy loading in the model and use it declaratively. The tab widget should be able to hide the fact that a template call is postponed.

Using the *with/require* syntax, template calls can be passed around. Since those templates can be used in actios as well, we can use the template passing system to implement lazy loading, as shown in listing 15.5. The *tabs, tab* and *lazytab* templates can be called from the main view of the Outliner application, which is shown in listing 15.6.

```
define no-span template lazytab(title:String) requires loading(),contents(){
  <div dojoType="dijit.layout.ContentPane" title=(title)>
    <script type="dojo/method" event="onShow">
      console.log("Loading lazy tab");
      findTopDown(this.domNode,'loader').onclick();
    </script>
    form[class:=hidden]{
      button(">", action{replace(tabcontents, contents());})[id:=loader]
```

```
    }
    placeholder tabcontents {
      loading()
    }
  </div>
}
```

Listing 15.5: Definition of a lazy loading tab in WebDSLx

```
define main(doc: Document) {
  tabs[height:="550px"] {
    tab(doc.name) {
      masterdetail() with { masterview() {
            documentTree(doc)
        } detailview() {
            detailView(doc.root)
    } } }
    lazytab("Print preview") with { contents(){
        printpreview(doc)
    } loading() {
        "loading print preview.."
} } } }
```

Listing 15.6: Outliner's main view; example usage of tabs, tab *and* lazytab. Printpreview *will be loaded lazy.*

15.3.2 *Tree and drag and drop widgets*

The tree and drag and drop widgets are the other widgets based on the Dojo toolkit. They are developed in a similar way as the tabcontrol widget. A complication of the Dojo Tree widget is that it requires a JSON (or comparable) data-source. For this reason the call to the tree (Listing 15.7) is parameterized with the URL of the *documentoutline* page, which returns the structure of the document in JSON format. Since WebDSL can generate non-HTML based pages, we are able to express a JSON generator in the language. The second argument contains the ID of the tree element we want to use as root, which is the root of the document itself in this case. Dojo strongly relies on the use of unique identifiers. Since every WebDSL object has an *id* field containing such identifier, this abstraction matches perfectly. When the user selects an item in the tree, the *selectHeader* callback will be invoked, with the id of the selected object as argument.

```
dojoTree(navigate(documentoutline(doc)), doc.root.id.toString())
  [onselect:=selectHeader(null), width:= "180px"]
```

Listing 15.7: Invocation of the (Dojo based) Tree

For drag and drop three widgets have been developed. The *dndSource* allows *dndItem*s to be dropped onto it, and invokes the *ondrop* handler which provides as arguments the *item* that has been dropped, the *target* it has been dropped on, and the *index* to determine the drop position. If a *dndHandle* is used inside a *dndItem,* the user can use this object to start dragging. If a *dndHandle* is not provided, the whole item can be used to start dragging. However this default behaviour might interfere sometimes with other input elements used inside the *dndItem.* The drag and drop widgets are used in the *viewHeader* template, which is shown in listing 15.9.

The development process of the Outliner application has been eased significantly by reusing existing Dojo widgets. Wrapping non-WebDSL widgets in WebDSL templates can hide many details, such as the fact that the tree widget is actually build using three different Dojo widgets. Although the developer of the widgets is required to dive into the details of the Ajax library and to have JavaScript skills, this knowledge can be boxed into definitions which are easy to reuse and do not require this knowledge.

Only the tree widget shows a small architectural mismatch; it requires a JSON data source. This problem has been solved by adding an additional page which generates the JSON data. A better solution would be adding a JSON generator and parser to the compiler. This would solve many potential interoperability issues since it is an commonly used data format in Ajax libraries and web services.

15.4 The Outliner application (continued)

```
collapsePanel
collapseUp
collapseLeft
dndSource
dndItem
dndHandle
dojoTree
```

```
dojoDndTree
inplaceFieldEdit
inplaceTextArea
loadCSS
twoColumns
footerLayout
masterDetail
```

```
loadDojo
popup
rndButton
tabs
tab
lazytab
webdslTree
```

Listing 15.8: Widgets developed for the Outliner application

Defining the behavior of the Outliner application becomes trivial when ready-to-use widgets take care of the fine grained interaction patterns. Listing 15.8 lists all widgets which have been developed for the Outliner application. Because the widgets take care of the details, the remainder of the Outliner sources are only slightly more complex than the sources of the MyToDo application.

