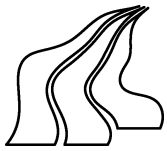
Aalborg University
Department of Computer Science



Database and Programming Technologies

Title:
Server Side LAML Framework

Topic:
Programming Technologies

Project Period:
4/2-2002 – 14/6-2002

Project Group: Dat6, D601A
Mikael M. Hansen
Paw Iversen
Jimmy Juncker

Supervisor:
Kurt Nørmark

Number of appendixes: 2

Total number of pages: 143

Number of pages in report: 112

Number of reports printed: 7

Abstract:

In this project four problems found in the area of Web applications development are analysed. These problems are named the *State handling* problem, the *Validation* problem, the *Complex forms* problem and the *Reusability* problem. Existing work in the area is analysed. Based on these problems and the existing solutions three hypothesis are presented as the goal of this project. During the design, solutions to the problems are developed. Example Web applications are implemented to illustrate the usage of the designed solutions. Based on these Web applications we conclude on the project. Introducing the session concept inspired by Bigwig solves the *State handling* problem. Additional experiences are still needed to fully evaluate the *Reusability* problem. The *Complex forms* problem is solved by the designed solution, based on building an object structure representing an HTML form. The implemented solution relies on the session framework, but a CGI based solution is considered possible. The *Input validation* problem is solved by relying on the session framework. Validation is made available on both the object and the page level. It is concluded that relying on the session framework have the advantage that validation functions are available when they are needed. This is not the case in a CGI solution. Improvements can be made on various places in the solution. This is recommended as future work.

Resume

I denne rapport præsenterer vi vores arbejde gennem specialet. Dette arbejde ligger i forlængelse af vores Dat5 arbejde. Vi arbejder med Web udviklingen indenfor programmeringssproget Scheme med LAML bibliotekerne. I modsætning til den gængse måde, hvor flere sprog kombineres for at opnå den ønskede fleksibilitet, tillader en programmatisk tilgang til Web udvikling, at både præsentation og dynamik foregår i det samme sprog. I løbet af Dat5 udviklede vi et modul til Apache Web serveren. Dette modul bruges som en del af konteksten til dette arbejde.

Vi begynder med en analyse af nogle ofte forekommende problemer man støder på som Web udvikler. Et af disse problemer er *Tilstands håndtering*, som består af to delproblemer. Det første delproblem er data håndtering mellem klient og server. Et andet delproblem omhandler kontrol-flowet af en Web applikation. Mulige løsninger der findes i dag bliver analyseret og vurderet. Det andet ofte forekommende problem i Web udvikling er *Input validering*. Der er to alternativer som oftest bruges. Enten klient side validering ved hjælp af JavaScript eller server side validering ved hjælp af det sprog som Web applikationen er skrevet i. Fordele og ulemper ved de to alternativer overvejes og vurderes. Det tredje problem der analyseres og beskrives er problemet med opbygningen af *Komplekse strukturer* på serveren, samt manglen af samme struktur når den har været vist til klienten. Det sidste problem der bliver behandlet er *Genbrugeligheds* problemet. Som udvikler er man vant til en vis grad af genbrugelighed, men primært på funktionalitet niveau. Ved fremkomsten af side centrerede teknologier som PHP, ASP og JSP er en lille grad af genbrugelighed blevet kutyme, men det er ikke noget der finder sted i stor stil. Ønsket om en øget grad af genbrugelighed er fremsat.

Efter præsentation og gennemgang af problemerne kigger vi nærmere på relateret arbejde. Formålet med dette er todelt. Både at finde inspiration til løsninger til problemerne, og finde teknologier som allerede har løst dele af problemerne. På baggrund af analysen af problemer og det relaterede arbejde, fremsættes et antal hypoteser, der ligger til grund for det videre arbejde.

Efter analysen præsenteres designet, hvor vi med udgangspunkt i det enkelte problem præsenterer vores overvejelser omkring dette problem. Ud over overvejelser præsenterer vi også den designede løsning som skal ligge til grund for en implementation. Vi begynder med problemet omkring *Tilstands håndtering*. Vi præsenterer tre forskellige løsninger, og konkret vælges at designe en løsning inspireret af Bigwig. Denne løsning involverer introduktion af et primitiv der pauser evalueringen af Web applikationen når en side vises til klienten. Løsningen involverer også introduktionen af et primitiv der introducere et leksikalsk scope i et program. Disse to primitiver er den del af sessions begrebet som også medfører en løsning

til *Genbrugeligheds* problemet.

Til behandling af problemet omkring *Komplekse strukturer*, overvejes både en tilgangsvinkel der involverer specifikation af nestede lister, samt introduktion af et indlejret domæne specifikt sprog til løsning af problemet. Da vi finder en objekt orienteret tilgangsvinkel til problemet er den bedste, designes en løsning baseret på objekt orienterede principper. Composite design patternet bruges som inspiration til introduktion af forskellige klasser, der repræsentere forskellige elementer i en objekt struktur. Denne objekt struktur indeholder mulighed for associering af en præsentation med det enkelte objekt i strukturen. Ydermere kan den indeholde data som er associeret med de enkelte objekter i strukturen. Slutteligt behandles problemet omkring *Input validering*. Server side validering betragtes som den rigtige vej frem, selvom visse ulemper findes. Derfor designes en løsning til validerings problemet der passer sammen med resten af den designede løsning. Dette tillader validering både på enkelte sider, samt på komplekse strukturer.

Dernæst præsenteres overvejelser og problemer som vi har arbejdet med på det lave niveau (omkring server modulet) i forbindelse med implementationen. På det lave niveau kommer vi også med enkelte anbefalinger for teknologier som vi mener kan bruges til en endelig implementation. Nogle begrænsninger bliver truffet, for at forhindre at vores arbejde skulle skifte fokus fra det høje niveau (Scheme, LAML) til det lave niveau (Apache/C). Efterfølgende introduceres nogle proof of concept applikationer, hvilket illustrerer og motiverer den designede løsning. Disse bliver gennemgået for at give læseren en forståelse af den implementerede løsning. På baggrund af erfaringer med disse præsenteres vores overvejelser med hensyn til brugbarheden af den designede løsning. Anbefalinger, og ting der efter vores mening skal laves anderledes, bliver gennemgået og diskuteret, så videre arbejde indenfor dette område kan drage nytte af vores arbejde.

Slutteligt konkluderes der på de fremsatte hypoteser og de designede løsninger. Vi vurderer i hvor høj grad vi har opfyldt vores mål for dette projekt, og eventuelle afvigelser er forklaret. Vi konkluderer at introduktionen af sessions begrebet, generelt set løser problemerne med *Tilstands håndtering*. Introduktionen af et sessions primitiv tillader udvikleren at få et overblik over hele applikationen, da flere interaktioner med klienten foregår i det samme leksikalske scope ved hjælp af `slaml-show` primitivet. *Genbrugeligheds* problemet formodes løst. Vi må konkludere at en længere periode til evaluering at dette er nødvendig. Problemet omkring *Komplekse strukturer* er løst fornuftigt ved brug af objekt orienterede principper. Selvom de objekt orienterede principper ikke passer godt i konteksten af dette arbejde (Scheme og funktionel programmering) så er det rimeligt let og fleksibelt at arbejde med objekter til repræsentation af komplekse strukturer. Med hensyn til *Input validerings* problemet så kan vi konkludere, at vi har lavet et validerings apparat der passer godt sammen med sessions apparatet. At bygge validering på sessions apparatet sikrer, at det at få data og validere det er en atomar handling, i modsætning til CGI hvor det er to separate handlinger.

Preface

This report documents our Dat6 semester project at the Department of Computer Science, Aalborg University, Denmark. The Dat6 semester is the semester where we complete our master thesis. Preparatory work for the master thesis is our Dat5 project, where we worked in the area of Web application development using LAML.

Report Conventions:
Throughout this report all references to the bibliography are shown as [reference]. Special concepts are written in abbreviation followed by a full length name in parenthesis, the first time encountered. Through the remainder of the report the abbreviation is used. References to figures and tables are written like x.y where x is the number of the chapter and y is the number of figure or table in the given chapter. E.g. Figure 2.3 is figure three in chapter two. Text in figures are written in a special font to make it distinguishable from the normal text. An example of this font is this sentence. *When referring to specific contents of figures an italic notation is used, like this sentence.* Italic notation is also used when referring to the named problems throughout the report. Primitives that are part of the SLAML framework are written in `this font` to distinguish them from the rest of the text. Throughout the report the word he, will refer to he or she. A gray box is used to give the implementation considerations regarding the various primitives in the design. The contents of this is targeted at the reader familiar with Scheme. The first time a primitive is introduced a parenthesis will follow with a page number. This page number refers to the page in the SLAML reference - in Appendix A - containing a detailed description of the primitive.

The project period began February 4, 2002 and lasted to June 14, 2002.

Aalborg University, June 14, 2002.

Mikael M. Hansen

Paw Iversen

Jimmy Juncker

Contents

Introduction

The task of developing Web applications has become more important during the recent years, as the use of Web applications has become more common. Following the increase in use of Web applications, focus has increased on inventing technologies and practices for improving the efficiency of a Web application developer. At the same time focus on expanding the possibilities of the technologies used today has increased. This ongoing task of improving, and expanding the possibilities in the domain of Web applications development is possibly one of the fastest growing areas in computer science today.

Some of the most interesting tendencies in Web applications development is the use of XML[W3C02a] as a uniform way of sharing and distributing data. Other important aspects of the fields of Web applications development is the J2EE[Inc01] architecture introduced by Sun Microsystems. J2EE includes several new technologies that are intended to aid Web applications developers in their task of creating Web applications.

Apart from the mainstream tendencies people work on various niches that better suit their needs. Niches that rely on more specialized technologies for Web development. Examples of such niches is the Bigwig language, or the WASH/CGI library[Pet] for Haskell[JO02]. Another niche is introduced by Kurt Nørmark, as he has created the LAML libraries[lam01] for the Scheme programming language[KCR+98]. This is done since he finds that the functional paradigm fits well into the development of Web applications[Nør00]. Furthermore the syntactical nature of lisp languages fit well with Web development

During the preparatory work[MPJ02] for this master thesis we worked in the area of Web applications written in LAML. Server side LAML (SLAML) was introduced, as the possibility to execute LAML applications on the server without using CGI. This was achieved by creating an Apache server module, which we named mod_laml. mod_laml allow executing Web application written in Scheme using the LAML libraries roughly twice as fast as it is done by relying on CGI[cgi01]. The work conducted during the preparatory work for this thesis was mainly on a low level. Most of the work consisted of creating the Apache module, including various features often present in an Apache server module. A second aspect of the preparatory work was to discover new ideas and principles that help to make Web development easier. We concluded that we would conduct a further analysis of some of these concepts and possibly implement them using mod_laml. As a continuation of this strategy, the focus has been shifted from the low level to a higher level, namely from the C level to the Scheme level. In this project we focus on providing new aspects into the area of Web applications development by using mod_laml as the basis for further development relying on Scheme and the LAML libraries.