The most interesting part of the application is the *viewHeader* template (Listing 15.9), which is responsible for rendering a *header* item and its contents in the main view (Figure 15.1 item 8). The template starts with rendering a *collapsePanel* which requires a *caption* and some *contents*. The *caption* contains an in-place input field, which invokes the *savehdr* action when the value is changed by the user. The *contents* contains a drop region (*dndSource*). Arbitrary Outliner items can be dropped on a header item. The action *dropelement* will be executed when such an item is dropped. The action first needs to convert the provided strings into objects (callbacks only support string types in the current implementation). After that it checks whether the drop operation would create cycles in the tree, before persisting the change.

A *dndItem* will be created for every child element of the header. Again, the *id* field is provided as unique identifier. Each child is rendered in *two columns*, on the *left* the context sensitive buttons (figure 15.1 item 11) and on the *right* the item itself, which is a recursive call to the *nodeView* template, which acts as a dispatcher. The *hand* image acts as *dndHandle* which starts the drag and drop operation.

```
define no-span viewHeader(item: HeaderNode) {
 block[id:= viewHeader, class:=[scopediv, viewHeader]] {
  collapsePanel(false) with {
    caption() {
      inplaceFieldEdit(item.caption)[onblur := savehdr(item, null)]
    }
    contents() {
      dndSource(item.id.toString(), true)
        [ondrop:= dropelement(null,null,null), style:= "min-height: 10px"]
      {
        for(child : TreeItem in item.children) {
          dndItem(child.id.toString()) {
            twoColumns[left:="10px"] with {
              left() {
                dndHandle() { image("/images/hand.png")[height := "16px"] }
                if (child isa HeaderNode) {
                  break
                  image("/images/system-search.png")
                    [onclick:=zoomac(child as HeaderNode),height := "16px"]
                }
                break
                image("/images/dialog-cancel.png")
                  [onclick:=remac(child),height := "16px"]
              }
              right() {
                nodeView(child)
      } } } } }
      itemadderhidden(item)
 }} }
  /* TEMPLATE ACTIONS */
  action savehdr(item: HeaderNode, value: String) {
    item.caption := value;
    replace(statusBar, template{ "Saved header " output(value) });
  }
  action remac(child: TreeItem) {
    var p := child.parent;
    p.children.remove(child); child.parent := null;
    p.save(); child.delete();
    replace(statusBar, template { "Object removed from tree "});
    replace(viewHeader, nodeView(p));
  }
  action zoomac(child: HeaderNode) {
    replace(detailView, detailView(child));
  }
  action dropelement(item: String, target: String, index: String) {
    var targetNode: HeaderNode := loadHeaderNode(UUIDFromString(target));
    var itemNode: TreeItem := loadTreeItem(UUIDFromString(item));
    //check recursion
    if(canMove(itemNode, targetNode)) {
      //update depth
      if (itemNode isa HeaderNode) {
        var h: HeaderNode := itemNode as HeaderNode;
        h.depth := targetNode.depth + 1;
      }
      //update parent
      itemNode.parent.children.remove(itemNode);
      itemNode.parent := targetNode;
```

```
      targetNode.children.insert(index.parseInt(), itemNode);
      //update UI
      replace(documentTree, documentTree(getDocument(targetNode)));
      replace(statusBar, template{ "Move action persisted" });
    }
    else {
      replace(statusBar, template{ "Could not persist move action;
        it would create a cycle in the tree" });
} } }
```

Listing 15.9: The viewHeader template

Again, the source of the application is surprisingly concise. The application is defined in less than 500 LOC including whitespace. The widget library is another 400 LOC of code, from which 220 lines consists of Dojo based widgets. In the Dojo based widgets half of the lines is dedicated to JavaScript. All in all a satisfying result for a full blown rich interface application.

chapter 16 Conclusion

This document starts with the question *"Can we devise a declarative model that captures the complex asynchronous behavior and rich experience of dynamic web applications?"*. In the first part, the investigation of several development approaches used for desktop applications provides the necessary knowledge to identify problems which complicate the development of rich web application. In addition, the investigation provides an idea which notions the declarative model should support in order to take advantage of successful experiences in the development of desktop applications. Based on this research the WebDSLx extension adds several new constructions to the WebDSL language.

This thesis contributes to the WebDSL language in several ways. First the *template* constructions have been extended significantly. Generic, optional *attributes* are supported and furthermore templates can be passed on to other templates using the *with/require* syntax. In addition, both *actions* and *templates* can be inlined. Those extensions can be applied in non-Ajax applications as well. Moreover, by using WebDSLx, asynchronous behavior can be achieved by invoking *delta update operations.* Those operations generate *delta updates* to partially update pages on the fly, which results in very responsive applications. Last but not least WebDSLx turns the notion of templates into a powerful widget system. Those abstractions can be used to develop an extensive widget library for WebDSL. Many of those widgets are still applicable in non-Ajax web applications, although event triggered actions are not supported in that case.

16.1 Complex asynchronous behavior

The constructions that model *the complex asynchronous behavior* are introduced in chapters 9-11 and a corresponding case study is presented in chapter 14. The MyTodo application demonstrates that the abstractions are elegant and concise; a responsive task management application has been achieved in 200 lines of code. This applications however also shows the major problem in asynchronous interfaces: by caching parts of the user interface in the browser, an old state of a certain object might still be displayed (Section 11.2).

Chapter 6 mentions a solution to this problem: automatic constraint management might detect that an object, which is used to render a certain part of the page, has been changed in which case the corresponding part needs to be updated. The disadvantages of this approach are also explained, the most important one being the introduction of automagic behavior, which complicates debugging by introducing sometimes unexpected behavior. For example a user might be editing an object, but suddenly the form is reloaded and the potential changes are discarded because the object was changed on the server,

Keeping the page up-to-date is an issue that becomes more complicated when the developer decides to use the Ajax abstractions. Therefore this is the most leaky part of the abstraction and an interesting subject for future research. It might be useful, for example, to be able to mark templates as 'autoupdate', which allows the client to reload that template when one of the subjects of the templates has changed. However this requires the client or server to keep track of actual templates in the page to determine for which changes the client should be notified. The author expects that such behavior can be achieved automatically, however it requires solid research to collect all the complex situations which might occur in asynchronous interfaces. This is definitely an interesting subject for future research.

Besides this update drawback, which is not hard to overcome in most situations, the abstractions seem to be very applicable to build applications which support typical asynchronous interaction patterns.

16.2 Providing a rich experience

In order to enrich the user-experience of WebDSL applications, WebDSLx provides abstractions to create reusable widgets. Such a widget can be written completely by a WebDSL developer, but another possibility is to wrap an existing widget, taken from a Ajax library. The Outliner application demonstrates that this is possible and features several widgets and interaction patterns common in desktop applications, such as trees, drag and drop, tab pages etcetera.

The WebDSLx extension does not ease the development of such widgets significantly, however, it enables to put the complexity of such widget in a 'box' which can bes stored in a library, causing the widget to be reusable among many applications and accessible through a common interface.

As suggested in chapter 7, a visual interface designer might add to the attractiveness of WebDSL. A visual designer is easier to implement when the widgets have a uniform interface. Delta updates complicate the design of the user interface, since the interface is built using numerous small parts. However a visual designer designed for this purpose might still do the job, by providing the possibility to create placeholders and preview them using an existing template. Ajax oriented interface builders has been developed before [54; 56], but are far from common and for that reason an interesting research subject.

16.3 Regarding the recommendations

Chapter 7 provides several recommendations concerning the development of user interfaces. Some of them are met by WebDSL (Section 8.5) and others by the WebDSLx extension. Still, some recommendations might be considered in the future development of WebDSL.