This project addresses some of the often encountered problems in Web applications development relying on todays technologies. This is done in four steps. First, often encountered problems when working with Web development are found. Second, analysis of the problems and possible solutions are conducted, while at the same time considering aspects of new technologies used for Web applications development. Third, solutions to the problems are discussed and designed to integrate with the context created and motivated during our preparatory work for this master thesis. Fourth, a number of example Web applications are developed to illustrate the solutions to the problems in Web applications development. The example applications are used as the motivation for discussions and reflection on the problems, their solution and recommendations for further development in the area of Web applications development in LAML and mod_laml are presented last.

1

Analysis

Contents

In this chapter an analysis of problems related to Web development are conducted. Next is an introduction to different approaches to Web development. Following this is an analysis of work related the problems described. The solutions used in related work - in the area of session based approaches - are presented and discussed. Finally a problem definition including hypothesis regarding the goals of this project is presented.

1.1 Problems in Web Development

We concluded our previous work with the fact, that we will attempt to make Web development easier for the developer. Making Web development easier for the developer, is done by introducing abstractions in the language used for development, and by introducing tools that supports the developer when solving often encountered problems. This section presents problems that are encountered when developing Web applications. Four problems have been identified based on our knowledge with the development of Web applications, and they are:

1. State handling
2. Input validation
3. Complex forms
4. Reusability

The following will give a short introduction to the problems. In addition each of the problems are presented and discussed in greater detail in its own section, see Section 1.1.1, 1.1.2, 1.1.3 and 1.1.4.

The first problem (*State handling*), is based on the characteristics that a Web application must underlay the stateless nature of the HTTP protocol. An application build on CGI, actually exist of an amount of small "applications". Each small "application" corresponds to the execution of a single CGI script, which results in the presentation of a single page. This observation is build on the fact that the processing of a single request to a CGI based Web application corresponds to execution of one CGI script. The developer of Web applications will be aware that more requests corresponds to more CGI scripts. He will therefore have to concentrate on the development of many small "applications" that must interact with each other, instead of focusing on the whole Web application as one unit. If data and information about state must survive more requests, it must be handled explicit by the developer. The reason for this is the stateless nature of the HTTP protocol. The need to explicit handle data and information about state is seen as a problem. This problem is named *State handling*, and it is discussed in greater detail in Section 1.1.1.

The second problem is named *Input validation* and is about validating data submitted by a client. When a client submits data to a CGI script, it has to be validated in order for the data to be valid in the context it is used. The problem is present because all data submitted to a script is received as strings. Since not all operations are done on strings (e.g. adding two numbers), it is often necessary to perform checks on the input from the user. This is in most cases done explicitly by the developer. However, validation imposes problems, since the validation process is error prone if the developer is not systematic. *Input validation* is discussed in greater detail in Section 1.1.2.

The third problem is related to the way data is structured in HTML. When writing CGI scripts, data can be placed in data structures to raise the level of abstraction. This is done by using primitives available in the programming language used to write the CGI script (e.g. arrays, tree structures, hash tables). A similar structure can be modeled as layout in HTML (an array of records can e.g. be modeled as a table with each record presented as a row), but a problem exist when data is received from the client. The developer can present complex structures in HTML forms, with respect to layout, but when data is received from the client, the data structures are lost. This is the case, since information from an HTML form is encoded as a string containing key/value pairs. The relation between recreation of large and complex structures and forms based on a key/value pairs, is seen as a problem. We have named this problem *Complex forms* and it is discussed in further detail in Section 1.1.3.

The fourth problem considered is the problem of reusability of program units that are larger than one page. In CGI, the developer can create functionality, which can create parts (often single pages) to a Web application. However, in order to reuse a whole Web application, all pages related must be included. The problem is that a Web application in CGI is not equal to a single unit, but instead a series of pages. We see the lack of considering an application as a single unit as a problem, which we have named the *Reusability* problem. Reusability is

discussed in Section 1.1.4.

To summarize, four problems - *State handling, Input validation, Complex forms and Reusability* - that are present when developing large Web applications have been identified. The following sections specifies the problems in greater details and gives examples that presents the nature of the specific problem. Current solutions to the problems will be presented, but considerations and choices regarding the solutions are presented in the Design chapter (see Chapter 2).

1.1.1 State Handling

When a developer uses the CGI protocol to write Web application, difficulties regarding the CGI protocol arises. The main problem with the CGI protocol is that it is stateless (because of the stateless nature of the underlying HTTP protocol). This means that state information have to be handled explicit by the developer in order to maintain state. A second problem is that the CGI protocol dictates that a program written using CGI must end after a response is send to the client. There is no possibility of writing the Web application as one program and rely on the interactions with a user returning control to the surrounding code, as it is done in non-Web related programming. This has some consequences that are explained in this section.

The *State handling* problem can be divided into two subproblems. The subproblems are presented below. After the presentation of the two subproblems, current solutions are presented.

1. Data flow handling
2. Control flow handling

Data flow handling concerns the need for the developer to explicit handle the data (values) already received from the client. Consider an application, that consists of three pages. The first two pages each take an input, and the third page presents the input entered. Since the HTTP protocol is stateless, the data from the first page must explicit be stored - or sent to the next page - by the developer. This is needed for the data to be present after the request to the second page, so it is available when the third page is presented. The problem is illustrated in Figure 1.1.

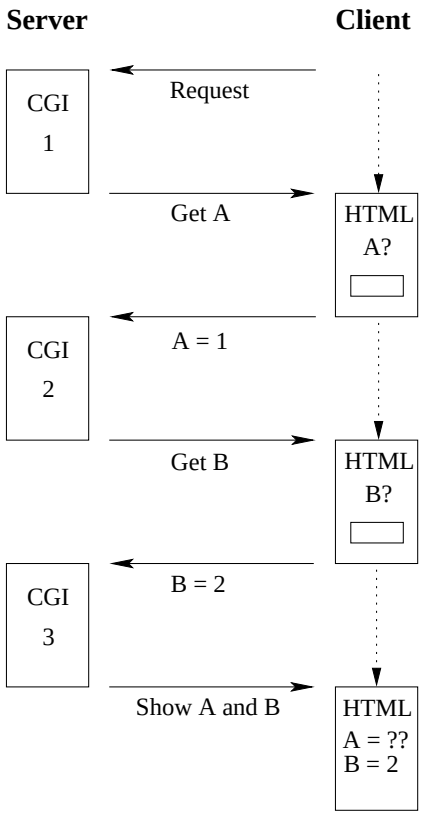


Figure 1.1: The *Data flow handling* problem. Since each of the CGI scripts are handled as a single request, the data cannot "survive" more requests. In this example, the *A* value is not present after the response from the second CGI script (*CGI 2*).

The *Control flow handling* problem is also (like *Data flow handling*) related to the stateless nature of the HTTP protocol. Figure 1.2 presents an example, where multiple choices can be made through the execution of a Web application. When the application is running, it is not possible to determine the current position, of all the positions in the application. Considering the figure, it is e.g. not possible to determine if *D* has been visited if the current page is *C*. The reason for this is, that each page is presented by the execution of a single script, and each script terminates after each request. This means, that each page is presented without returning to the specific point in the application from where it was called. Information needed to maintain the interaction between the different parts of the application, must therefore be handled explicitly by the developer. This is typically done by associating the next script to be invoked with a button or a link on the current page.

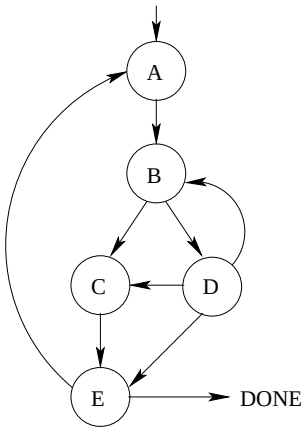


Figure 1.2: Each node in the tree represents a page (e.g. a single CGI script) and an edge symbols a possible selection. The collection of all the pages is the entire application and a path through the tree structure, represents a possible execution of the application.

The two subproblems related to *State handling* have been presented. Current solutions to these problems are presented and discussed in the following.

Current Solutions to Data Flow Handling

The *Data flow handling* problem can be solved by either storing the data on the client or on the server. The two alternatives are presented in Figure 1.3, which is based on Figure 1.1.

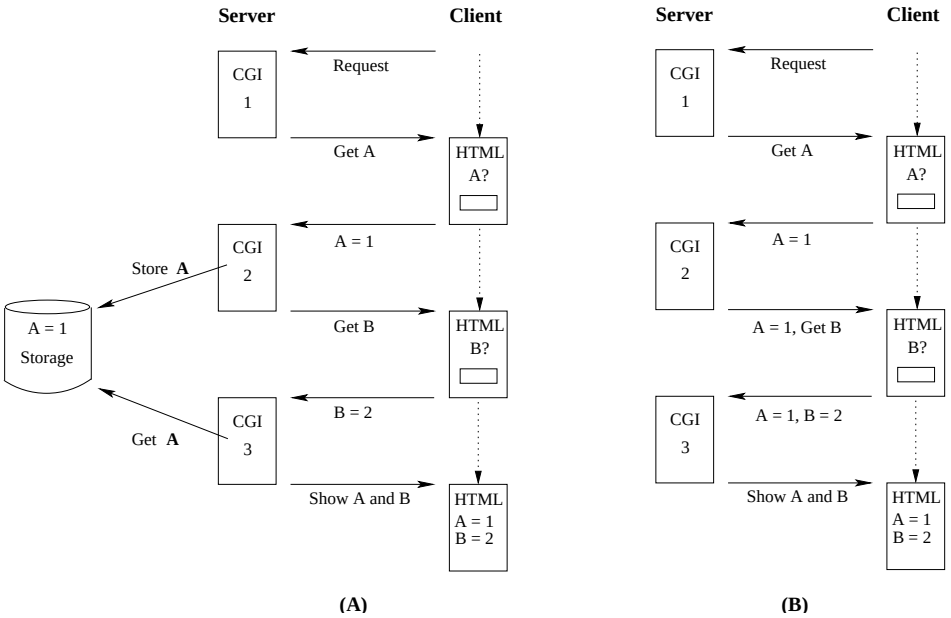


Figure 1.3: *(A)* relies on the possibility to make data persistent on the server, whereas *(B)* sends data to the client.

The solution presented in Figure 1.3 *(A)* relies on the possibility to store data on the server. This can e.g. be in files on the local server's filesystem. However, it must be possible to distinguish between the different clients. If this is not possible, the information stored on the server, cannot be related to a specific client. Many page centered approaches to Web development (PHP [The02], JSP [SM02], ASP[asp01] etc.) let the developer specify which variables must be accessible at a later point in the dialog with the client. The client is given an id to identify it from the other clients. The id is the only information sent to the client at the end of each request (by storing it in a hidden `input` element in a way described later). All the registered variables are serialized and written to a file on the server without the developer having to know about it.

A problem with the solution presented in Figure 1.3 *(A)*, is that the client can stop his interactions with the server and bookmark the current page. After a while (minutes, hours, days or even longer) a client can return - by using the bookmark - in order to complete the application once left. There is no way to determine how long the client will be "idle" in the execution of the application, or if the client will ever return. If the client never returns, data related to the interactions with the server will take up space even if it is never used. If large applications are running on a Web server, storing never used informations is a problem, that must be considered.