Book-markable URls. WebDSLx provides no new abstractions for book-markable URLs. WebDSL however supports the usage of nice URLs. WebDSLx can be used to build a *Single Page Interface*, but it is recommended to use a hybrid construction; multiple pages which use small Ajax templates. This allows to combine the best of two worlds. Not only does this provide book-markable URLs, it also simplifies the update inconsistencies problem.

A interface builder and a preview function are both not offered by WebDSL nor the Ajax extension. As mentioned, this is an interesting topic for future research.

Sub-classing widgets. WebDSLx widgets have no notion of inheritance. However by wrapping a very generic widget in a purpose specific widget, similar effects can be achieved.

Styling. No styling issues are addressed in this research. However using the attribute functionality, *style* and *class* attributes can be provided for most built-in and user defined widgets.

Detect which code should be executed where. In the current implementation, every action invocation results in a server call, even when the invoked action could be executed completely at the client. Recognizing this during compilation might further optimize the asynchronous behavior.

16.4 Open issues

During the development of the WebDSLx extension, several issues which are worthy to mention were encountered.

Classic form submits. Using WebDSLx, the compiler translates all form submits to the new Ajax submit method (data is send asynchronously in the background), to ensure that the page remains available until a new page is received. However by using this method files cannot be send to the server. As a solution to this problem, setting the *classic := true* attribute on an *action* element results in a classic form submit.

No interface support. Certain complex widgets require the provided objects to support a specific interface. Since WebDSL supports no interfaces, super-classes are used for this purpose, which is undesirable since this implies restrictions for the data model of the application.

No generic JSON support. Many Ajax libraries use the *JavaScript Object Notation* as raw data type. For this reason Outliner application provides a conversion from the data model to JSON objects. Such conversion should be provided by the compiler since this would improve the interoperability, not only with Ajax libraries but also with external web services.

No real JavaScript embedding. The usage of SDF allows to intertwine multiple languages in the WebDSL language. This is already done for instance for XML and HQL. However JavaScript snippets are currently not (syntactically) checked by the compiler.

Limited typechecking. The type-checker provided by WebDSLx is limited. It would be useful for example to check which attributes are valid on certain built-in elements and which arguments are expected in event-handlers.

Validation support. Recently, advanced input validations are added WebDSL. Ajax could be used to provide quicker feedback to the client, but currently the validation abstractions do not take advantage of the possibilities offered by WebDSLx.

Debugging becomes more complex. Another drawback of WebDSLx applications is that debugging becomes more difficult. Validating the JavaScript code needs to be done client-side. So when an application does not behave as expected, it is sometimes necessary to debug the Javascript in the browser in order to determine if the proper requests are send to the server.

chapter 17 References

[1] Beshers, C. M. & Feiner, S. (1989). *Scope: automated generation of graphical interfaces*. In: (Ed.), *UIST '89: Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology*, ACM.

[2] Book, M. & Gruhn, V. (2007). *An Instant Messaging Framework for Flexible Interaction with Rich Clients*. In: (Ed.), *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, IEEE Computer Society.

[3] Da Silva, P. (2001). *User interface declarative models and development environments: A survey*, Lecture Notes in Computer Science : 207-226.

[4] Fielding, R. T. & Taylor, R. N. (2002). *Principled design of the modern Web architecture*, ACM Trans. Internet Technol. 2 : 115-150.

[5] Fowler, M. (**2006**). *GUI Architectures*, .

[6] Garrett, J. J. (**2005**). *Ajax: A New Approach to Web Applications*, .

[7] Gharavi, V.; Mesbah, A. & van Deursen, A. (2008). *Modelling and Generating Ajax Applications: A Model-Driven Approach*. In: Gaedke, M. & Bieliková, M. (Ed.), *Proceedings of ICWE'08 Workshops, 7th International Workshop on Web-Oriented Software Technologies (IWWOST'08)* , STU.

[8] Goldberg, A. & Robson, D., 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Groenewegen, D. & Visser, E. (2008). *Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns*. In: Schwabe, D. & Curbera, F. (Ed.), *Eighth International Conference on Web Engineering (ICWE 2008)*, IEEE CS Press.

[10] Hemel, Z.; Verhaaf, R. & Visser, E. (2008). *WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications*, Lecture Notes in Computer Science 5301 : 113-127.

[11] Hemel, Z. & Visser, E. (2009). *PIL: A Platform Independent Language for Retargetable DSLs*, .
[12] Henderson, Jr., D. A. (1986). *The Trillium user interface design environment*, SIGCHI Bull. 17 : 221-227.
[13] Hill, R. D. (1992). *The abstraction-link-view paradigm: using constraints to connect user interfaces to applications*. In: (Ed.), *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM.
[14] Holwerda, J. (**2008**). *Separation of Concerns in Web User Interface Design*, .
[15] Järvi, J.; Marcus, M.; Parent, S.; Freeman, J. & Smith, J. N. (2008). *Property models: from incidental algorithms to reusable components*. In: (Ed.), *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, ACM.
[16] Kasik, D. J. (1982). *A user interface management system*, SIGGRAPH Comput. Graph. 16 : 99-106.
[17] Krasner, G. & Pope, S. (1988). *A description of the model-view-controller user interface paradigm in the smalltalk-80 system*, Journal of Object Oriented Programming 1 : 26-49.
[18] Mesbah, A. & van Deursen, A. (2007). *An Architectural Style for Ajax*. In: Paulish, D.; Gorton, I.; Tyree, J. & Soni, D. (Ed.), *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, IEEE Computer Society.
[19] Mesbah, A. & van Deursen, A. (2008). *A Component- and Push-based Architectural Style for Ajax Applications*, Journal of Systems and Software (JSS) 81 : 2194-2209.
[20] Michotte, B. & Vanderdonckt, J. (2008). *GrafiXML, a Multi-target User Interface Builder Based on UsiXML*. In: (Ed.), *ICAS '08: Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, IEEE Computer Society.
[21] Myers, B. A. (1995). *User interface software tools*, ACM Trans. Comput.-Hum. Interact. 2 : 64-103.
[22] Myers, B.; Hudson, S. E. & Pausch, R. (2000). *Past, present, and future of user interface software tools*, ACM Trans. Comput.-Hum. Interact. 7 : 3-28.
[23] Nielsen, J. (1993). *Noncommand user interfaces*, Commun. ACM 36 : 83-99.
[24] Olsen, Jr., D. R. (1986). *MIKE: the menu interaction kontrol environment*, ACM Trans. Graph. 5 : 318-344.
[25] Parr, T. J. (2004). *Enforcing strict model-view separation in template engines*. In: (Ed.), *WWW '04: Proceedings of the 13th international conference on World Wide Web*, ACM.
[26] Phanouriou, C. (2000). *Uiml: a device-independent user interface markup language*, Virginia Polytechnic Institute and State University.
[27] Rosenthal, D. (1987). *A simple X11 client program, or, how hard can it really be to write "Hello World"*. In: (Ed.), *Proceedings of USENIX*, .