Figure 1.3 *(B)* presents a solution to *Data flow handling*, where data is send to the client in hidden `input` elements when needed later. This means, that data can be extracted from the HTML page in the same way as data from other `input` elements. The only difference is that data in hidden `input` elements are not shown in the layout of the HTML page. If many values are needed at a later point, it is a cumbersome task for the developer to create a hidden `input` element for each value. Persistent.pm [Pra02] is a Perl module that can generate URL strings or hidden `input` elements from variable names. In this way the data can be sent to the client and back in the normal way, but the developer just has an easier task of doing it. We find this abstraction useful, since the developer can access the values in the variables, without the need to handle the data from the HTML form explicit.

Sending data to the client in hidden `input` elements is a problem, since the data is accessible to the client. It is not presented on the HTML page, but it is present in the page source. A client can modify the values in the hidden `input` elements and submit them. This can result in the server receiving unexpected or invalid information. Security is also an issue here. Imagine that the developer changes the unique id which identifies the client (created to distinguish between clients, so data can be stored on the server). If a client guesses (or knows) the unique id of another client, it can pretend to be that client. This allows access to information not related the clients own interactions with the server.

Current Solutions to Control Flow Handling

This section presents two solutions to handle the flow in a Web application. The first solution is based on primitives in the programming language used. The entire application (or larger parts of it) is placed in a single file, and when a request is received, a variable (maintained

by the developer) determines which page to show. Figure 1.4 shows how a `cond` special form from Scheme can be used for this.

```
(cond
 ((string=? page "page1")
  (display
   (html
    .
    .
    (input 'name "page" 'type "HIDDEN" 'value "page2"))))
 ((string=? page "page2")
  (display
   (html
    .
    .
    (input 'name "page" 'type "HIDDEN" 'value "page3"))))
 ;etc
 )
```

Figure 1.4: An example of how the `cond` special form from Scheme can be used to control the flow of a CGI program. *page* is a string extracted from the HTML form. The *"page1"*, *"page2"* and *"page3"* strings are used to determine the parts of the application and is maintained by the developer.

By using the above solution, the developer has overview of all pages (and the flow between them) in the Web application. However, if a large application is created, the gathering of all pages in the same file, will make it difficult for the developer to maintain the overview. This is the case, both because of the amount of lines of code present in the file, but also because there exists no grouping of pages which are related to a specific part of the application. In an online bookstore, for instance, the login page is not related to the page where a user can search for a specific book.

The second solution to the control flow problem is to rely on the `action` attribute from the HTML `form` element. Instead of placing the application in a single file, it is placed in a number of files (e.g. one file for each page in the application). The pages can then - by using the `action` attribute of the HTML `form` element - be linked together. The connections between the pages presented in Figure 1.2 can then be establish if page A (a single file) contains a `form` element that links to page B. It will look like:

```
(form 'action "B.cgi" ... )
```

Page B must then contain two `form` elements: one that links to page C and one that links to page D. By following this pattern, the entire connection between pages shown in Figure 1.2 can be created.

The linking between individual scripts allows the developer to create a splitting of the application (e.g. one script per page). The flow of the program is difficult to maintain, since it requires the developer to open various files in order to follow a specific series of actions (one file has a reference to one or more of the other files related to the application, which again has one or more references etc.).

In this section approaches to the two subproblems of the *State handling* problem, namely the *Data flow handling* and the *Control flow handling* problem were presented. The *Data handling problem* can be solved by storing the data on either the server or the client. The *Control flow handling* problem can be solved by using primitives in the language or by linking small applications together using the `action` attribute of HTML `form` elements. The next section focus on the *Input validation* problem and possible solutions to it.

1.1.2 Input Validation

In this section the *Input validation* problem is analyzed. In order to get information from a client in a Web application, input elements are used. These input elements are added to an HTML page by using HTML elements - often the HTML `input` element (others, like the `textarea` element, can also be considered input elements). There exists ten different types of the HTML `input` element [W3C02b], and the content of each element is in HTML handled as textual input. This means, that the representation of all values - regardless of the input type - are string values when extracted from an HTML form.

To see the problem with the lack of types in HTML, consider the following example, which consists of two HTML pages with HTML `input` elements (see Figure 1.5). The HTML `input` elements are created in basically the same way (the `name` attribute will differ):

```
(input 'type "TEXT" 'value "" 'size "5" 'name "a")
```

As seen, the type of the HTML `input` element used is `TEXT`. This means, that text `input` elements are used to gather information about numbers. In this example, the first page takes two "numbers" as input, and it contains a submit `input` element (Figure 1.5 *(A)*). The second page presents the sum of the two "numbers" entered in the first page (Figure 1.5 *(B)*).

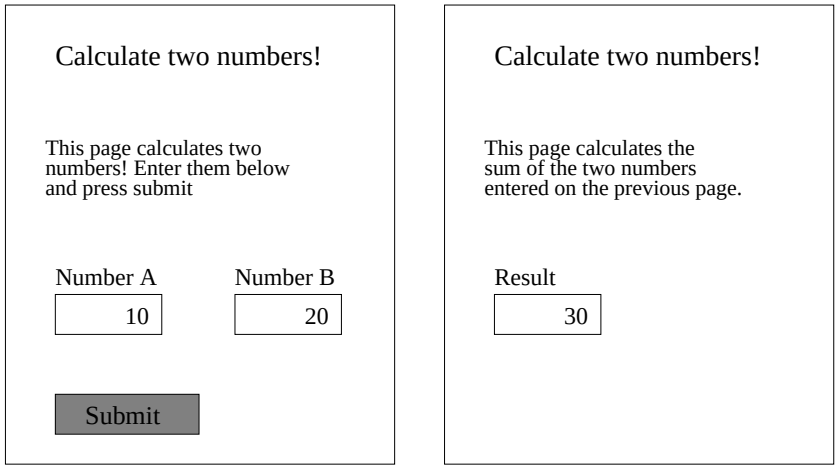


Figure 1.5: An example of input fields. The sum of the two numbers entered in *(A)* is presented in *(B)*.

The problem in this example, is that it cannot be assured, that the client enters numbers in the `input` elements on the first page. Since information from the `input` elements are handled as strings, the client can enter e.g. *"asdf"* as the first number. This gives the developer of a Web application the need to perform checks on input, since it will not - in the example used - make sense to add a string to a number.

On the basic HTML/CGI level, there is no way of checking for specific input types. If validation is needed, the developer must use other technologies. The validation part of an application can be handled on two different levels; on the client or on the server.

Generally, validating input on the client side will mean faster validation. The reason is, that data will stay on the client until it is valid. If validation is handled by the server, input must be send between client and server, until it has been validated. A problem with client side validation is, that a specific technology (such as JavaScript[Net02]) must be present on the client. If the technology is not present, a validation will not be accomplished. With server side validation, however, it is not required for the client to have specific technologies present. Instead, technologies present on the server are used for validation.

Client Side Validation

Often used technologies for client side validation are JavaScript[1] and JScript[Mic02b] (we focus on JavaScript), which are scripting languages developed by Netscape and Microsoft, respectively. Both are standardized as ECHMAScript[ECH02], which is a standard for scripting in a host environment. JavaScript can be used inside HTML documents, and the Web browser will execute the script immediately or at a later event (e.g. when a client submits

[1]Server side JavaScript exist, but when we use the term JavaScript we mean client side only

an HTML form or changes the content in an `input` element). This gives the developer many possibilities, and one of them is validation of `input` elements. The first page from the example above, is shown in Figure 1.6, with JavaScript included (body text and layout is not included).

```
<html>
 <head>
  <title>Calculate two numbers!</title>
  <script type="text/javascript">
   function checkNumbers() {
    var anum = document.sum.a.value //value from 'a' input field
    var bnum = document.sum.b.value //value from 'b' input field

    //check if any is not a number .. '10' is the radix
    if (isNaN(parseInt(anum, 10)) || isNaN(parseInt(bnum, 10))) {
        alert("You must enter valid numbers!"); //Show error alert
        return false; //the user must try again
    }
    else
      return true; //accepted .. continue
   }
  </script>
 </head>
 <body>
  <!-- body text and layout is not included! -->
  <form name="sum" type="GET" onSubmit="return checkNumbers()"
  'action="result.html">
   <input type="text" value="" size="5" maxlength="5" name="a">
   <input type="text" value="" size="5" maxlength="5" name="b">
   <input type="submit" value="submit">
  </form>
 </body>
</html>
```

Figure 1.6: An example of input validation in JavaScript. The result page is not called unless the information entered is validated as numbers.

As seen in the figure, the `input` elements are placed inside a `form` element. This makes it possible to specify in the JavaScript (included in the `script` element), which values that are of interest (as it is done in the first two lines of the *checkNumbers* function). The example also shows, that there is a problem related to the need to master two technologies; a scripting language with a C/Java like syntax together with HTML. Another problem with JavaScript is, that some functionality differs between different browsers (e.g. Netscape and MS Internet Explorer). This means, that some functionality must be browser specific and therefore written twice.

Server Side Validation

Leaving input validation as a server task, means that input from the client is sent to the server in order to be validated. If the input is not valid, appropriate errors messages must

be presented. There is no special technology used for server side validation (like JavaScript is used for client side). Instead, the programming language used to generate the HTML content is also used for validation (like VBScript [Mic02c] in ASP, Java[Inc02b] in JSP etc).

The following example is implemented in JSP, and is - like the client side validation example - an implementation of the first page in the initial example. The idea is, that we are interested in creating a loop between server and client, that runs until the client submits valid input. Such a loop can be created by using a wrapper page, which performs the needed validation. If the input is valid, the client is directed to the next page. Otherwise, the client is send back to the original page, so input values can be entered again. Another possibility is to embed the validation into the presentation. The latter is used in the example, which can be seen on Figure 1.7[2].

```
<html>
 <head>
   <title>Calculate two numbers!</title>
 </head>
 <body>
   <form action="" name="sum" method="GET">
    <%
    // if parameter is not present, getParameter returns null
    String astring = request.getParameter("a");
    String bstring = request.getParameter("b");
    // the initial request (no paramters exist), will result in both being null
    if (astring == null || bstring == null)
      ; //dont do anything
    else {
      try {
        int anum = Integer.parseInt(astring);
        int bnum = Integer.parseInt(bstring);
        response.sendRedirect("result.html?a="+anum+"&b="+bnum);
      } catch (NumberFormatException e) {
   %>
        <b>You must enter valid numbers!</b>
   <%
      } // end of try-catch
    }
   %>
   <input type="text" value="" size="5" maxlength="5" name="a">
   <input type="text" value="" size="5" maxlength="5" name="b">
   <input type="submit" value="submit">
  </form>
 </body>
</html>
```

Figure 1.7: An example of server side validation. Here the server side language is embedded in the page.

[2]The example is very basic, and no specific strengths from JSP are used.

As seen on this figure, the error message and the validation are embedded in the HTML. Special tags are used - `<%` and `%>` - to escape from HTML into the programming language used (Java in this example). The server converts the entire page into a small Java program, which is executed when requested (and compiled at the very first request). The appropriate HTML page is then created by a series of `System.out` statements. Just as with client side validation, the developer must master both a programming language and HTML. However, the programming language available on the server, will in most cases be more comprehensive than the scripting language used for client side validation.

This section has presented two different approaches to the *Input validation* problem. The first is client side validation which requires special technology on the client. The second is server side validation, where the programming language for generating Web pages are used to perform check on the input from the user. Client side validation yields faster evaluation, but the client can disable the functionality, that validates the input. In order to perform server side validation, a client/server loop must be maintained by the developer. However, server side validation ensures validation of client data. The next Section presents the *Complex forms* problem.

1.1.3 Complex Forms

In this section the *Complex forms* problem is discussed. The problem concerning complex forms exists whenever the developer has a complex structure, that it is of interest to get filled with data entered by a client. The structure can be presented as layout in HTML (an example is given in Figure 1.9). After the client has entered the information wanted, the HTML form is submitted. When the information is submitted it is converted to a key/value pairs string. This makes it difficult to recreate the structure as it was presented on the HTML page. The reason for this is, that the string does not contain any information about composition of elements in the structure, but only information about values from the basic input elements in the HTML form.

Consider the following example of a person structure, which must be filled with information from the client.
The person structure can be considered a record structure from the Scheme programming language. As seen, the *person* has nested records, like e.g. the *street-name*. A *street-name* is part of a *street*, which again is part of an *address*, which again is part of a *person*. In order to get such a structure filled with information from the client, the developer must complete two steps:

1. Present the structure in an HTML form

2. Reconstruct the structure from a key/value pairs string

The first step - presenting the structure in a complex form - can be handled by using HTML elements. Various possibilities exist, like labels, input elements, various fonts, tables etc.

```
(person
 (name
  (first-name "")
  (last-name ""))
 (address
  (country "")
  (city
   (city-name "")
   (postal-number ""))
  (street
   (street-name "")
   (house-number ""))
  )
 (email "")
 (phone "")
 (age "")
 )
```

Figure 1.8: A structure representing a person.

The *address* part from the structure shown in Figure 1.8 can be presented in an HTML table like illustrated in Figure 1.9.

```
(table
 (tr (center "Address")
      (td "Country" (br) (input 'type "TEXT" 'name "country") 'align "center")
      (td
       (table
        (tr (center "City")
            (td "City-name" (br)
              (input 'type "TEXT" 'name "city-name") 'align "center")
            (td "postal-number" (br)
              (input 'type "TEXT" 'name "postal-number") 'align "center")
            )
        'border "1")
       )
      (td
       (table
        (tr (center "Street")
            (td "Street-name" (br)
              (input 'type "TEXT" 'name "street-name") 'align "center")
            (td "House-number" (br)
              (input 'type "TEXT" 'name "house-number") 'align "center")
            )
        'border "1")
       )
      )
 'border "1")
```

Figure 1.9: The HTML layout of an address from a person record.

By writing tables inside tables, the developer can create a tree presentation of the person structure. Each of the leafs are `input` elements, that the client can fill with information.

The difficulties emerges when the developer receives the form information submitted by the client and must reconstruct the person structure (the second step). Consider the entire *person* being build in a similar way as the *address* part. When the client presses the submit button, the information from the `input` elements are gathered in a key/value pairs string. All information from the HTML form will be the string (// is added as a line break for readability):

```
first-name=nick&last-name=hansen&country=denmark&city-name=thy&postal-number=1234& //
street-name=highroad&house-number=42&email=nick%40freemail.com&phone=12345678&age=42
```

And as an association list in Scheme:

```
(urlparms (age . "42") (phone . "12345678") (email . "nick%40freemail.com") //
(house-number . "42") (street-name . "highroad") (postal-number. "1234") //
(city-name . "thy") (country . "denmark") (last-name . "hansen") //
(first-name . "nick"))
```

All the basic information from the `input` elements are present in the data from the HTML form received, but there are no information telling the developer anything about the structure. The following section, will present possible ways to handle the recreation of the complex structure.

Rebuilding the Structure

When form parameters are submitted by the client, all information about the composition of the elements are lost. There are e.g. no information telling, that the *first-name* and the *last-name* are actually parts of the composite element *name* in the person structure. Information about the composition could be handled by hidden `input` elements in the HTML form. To solve this, a hidden `input` element named *name*, which has the value *first-name+last-name* can be created. The information from the hidden `input` elements are included in the information from a submitted HTML form, so the hidden `input` element *name* will result in the string *name=first-name+last-name* when received from the client. This tells the developer the value of *name*, but it will be on the same level as all the other information from the input fields. This means, that the only way to distinguish between the information representing client input and information representing structure, is the key (*name* in this example) in the received data from the HTML form.

To recreate the structure on behalf of a parameter string, where information about the structure is mixed with data from the client is difficult. First, the developer must know the names of all the keys representing the structure, in order to rebuild it. Second, the developer must be sure, that names related to a "data" `input` element are not in conflict with names related to a hidden `input` "structure" element. Third, functions that create complex structures on behalf of strings must be created.

Instead of giving each part of the structure its own hidden `input` element, the entire structure can be stored in a single hidden `input` element. This requires the developer to first create a "template" of the structure, and then place it in a hidden `input` element when the HTML `form` element is created. The *person* structure can e.g. be placed in a hidden `input` element named *form-structure*. When the data from the HTML form are submitted, the developer only needs to find the value of the *form-structure* key in order to have a representation of the structure. When having the structure at hand, it can be updated with the data from the client. Using this way of handling HTML forms, the developer is required to perform three steps. These steps are illustrated in Figure 1.10.

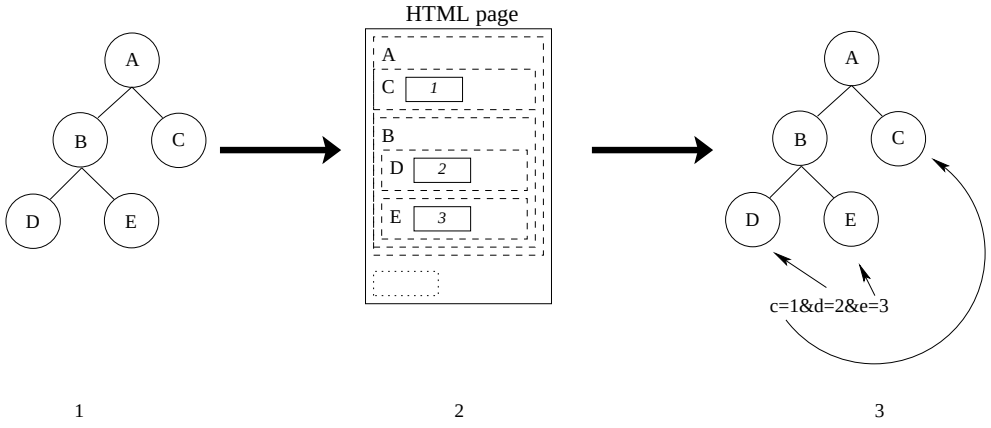


Figure 1.10: In step *(1)* the developer creates the structure wanted. Step *(2)* presents a similar structure as an HTML form. The structure created in step *(1)* is stored in a hidden `input` element (the dashed box in the bottom of *(2)*). In step *(3)* the structure is extracted from the HTML form, and updated with the values entered by the client.

The structure template from step *(1)* - in Figure 1.10 - can be the person structure already presented. By doing the layout in HTML as done in Figure 1.9 and storing the person structure in a hidden `input` element, step *(2)* can be achieved. Updating the structure (step *(3)*) is done by first extracting the structure, and then in turn handle each key in the data from the HTML form. If a key is present in the structure, update the structure with the keys value. Otherwise continue to the next key. The above two alternatives (hidden information about the composition or the entire structure) are self contained, since information about the structure are available in the data received from the client. Instead of storing the entire structure in a hidden `input` element, a reference to the template can be stored (like *structure=person*). This can be handled in the same way, as data are stored on the server. This results in the structure not being self contained, as it cannot be reconstructed from the data received from the client. Information about the structure is needed from the server.

When creating an HTML form based on a template or directly in the language, it is difficult to see the relation between the representation of the structure and the structure itself. A relation is made, if functions that can create an HTML form on behalf of a structure exists.

It is then a specific structure that is presented as an HTML form, and if the structure is modified, it is mirrored in the HTML form. A similar relation is achieved if the developer has the possibility to specify the HTML layout of a structure, e.g. by specifying the layout of each of the elements.

The *Complex forms* problem was presented in this section. First, it was seen, how information about a structure can be stored in hidden `input` elements. This allows the developer to recreate the structure presented as an HTML form, and fill it with the data entered by the client. Instead of recreating a structure on behalf of information in hidden `input` elements, it was presented how the developer can work with a "template" of a structure. The latter approach consisted of three steps, namely *creating*, *representing* and *updating* a structure.

1.1.4 Reusability

During the advances in programming technologies the concepts of modularity and reusability has become natural to developers. A developer will have an instinct that ensures reusability of some of the code by applying modularity to it. During the programming task the developer need functionality that performs a specific task, be it extracting data from a data structure, or perform computations, based on a certain algorithm. Rather than creating the functionality on the spot a function is written, that - based on parameters - performs the computation and returns a result. By the use of this function reusability emerges. The function can be used again in different contexts, where that particular functionality is needed.

This is the reusability that developers has become accustomed to, namely reusability in terms of general purpose functionality. The same amount of reusability has not been introduced in Web development, when not considering general purpose functionality. It is of interest, that ideas and concepts as modularity and reusability from non-Web development can be used in Web development. Likewise, is the principle of information hiding - often employed in relation to the module concept - of interest.

Often a Web developer creates a function that performs a task and thus use this function as an abstraction over more detailed actions. An example is e.g. writing a function that outputs a header of an HTML page. This kind of reusability is on parts of pages. Reusability in terms of pages does also exist, in technologies such as PHP and ASP - see Section 1.2 - due to the template like nature of the pages developed. It is interesting to consider if it is possible to extend this kind of reusability to entire sequences of interactions with a client (inspired by sessions in Bigwig). Imagine a situation where a number of pages have been developed for one Web application, but on the next project the developer needs some pages that represents the same task. Most likely the developer will copy the prior made pages and edit them to fulfill the needs in the current project.

An example of a series of pages that is subject to modularization is a login sequence. A number of pages responsible for getting credentials from the client, or if the client does not have any, then offer the opportunity to receive credentials, by performing a registration of the client. This is functionality that is applicable in many Web applications. Imagine the

developer having defined a module and having defined a flexible interface - in terms of parameters - to the module. Then the developer can use that module, specifying the values for the various parameters.

In this section the *Reusability* problem has been discussed. This problem is related the interest of introducing some of the programming concepts only present in non-Web development. The possibility to define a series of pages as a module is sought, since it allows the developer to reuse a module in different contexts. The module can be created to take parameters, which e.g. specify information about the layout of the pages it includes. Creating modules also makes for information hiding possible.