[28] Shneiderman, B. (1983). *Direct Manipulation: A Step Beyond Programming Languages*, Computer 16 : 57-69.
[29] Stirewalt, R. & Rugaber, S. (2000). *The model-composition problem in user-interface generation*, Automated Software Engineering 7 : 101-124.
[30] Szekely, P. A.; Sukaviriya, P. N.; Castells, P.; Muthukumarasamy, J. & Salcher, E. (1996). *Declarative interface models for user interface construction tools: the MASTERMIND approach*. In: (Ed.), *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, Chapman & Hall, Ltd..
[31] T. Berners-Lee, R. C. (**1990**). *WorldWideWeb: Proposal for a HyperText Project*, .
[32] Thomas, J. J. & Hamlin, G. (1983). *Graphical input interaction technique (GIIT)*, SIGGRAPH Comput. Graph. 17 : 5-30.
[33] V. Bullard, K. S. & Daconta, M., 2001. *Essential XUL Programming*. Wiley, .
[34] Vander Zanden, B. (1996). *An incremental algorithm for satisfying hierarchies of multiway dataflow constraints*, ACM Trans. Program. Lang. Syst. 18 : 30-72.
[35] Visser, E. (2004). *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9*. In: Lengauer, C. & others (Ed.), *Domain-Specific Program Generation*, Spinger-Verlag.
[36] Visser, E. (1997). *Syntax Definition for Language Prototyping*, University of Amsterdam.
[37] Visser, E. (2008). *WebDSL: A Case Study in Domain-Specific Language Engineering*. In: (Ed.), *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Springer.
[38] . *The Common Gateway Interface*, http://hoohoo.ncsa.illinois.edu/cgi/overview.html (1993).
[39] . *Extensible Markup Language (XML) 1.0*, http://www.w3.org/TR/1998/REC-xml-19980210 (1998).
[40] . *JavaScript Object Notation (JSON)*, http://json.org/ (1999).
[41] . *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*, http://www.w3.org/TR/xhtml1/ (2000).
[42] . *Burton Group annual survey*, http://www.surveymonkey.com/sr.aspx?sm=fXLiKcnKlD6cO5bRe961aBB6NCCWytRyY3rParAYmwA_3d (2007).
[43] . *Owasp Top Ten 2007*, http://www.owasp.org/index.php/Top_10_2007 (2007).
[44] . *XML Binding Language (XBL) 2.0*, http://www.w3.org/TR/xbl/ (2007).
[45] . *Backbase - Enterprise Ajax Widgets*, http://www.backbase.com/products/enterprise-ajax/ajax-widgets/ (2009).
[46] . *Mendix - Leader in Model-driven Application Delivery*, http://www.mendix.com/ (2009).
[47] . *Dojo - the Javascript toolkit*, http://www.dojotoolkit.org/ (2009).
[48] . *Firebug web development tools*, http://getfirebug.com/ (2009).

[49] . *Standard ECMA-357*, http://www.ecma-international.org/publications/standards/Ecma-357.htm (2009).
[50] . *HTML / XHTML Standard Event Attributes*, http://w3schools.com/tags/ref_eventattributes.asp (2009).
[51] . *Announcing Atlas*, http://cappuccino.org/discuss/2009/02/28/announcing-atlas/ (2009).
[52] . *Visual Ajax Builder – Code named Telamon*, http://www.backbase.com/products/visual-ajax-builder/ (2009).
[53] . *XML Path Language (XPath) Version 1.0*, http://www.w3.org/TR/xpath (2009).
[54] . *Google Web Toolkit*, http://code.google.com/webtoolkit/ (2009).
[55] . *Ext JS: Cross-Browser Rich Internet Application Framework*, http://extjs.com/products/extjs/ (2009).
[56] . *Smarty template engine*, http://www.smarty.net (2009).
[57] . *Adobe Dreamweaver CS 4*, http://www.adobe.com/products/dreamweaver/ (2009).
[58] . *RFC1122 Requirements for Internet Hosts -- Communication Layers*, http://tools.ietf.org/html/rfc1122 (2009).
[59] . *RFC2068 Hypertext Transfer Protocol -- HTTP/1.1*, http://tools.ietf.org/html/rfc2068 (2009).
[60] . *HTML 4.01 Specification*, http://www.w3.org/TR/html401/ (2009).
[61] . *Internet Explorer box model bug*, http://en.wikipedia.org/wiki/Internet_Explorer_box_model_bug (2009).
[62] . *XML User Interface Language (XUL) 1.0*, http://www.mozilla.org/projects/xul/xul.html (2009).
[63] . *There is no data. There is only XUL. (Official XUL namespace)*, www.mozilla.org/keymaster/gatekeeper/there.is.only.xul (2009).
[64] . *RDF/XML Syntax Specification (Revised)*, http://www.w3.org/TR/rdf-syntax-grammar/ (2009).