1.1.5 Summary

Section 1.1 introduced four problems in Web development. These are the *State handling*, *Input validation*, *Complex forms* and *Reusability* problems. The *State handling* problem is split into two sub-problems, namely *Data flow handling* and *Control flow handling*. The *Data flow handling* problem, concerns the need for the developer to handle data received from the client explicitly, in order for data to survive multiple interactions. This can be done, by storing the data in hidden `input` elements, or storing data on the servers filesystem. The *Control flow handling* problem is the problem, that concerns the need to maintain the interactions between server and client. Possible solutions to maintain information about the interactions were presented. The first is to use primitives from the programming language, like the `cond` special form in Scheme. Another possibility is to use the `action` attribute on the HTML `form` element. Both sub-problems in the *State handling* problem, are rooted in the stateless nature of the HTTP protocol.

The *Input validation* problem is related the need to validate data from the client. Validation can be handled on the client or on the server. If validation is handled on the server, a client/server loop must be maintained. However, server side validation ensures that data submitted by the client is validated. This is not necessarily the case with client side validation, since the client can disable the technology used for validation. Such a technology is e.g. JavaScript. Validation on the client yields faster validation, since it is not needed to maintain a client/server loop, which uses bandwidth.

A Web developer can present a complex structure in an HTML form, by using HTML elements. Information about this structure is lost, when the client submits the data entered in the HTML form. This is seen as a problem, which is named the *Complex forms* problem. Possible solutions to how the structure presented as an HTML form can be rebuild after the HTML form is submitted are presented. The HTML form structure can be specified in hidden `input` elements (by specifying information about the composition of the individual elements) or by using a "template" approach. Three steps must be performed if the template approach is used. These are *creating*, *presenting* and *updating* a complex structure.

The final problem presented is the *Reusability* problem. It is of interest, that the developer can define series of pages as a module. By letting the module take parameters, the pages

it represents can be customized in the way defined by the developer. Such a customization can e.g. be a style sheet.

1.2 Approaches to Web Development

During the preparatory work we found related work that might hold a solution to some of the problems just analyzed. We adapt the line of thought presented by Bigwig, that Web development can be divided into three different approaches or paradigms; namely the script-centered, page-centered and the session-centered approach. The script-centered approach is by Bigwig characterized as follows:

> *"The script-centered approach builds directly on top of the plain, stateless HTTP/CGI protocol. A Web service is defined by a collection of loosely related scripts. A script is executed upon request from a client, receiving form data as input and producing HTML as output before terminating. Individual requests are tied together by explicitly inserting appropriate links to other scripts in the reply pages."* [CAM02]

In the script-centered approach the individual scripts are in focus. Normally traditional languages are used for writing the scripts, i.e. not languages written or created especially for this purpose. Examples of these languages include Perl and C. It is the program code that is the essential part here, HTML is written as the output from the script. Therefore the HTML pages are generated in a top-down manner using print-like statements, requiring the developer to be more structured in his development style. One of the disadvantages by this is the lack of flexibility in the generation of the pages. For example once the HTML `title` element has been written it is to late to write the `head` element.

According to Bigwig the page-centered approach considers Web development in quite a different manner:

> *"The page-centered approach is covered by language such as ASP, PHP, and JSP, where the dynamic code is embedded in the HTML pages. In a sense, this is the inverse of the script-centered languages where HTML fragments are embedded in the program code. When a client requests a page, a specialized Web server interprets the embedded code, which typically produces additional HTML snippets while accessing a shared database. In the case of JSP, implementations work by compiling each JSP page into a servlet using a simple transformation."* [CAM02]

In the page-centered approach, the Web developer designs the page layout with graphics etc. The parts of the page where dynamics are needed, the developer escapes the design or HTML and writes the program that generates the wanted dynamics. This make the Web development process more focused toward the design of the final look of the page, unlike the script-centered approach which is more like non-Web programming. It helps to increase the overview of the Web development for the developer if only small amounts of dynamics are needed. But if a page is filled with program fragments it clutters the developers overview in

the same way as the script-centered approach does. Therefore there is a trade off between simplicity and dynamics in this approach. It is often simple pages that are written in this style. The developer still has to explicit link various pages together to create the illusion of coherence between a number of pages. The page-centered approach to Web development also introduces sessions. This is done, by maintaining a global state, which contains information about data received from clients. This is e.g. done by PHP [The02].

According to Bigwig there is in the session-centered approach a coherence of the individual pages shown to the client. The developer writes a session as one program, that encapsulates the presentation of the individual pages. This program is executed and represents the session.

> *"A service is here viewed as a collection of distinct sessions that access some shared data. A client may initiate a session thread, which is conceptually a process running on the server. Interaction with the client is viewed as remote procedure calls from the server, as known from classical construction of distributed systems but with the roles reversed."* [CAM02]

By writing the entire interaction between the server and the client as one program the developer obtains a better overview of the development process. Writing several pages as part of a session (a program) it is possible to share data between the individual pages without having to explicitly transfer the data between the individual pages.

This section has presented three different approaches to Web development, namely the *script-centered*, the *page-centered* and the *session-centered* approach. It is chosen, that focus is placed on the *session-centered* approach to Web development, when solutions to the four problems - *State handling*, *Input validation*, *Complex forms* and *Reusability* - are designed. The reason for this is, that a session introduces encapsulation of pages, which is a solution to the *Reusability* problem. Each session is then equal to a module of pages. If a module can be customized with parameters (e.g. a style sheet) when activated, it can be used in various contexts. Furthermore, does a session represent the flow of pages in the Web application and allows for client data to be shared between pages. Handling data and flow in a Web application, are the two subproblems in the *State handling* problem. This means, that a session concept in Web development solves both the *Reusability* and the *State handling* problem. To identify possible approaches to session-centered Web development, the next section presents work related to sessions.

1.3 Related Work

Based on the problems presented, related work in the area of Web development has been analyzed, to find existing solutions to problems similar to those described in Section 1.1. There exists few examples of technologies that relies on the session centered approach to Web development. Bigwig, and its ancestor Mawl[DJ01] were the first encountered. Mawl will not be discussed, since Bigwig covers the same aspects as Mawl in relation to our problems. WASH/CGI also adhere to the session centered approach to Web development

and it does so with basis in the functional language Haskell. Furthermore, a library[3] called PAKCS/HTML for the language Curry[Mic02a], is analyzed. PAKCS/HTML is shipped with the PAKCS Curry implementation[Han02] and implements sessions as an optimization of plain CGI. These technologies are analyzed in the following sections, to uncover their relation to the session centered approach to Web development and to find ideas usable in the context of our problems.

1.3.1 Bigwig

The session concept - and the session-centered approach to Web development, which is the motivating factor behind the session concept - was originally presented by the Mawl language. In essence both Mawl and Bigwig handles sessions the same way. They operate with the session-centered approach to Web development as an alternative to the page and script-centered approach. An important factor behind the session-centered approach is that the developer thinks in sessions (whole series of interactions with the client) rather than individual pages that makes up a whole application. Furthermore these interactions are written as one large program since this gives a better overview of the development process and therefore helps to produce more structured and coherent Web applications. The reason for this is, that the developer has the overview to spend more time on the flow of the application, rather than linking the individual pages together.

Bigwig is a framework that rely on compilation and static checks, rather than what is normally used in Web programming, namely interpretation. Compiling the Web service enables type checking and static analysis which ensures - to some degree - the correctness of the service. Bigwig rely on static type checking, because it catches many of the errors that otherwise occur at run-time. Bigwig is a C and Java-like skeleton language that binds together a number of domain specific languages. Services written in Bigwig can by the compiler be translated into standard Web technologies such as HTML, CGI, JavaScript, Java applets and elements of HTTP. Bigwig see the use of only standard technologies as an advantage as these do not require special language support from the client.

A session in Bigwig is part of a service. The Web developer writes a service and creates a number of sessions as part of this service. A service corresponds essentially to a sequential program. The Web developer therefore writes a service as any other program and includes sessions as part of this program. For an example of this practice see Figure 1.11. A service is created which contains definitions of HTML pages, here the *Please* and *Greeting* HTML pages. It also contains sessions, here the *Hello* session which shows the *Please* HTML page to the client and receives the name entered by the client in the string variable s. Next the *Greeting* HTML page is shown with the just received data (placed in s) as part of the page. Notice the *show* function used to display the page. This function takes the page and shows it to the client. The program is continued like *show* is a normal procedure call.

[3]We will refer to this as PAKCS/HTML. The actual name is not known but when downloading the PACKS Curry system the library is included as HTML.

```
service {
   html Please = <html>  Please state Your name:
   <input type=text name=handle> </html>;
 html Greeting = <html>Hello <[moniker]>, how are you?</html>;
 session Hello() {
  string s;
  show Please recieve[s=handle];
  show Greating<[moniker=s]>;
 }
}
```

Figure 1.11: An example of how a service and a session are related and how a program is written in Bigwig [CAM02].

Using sessions in Web development makes the communication between the client - and the server running the session - roughly similar to remote procedure calls (see [AB84]), or as Bigwig state it:

> *"Communication is performed by showing the client an HTML page, which implicitly is made into a form with an appropriate URL return address. While the client browses the given document, the session thread is suspended on the server. Eventually the client submits the form, which causes the session thread to be resumed and any form data entered by the client to be received into program variables."* [CAM02]

To get a better understanding of this way of handling sessions, Figure 1.12 illustrates the client and the session thread during the flow of a session. The session begins with the client requesting the URL matching the session. The session thread computes the session until the first page is shown to the client. Then the session thread is suspended and goes idle on the server. The client receives the page and submits data. When the client is done the result of the page is send back to the server. Once the server receives the result it reinvokes the session thread and the computation is continued.

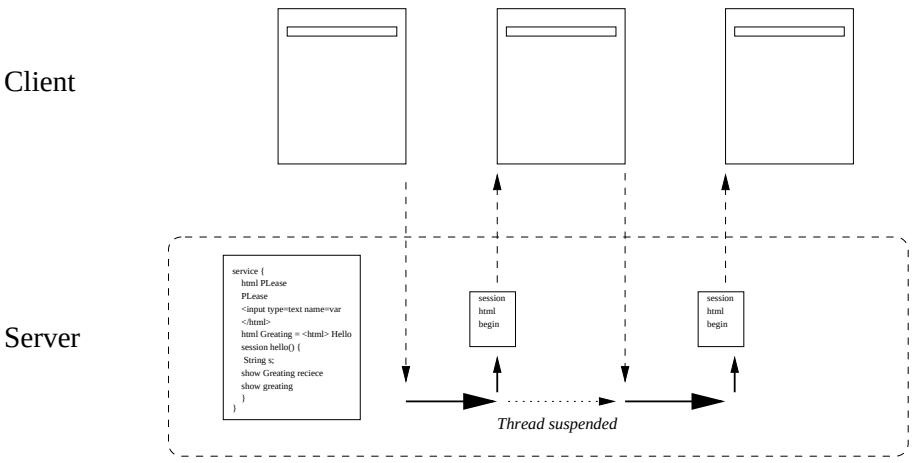


Figure 1.12: An illustration of the flow of a Bigwig session[CAM02].

The service concept as introduced by Bigwig makes it possible to share data between the individual sessions that belongs to a given service. This is achieved by creating variables etc. with the *shared* modifier. The ability to share data between sessions allows sessions to communicate with each other.

Bigwig see a problem in the limitation that all communication between the client and the server must be handled by presenting and submitting pages with HTML forms. Their example is a chat room. In a chat room the client receives new messages without having to reload the page. Therefore Bigwig has created the concept of seslets. Basically seslets is a limited session running on the client with permission to contact the server. This seslet can then be used to contact the server and retrieve new messages in the chat forum at a regular interval.

Bigwig also introduce a concept, that is used when validating input. They have named this Powerforms[BMRS01], which is a declarative way of handling validation. The developer specifies valid input to an `input` element and the validity of the `input` element is ensured by a JavaScript that is created automatically.

A number of interesting ideas has been used in Bigwig, some of which will be used to solve the problems described in Section 1.1. The most noticeable is the idea of sessions. Bigwig relies on the session concept for two things. First it is an entirely different way to develop Web applications, a way that more resembles non-Web development than CGI development. The second benefit with sessions is the ability to introduce persistence on the server, thus eliminating the need for sending data between the client and the server, for the data to be available at a later time during the session. To introduce functionality - in the form of the *show* command - that represents sending a page to the client and receive data submitted, seems like a good idea. This can help to conceptualize the interactive nature of Web application for the developer.

Bigwig have problems regarding stepping back and bookmarking a page in a session. The problem is that if any of these two event happens, the session is started from the beginning. The reason for this behavior is that Bigwig sees it as dangerous to step back in a session, since some actions might change the state and this state change is hard - if not impossible - to undo. Example of such actions are file writing and database updates. However, our opinion is that stepping back in a session can by dangerous, but reasonable to support. The reason is that the back button supports an explorative nature, when the client is browsing the application. If a back button instead is a link on the Web page, it is sometimes cumbersome to find.

1.3.2 WASH/CGI

Another session centered approach is WASH/CGI. WASH/CGI is a library - for Haskell - providing help when developing Web applications. Programs written in WASH/CGI are compiled. WASH/CGI considers the session concept as a structure that improves the overview of the Web development process, and it considers a session in the following way:

> *"A session is a dynamically evolving sequence of* `ask` *and* `io` *actions (in the CGI monad). Each of these actions queries the external world, either by displaying a form on a Web browser or by performing and IO action, and receives a response."*
> [Pet]

Just as with Mawl and Bigwig the Web developer thinks of sessions, when developing Web applications. Like Bigwig, WASH/CGI uses a primitive for displaying a page to the client. WASH/CGI relies on the *ask* function call - just as Bigwig relies on the *show* function - to ask and receive data from the client.

WASH/CGI does not suspend the process on the server - like Bigwig does it - when a page is send to the client. Instead execution of WASH/CGI applications are ended after a page has been shown to the client, as done in traditional CGI programming. Therefore some way of receiving the data and resume execution of the session with the appropriate data has to be used. To remember the data already asked from the client a list, called *inparm*, is used. Once a session is invoked and executed, it checks if the data asked for, are already present in the *inparm* list, and if so, the client is not asked for it again. Instead the data in the list is used. The data in the *inparm* list are stored with an association to the individual *ask*. This means, that the data received from each interaction with the client, are added the *inparm* list. Thus the *inparm* list contains all data already received from the client, and therefore acts as a session status. Persistence of this list is obtained by passing the list data with each page shown to the client in the form of a hidden HTML `input` element. This results in the entire session being computed up to the point of the page that is requested, at each request. It seems to be a waste of time to start the application from the beginning every time. But since the data needed from the client is retrieved from the *inparm* list, the time used to compute the page to be shown next time is minimal. Not only data from the client are stored in the *inparm* list, but also IO actions are stored in the list, since these must be undone if the client steps back in the computation.

Figure 1.13 illustrates the steps taken in execution of a WASH/CGI application. The client requests a WASH/CGI application which is executed. The application contains three interactions (one page for each interaction) with the client. First the *inparm* list is checked to determine if *Page1* is already present. If not, *Page1* is shown to the client. Since the application has just been started, *inparm* is empty, and *Page1* is shown to the client. The client enters data on the page and submits it to the application on the server. The server starts the application again and associates the data received from the client with an entry in the *inparm* list for *Page1*. The *inparm* list is checked to determine if it contains the data for *Page1*. This time it does and the next step is taken. Again the *inparm* list is checked to see if it contains an entry associated with *Page2*. It continues like this until the end of the application or the client stops submitting pages.

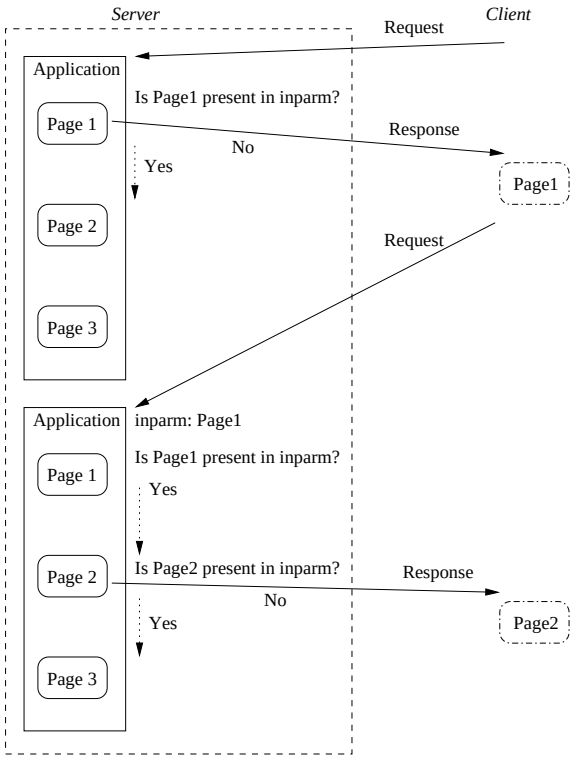


Figure 1.13: An illustration of the usage of the *inparm* list in WASH/CGI.

Having the *inparm* list present in every page sent to the client and relying on it when reestablishing the state of the session, it is possible to step back in a session. This works since the client pressing the back button results in re-sending an old form and the server reestablishing the state from the *inparm* list present in this page.

WASH/CGI includes interesting work involving HTML forms. WASH/CGI relies on an abstraction in the form of functions to generate HTML. It also has functions to generate HTML `form` and `input` elements. The function that generates the HTML `form` automatically sets the appropriate `enctype`, `method` and `action` attributes on the `form` element. The `enctype` attribute is used to specify the encoding of the contents of the HTML form when it is send to the server. The `method` attribute is used to identify whether the `GET` or `POST` method is used when submitting the HTML form. The `action` attributes specifies the functionality that receive the data from the HTML form. By setting these attributes automatically the Web developer is alleviated from this responsibility, and it is ensured that the attributes are always correct. The function used for generating HTML `input` elements - e.g. textual inputs, check boxes etc. - returns a handle to the `input` element. This handle contains - once the page containing the `input` element has been shown to the client - the value entered into the `input` element. The developer can then access this data using either the *value* or the *string* function returning the parsed value or the unparsed value respectively. To associate these handles with the data entered by the client, the submit `input` element - which is generated from a function as any other HTML element - is used. It is defined by passing it

functionality which is activated once the submit button is pressed. This action associates the data entered by the client with the handlers received when defining the `input` elements. The function generating the `input` elements automatically provides naming. It also has a means for presenting default values based on a log of previously received input.

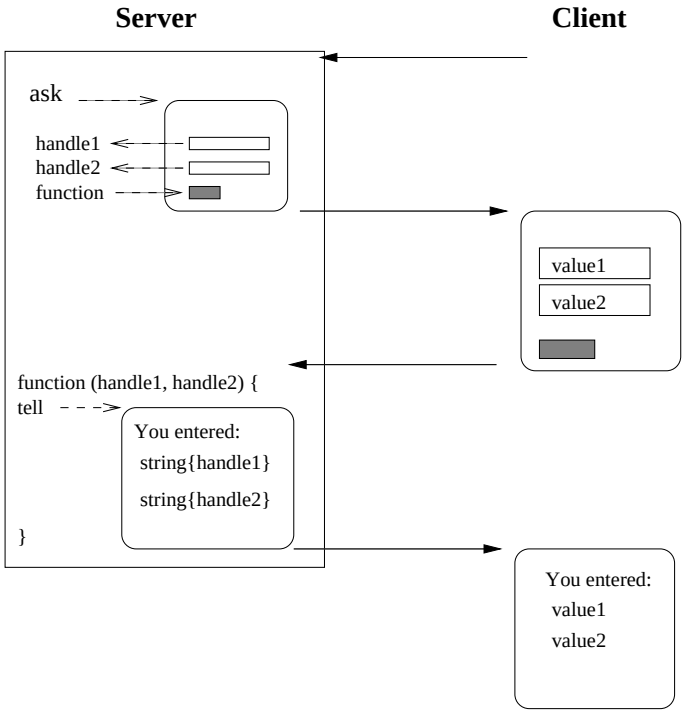


Figure 1.14: An illustration of form handling using call back functions and handlers in WASH/CGI.

Figure 1.14 illustrates how HTML forms are handled in WASH/CGI. First the developer specifies an HTML page and states - for each `input` element - that the values entered into this `input` element is associated with a given handler, here *handle1* and *handle2*. Next the developer associates a call back function - here *function* - with the `action` of the submit `input` element. Then the page is shown to the client with the *ask* function. The client enters data into the HTML form and presses the button. This results in the data being submitted to the server. When the server receives the data from the HTML form it activates the call back function *function* and the handles are passed as parameter to this function. The call back function extract the values from the handlers with the *string* function and presents them - with *tell* - to the client in a new HTML page.

WASH/CGI does not provide a solution to the problem with complex structures as it is not possible to receive the data from an HTML form in a defined structure. It is possible to represent a structure at the client, since the developer can construct complex structures based on HTML elements, but the structure does not survive an interaction with the client. Relying on a list to contain the data already received from the client, allows for a simulation of the session resuming its computation from the point it stopped, when sending a page to the client.

1.3.3 PAKCS/HTML

By relying on the mixed paradigm language Curry, Michael Hanus describes in [Han01] how solutions to the problems with the plain CGI approach to Web development can be solved. This is done by implementing a library for the Curry language. Curry is described as a mixed paradigm language and its constituents include elements from the functional, the logical and the concurrent programming paradigm. When used for Web development the developer does not write the HTML code as text strings in print-like statements in the language. Rather Web programming with PAKCS/HTML is done by relying on an abstraction layer above plain CGI, where HTML documents are constructed using a specific HTML data type representing the HTML (also referred to as an HTML expression). A wrapper function is responsible for translating the HTML data type to a textual representation, when the page is shown to the client. The introduction of this abstraction above plain CGI introduces a number of benefits which are described in the following.

The wrapper function is responsible for more than constructing the textual representation of the HTML data type. The wrapper function is also responsible for retrieving the data entered into HTML forms by the client. This is done by introducing elements of an abstract data type that the developer can use when constructing the HTML page. The idea is that the developer can specify an element of the abstract data type, and use a logical variable that is part of the data structure as reference to an `input` element in the HTML page. Introducing a logical variable as reference to the `input` element is done, since the variable is not instantiated until after the HTML page has been shown to the client. A logical variable is a way to express the delayed instantiation of a variable. When the HTML expression is processed by the wrapper, the textual representation is generated. At the same time, the wrapper instantiates variables, which are used as references to the `input` elements on the HTML page. When the client submits the HTML form, the data from the `input` elements are associated with the variables instantiated by the wrapper. Data from the HTML form can then be accessed by using the variables.

Another element of the abstraction is that the program that generates the HTML form - which is shown to the client - is also the program that is activated when the client submits the form and the wrapper has done its work. This allows a sequence of interactions to take place based on the control abstractions of the Curry language. The idea is to associate an event handler with each submit `input` element that is shown to the client. Once the wrapper has received the data from the HTML form, it activates this event handler passing a CGI environment as parameter. The CGI environment is a mapping from the names of the `input` elements present in the HTML form to the strings entered by the client. By requiring an event handler to return a new HTML page, containing a new HTML form, the concept of sessions has raised. The result of executing the event handler is to show a new page to the client. This allows nesting of event handler and thereby series of interactions can be obtained. This resembles the session-centered approach to Web development, since the developer is able to specify the entire interaction between the client and the server, as one large program.

It is also possible to obtain the session-centered approach to Web development without relying on nesting of event handlers. Since the various control structures of the entire Curry language is available the developer can rely on these. For example, the developer might rely on recursion to repeatedly show a page until the client has entered the correct data. Or a select statement can be used to determine - on basis of just received data - which page to show next.

For an illustration of how the interactions between the client and server is handled in PAKCS/HTML see Figure 1.15. The interaction begins with the evaluation of a function - here `function` - on request from the client. The entire box surrounding everything in the server is considered the session that the client activates. The server then executes the `function` function, which generates an HTML page containing two text `input` elements and a submit `input` element (here the hatched box on the figure). As seen the two text `input` elements are associated with the logical variables (here *logical_ var* and *logical_ var1*). The generated HTML page is shown to the client, filled with data and submitted again. After this, the event handler associated with the submit button (here *eventHandler*) is activated and an environment (here *env*) is passed as parameter. The event handler then generates an HTML page containing the values entered by the client. These values are obtained by applying the environment on the logical variables. The resulting HTML page is shown to the client and the session is terminated.

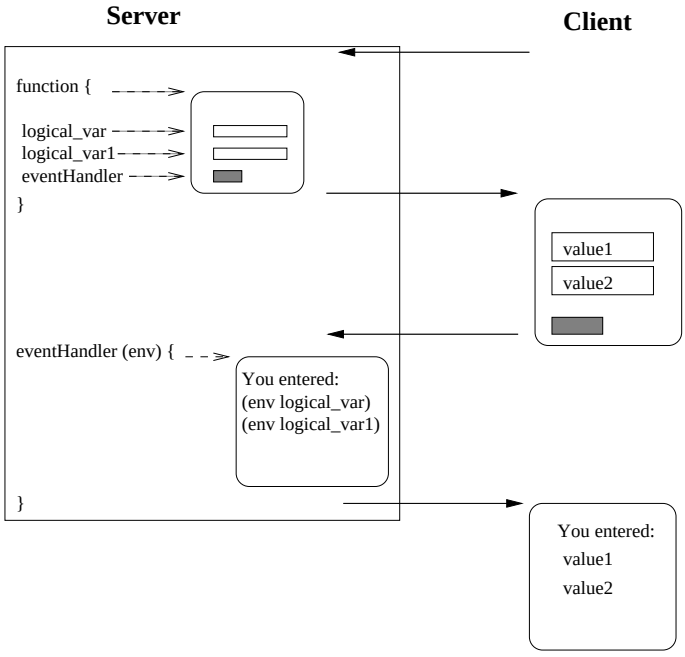


Figure 1.15: An illustration of handling an HTML form using event handlers and logical variables in PAKCS/HTML.

A problem with traditional CGI programming is the lack of state on the server due to the stateless nature of the HTTP protocol. Normally this is solved by placing the state on the

client either in cookies[DL02] or hidden `input` elements. In PAKCS/HTML this is solved in a simpler way. Since the entire interaction consists of nesting of event handlers, there cannot be two pages/event handler on the outer level. There must be one function which is responsible for the first interaction with the client, and the functionality representing following interactions must be nested inside it.

A number of interesting ideas has been used in this work. The idea of having a wrapper function decoding the parameters from the client and making them available is beneficial. However, it is not possible to have structure on the data, since the value of each `input` element is mapped to the value of a variable in the programming language. This means, that no information about the composition of `input` elements are maintained. Furthermore, it is impossible to uniquely identify the value of variable to be equal to the value entered in a specific `input` element. The reason for this is, that the same name can be used to identify `input` elements in different HTML forms.

The idea of allowing control structures of the Curry language to be used when developing a series of interactions seems natural, as it resembles non-Web related programming. However, sessions in PAKCS/HTML is different from sessions in Bigwig, as there cannot be more than one top-level function. Initially an execution of an application is done on request from a client by locating the appropriate script (the one that generates the first page in the application) and execute it. But the following executions in the application is done by calling the associated event handler (the one bound to the submit button that the client presses). Therefore, placing two functions in the same script, does not result in both being executed. Placing several functions on the top-level might be beneficial, since it introduces an overview of the flow of the application. The developer can specify a number of functions and rely on these being evaluated one by one, until the session terminates.

1.3.4 Summary

Section 1.3 introduced work related to the session-centered approach to Web development. This was done, since a session concept solves two of the four problems identified, namely the *State handling* and the *Reusability* problem. The related work analyzed were Bigwig, WASH/CGI and PACKS/HTML. Bigwig introduce sessions by a primitive in the language. This primitive results in a lexical scope forming an encapsulation of a sequence of pages. WASH/CGI and PACKS/HTML uses a nested handler approach to specify the sequence of pages represented by a session.

A third approach was found in the page-centered approach to Web development, namely sessions by global state. This approach was identified in Section 1.2.

1.4 Problem Definition

Following the analysis four problems in Web development exists. These are: the *State handling* problem, the *Complex forms* problem, the *Input validation* problem and the *Reusability* problem. These problems are formulated as three hypotheses.

The first problem is *State handling*, which - during analysis of related work - have been seen dealt with by Bigwig, WASH/CGI and PAKCS/HTML. We expect to solve the *State handling* problem and *Reusability* problem, by introducing a session concept inspired by Bigwig. The reason is, that it is possible to let interactions with a client, happen in the same lexical scope. This makes it possible to let interactions with a client share data. This is the solution to the *State handling* problem. The *Reusability* problem is solved since this lexical scope can be activated and thereby do interactions with a client. Because of the lexical scope, it is possible to regard more than one interaction with a client as a unit. This is formulated in the following hypothesis:

Hypothesis 1:

> ***A session-centered approach to Web development in SLAML solves the State handling problem of a Web application. Furthermore, a session concept makes access to several HTML pages as a single unit possible.***

This hypothesis is general and three problems are included in it. These three are the *Control flow handling*, the *Data flow handling* problem and the *Reusability* problem. To be more precise, the hypothesis is spelled out in three sub-hypotheses, each involving one of the subjects.

With respect to control flow:

Hypothesis 1.1:

> ***The Control flow handling problem is solved by introducing a session concept, where a primitive in the language displays an HTML page to a client and returns as a regular function.***

With respect to data flow:

Hypothesis 1.2:

> ***The Data flow handling problem is solved by introducing a session concept to SLAML, where interactions inside the same lexical scope (session) can share data.***

With respect to reusability:

Hypothesis 1.3:

> ***The Reusability problem is solved by introducing a session primitive that can activate a series of interactions with a client and rely on parameters at call time.***

The *Complex forms* problem is the second problem discussed. A developer will benefit from having a framework that when constructing complex data structures on the server can present them to the client as an HTML form and update them with data from the client. This lead to the second hypothesis.

Hypothesis 2:

> ***It is possible to construct a framework that helps the developer to build, present and update complex structures.***

The last problem is the *Input validation* problem. By construction a validation framework, validation of data from the client is done simple. This is formulated in the third hypothesis.

Hypothesis 3:

> ***It is possible to construct a validation framework that helps the developer to validate data from the client.***

Relying on mod_laml, a session-centered approach to Web development in SLAML will be developed. This framework is called the *SLAML framework*. As a part of the SLAML framework is the session framework and a solution to the *Complex forms* problem. Furthermore, a validation framework that fit within the SLAML framework will be developed.

2

Design

Contents

In this chapter three main sections discusses and presents the decisions made in the design phase of this project. The first section explains how the session concept is designed and what alternatives are possible. The second section explains the design of the extensions to the session framework to make construction, presentation and updating of complex structures possible. Last is the design of validation of data in the SLAML session framework.

Throughout this chapter new primitives are introduced and explained. For a complete description of the primitives a reference to Appendix A is given. To get a complete understanding of each primitive the reader is requested to consult this appendix. Furthermore we use a number of concepts, throughout the rest of this report. In the following box it is described what we mean by these concept.

Attribute: By attribute we mean a key/value pair consisting of a name and a string. An example is *type = "TEXT"* from an HTML `input` element. The whole is addressed as the attribute. *type* is addressed as the attribute name, and *TEXT* as the attribute value.

Elements: An element refers to an element as it is known from the SGML family of languages. An element consists of content and attributes. An example is; *<element attributes-name attribute-value ..> Contents </element>*.

Content: Content refers to everything inside a double tagged element. The contents of one element can be other elements.

Tag: By tag we refer to a symbol from Scheme present in a list. I.e. a tagged list is a list containing a symbol as the first element.

Primitive: By a primitive we refer to the name binding of a function or a special form.

Form parameters: Is the term used for the data entered by the client into an HTML form and submitted to the server.

The overall goal of this chapter is to present and discuss the constructed framework that solves the problems presented in the problem definition in Section 1.4. Furthermore, the framework is designed to work in a server context where mod_laml is used as an implementation platform. Therefore, it is not necessary that the framework fits with CGI. Part of the context of this project is Scheme and the LAML libraries. Therefore the solutions will adhere to the XML like syntax used in LAML. But as the framework makes use of sending functions as parameters to other functions, XML syntax will not always be possible. Where the syntax of XML is not followed directly a notice will explain why it is chosen to deviate from the LAML syntax.

2.1 Session Framework

In the problem definition (see Section 1.4) a hypothesis is presented regarding the use of sessions to solve the problems of *State handling* and *Reusability*. Three sub-hypotheses are presented to expand the first hypothesis. The sub-hypothesis state that introducing a session concept can solve the *Data flow handling* problem as well as the *Control flow handling* problem. Furthermore, allowing sessions to rely on parameters, sessions can solve the *Reusability* problem. The goal is to design a session concept in SLAML that solves the problems from the three sub-hypotheses and thereby the first hypothesis.

In this section the design of the session concept in SLAML is explained. First is considerations regarding the design of sessions in SLAML. Second, choices made regarding the design

is explained. Third, the flow of a session is explained, followed by an example.

2.1.1 Design Considerations

Three ways of constructing the session concept is found in related work (see Section 1.3).

***Sessions by Global State*:** This is the approach used by various page centered approaches to Web development. This includes PHP, ASP and JSP. Roughly a session is defined by a global state associated with a client.

***Sessions by Nesting Event Handlers*:** This approach to sessions is to nest event handlers thereby achieving a session concept. Event handlers are nested by letting one event handler present a page containing a reference to another event handler.

***Sessions as Lexical Scopes*:** This approach relies on lexical scope rules of the language to represent a session. All interactions taking place in the same lexical scope (session) share data.

These three approaches for constructing sessions are explained in details in the following sections. Each of the three approaches are discussed in relation to data flow and control flow.

Sessions by Global State

This is the concept of sessions used when most page centered approaches implements sessions. In this approach sessions are achieved by maintaining a global state on the server, so interactions with a client can access shared data. In this way the scripts can share data across invocations. The global state can be located on either the server or the client. The maintenance of the global state is often handled in the language. The consequence of using this strategy is that all pages share the same data. This means that there is no way of securing the data from other pages that must access the data (unless it is done explicitly by the programmer).

This approach to sessions is illustrated in Figure 2.1. Here it is seen that four requests from a client all accesses the same global state. If a new page is requested by the client, this request can access the global state as the others. In this approach it is not possible to protect the global state from being accessed by e.g. *Page 1* and *Page 2*. There exists no encapsulation to indicate that the global state must only be accessible from *Page 3* and *Page 4*. This is the first problem with this solution. There exists solutions where the client - based on an unique id (session id) - can get access or is denied access to global state. This is usually done by associating the global state with the session id. The second problem is that the flow of the application is spread across several scripts. The flow of the entire application is not placed centrally, allowing the developer to quickly overview the session. Rather the flow of the application is represented by the activation of various smaller parts one by one.

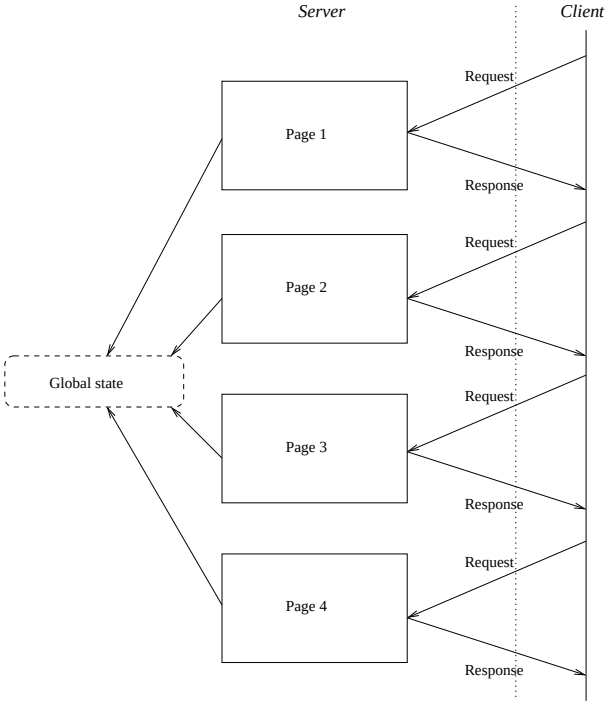


Figure 2.1: An example of how global state is used to share data between interactions with a client.

Introducing global state on the server results in increased requirements for the server in terms of memory. The amount of memory required depends on the amount of clients using the server. It does not increase drastically since session state is moved to disk after an amount of time. Regarding disk space there is increased requirements so there must be an expire time for session state. It is hard to determine the amount of time that a session must be on disk before it is expired. It is advisable to run the server with the sessions in a period (for example six months) and during this time gather statistics about the usage of the sessions on disk. This provides sufficient information to make a qualified decision.

Sessions by Nesting Event Handlers

The event handling approach to sessions is the approach used in WASH/CGI (see Section 1.3.2) and PACKS/HTML (see Section 1.3.3). The idea is to let the submit button in an HTML form on an HTML page be associated with an event handler. When the HTML form is submitted, the event handler is called. In this way it is possible to create an interaction sequence by letting the called event handler generate a new HTML form and associate this HTML form with another event handler. This makes it possible to share data among pages as these can be sent as parameters to the called event handlers. As the WASH/CGI and PACKS/HTML solutions rely on the CGI protocol, the script being executed as part of the application program has to end after having processed the request. The parameters have to be send with the response to the client, for them to be available at the next page. Furthermore, the flow of the application is spread across several event handlers. It is therefore

difficult to get an overview of a whole session.

This approach is illustrated in Figure 2.2. Here it is seen how the submission of one HTML form activates another event handler.

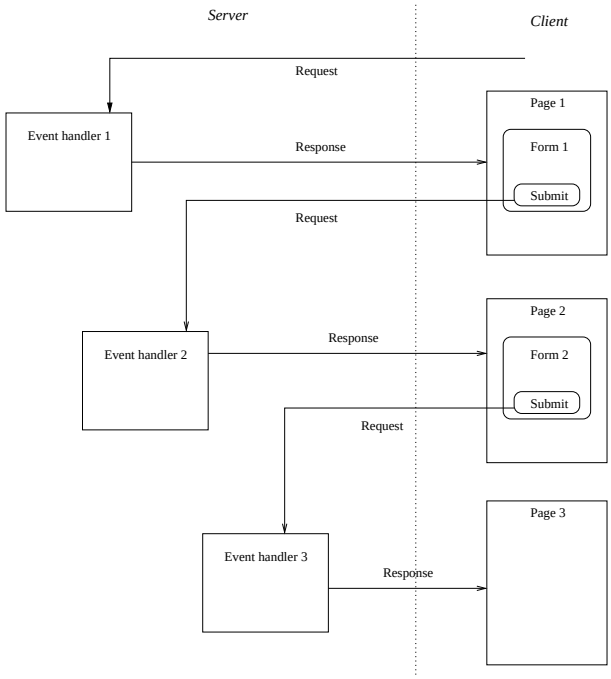


Figure 2.2: An example of how nested event handlers is used to do interactions with a client.

Sessions as Lexical Scopes

An interaction sequence in Bigwig is encapsulated in a session. A session is a lexical scope where interactions that are performed inside the same scope share data. This makes it possible for the developer to see which interactions with the client, that share data. This is not possible when a session is implemented as global state, due to the lack of a central overview of the entire application. Likewise it is not possible when event handlers are used for sessions. The reason is that the interactions with a client is not gathered at one place, but spread across event handlers. In Bigwig sessions can be defined inside a service. A service defines a lexical scope that consists of sessions. The reason for introducing the service is to encapsulate related sessions. Two sessions defined inside the same service can interact with each other and share data. Sessions in different services cannot interfere with each other or share data.

In Figure 2.3, this approach to sessions is shown. Here it is seen that the program is continued from the place where it left with the last response. In Figure 2.3 it is shown that *Page1* and *Page2* share data. In the same way it is illustrated that *Page3* and *Page4* share data. The pages in the figure represents interactions with the client. There are two essential things to be noted from this figure. First, interactions in one session cannot access data from

other sessions. Second, interactions within the same session share data and the flow of the interactions with the client are placed in the same program file.

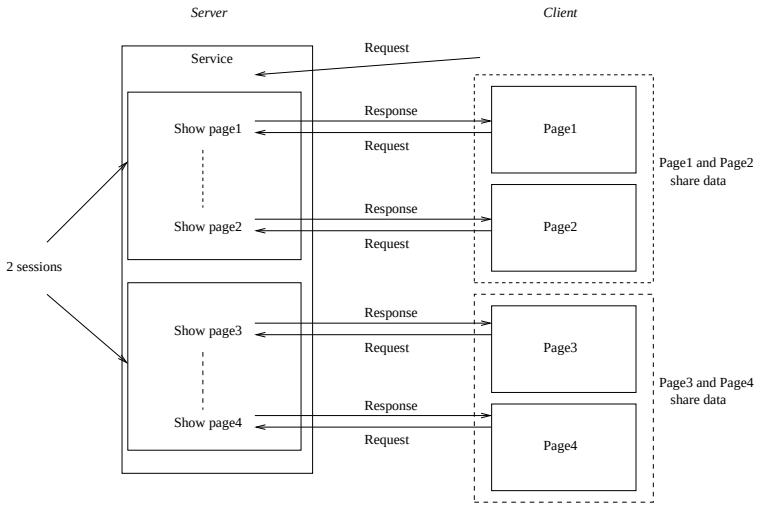


Figure 2.3: An example of how lexical scope is used to interact with a client.

Introducing sessions as a lexical scope requires more server processes. Since several interactions with a client is conducted as part of the activation of one SLAML session, the evaluation of such a session takes longer. Thereby it will use a server process for a longer period of time. A need for more server processes results in increased requirements for CPU power and memory usage. The amount of extra CPU power and memory usage needed can be determined by considering the amount presently used for a server process.

2.1.2 Design of the Session Framework in SLAML

In the above, three different session concepts have been discussed. It is decided to base the session concept in SLAML on the session concept from Bigwig (i.e. a new primitive encapsulate the interactions with a client by the use of lexical scope). The main reason for this is that the developer is able to see the whole flow of a session. Another reason for choosing Bigwig's concept of sessions, is that the developer is able to share data between the encapsulated interactions with the client. This gives the developer a better overview of a session. The reason for not choosing the approach found in WASH/CGI and PACKS/HTML is that the flow of an application is not expressed as clearly in these approaches as in session as lexical scope. The idea of having the flow of a session in one encapsulation - rather than as global state or separate event handlers - makes it explicit which pages share data.

Sessions in SLAML

In order to encapsulate an interaction sequence with a client, a new primitive is constructed. This primitive is called `slaml-session` (page 113), inspired by the session primitive in Bigwig. Interactions inside the same `slaml-session` can share data, by using variables in

Scheme. The lexical scope inside a session represents the flow of an interaction with a client. In order to activate a `slaml-session`, the `slaml-activate-session` (page 115) primitive is used. `slaml-activate-session` activates a session, and thereby starts the interaction with the client.

In some cases it is necessary to send parameters to a `slaml-session`. Imagine a session where the name of the person - who is logged in - is placed on the top of each page in a session. In such an example the possibility of sending parameters to a session is a great advantage. To obtain flexibility on a session, as is needed for sessions to be reusable, it is important that sessions can take parameters. Parameters are passed to `slaml-sessions` in the way seen in Figure 2.4

```
(slaml-activate-session
   (slaml-session (sessionparm)

    ;contents of the session

   ) 'sessionparm person-name
 )
```

Figure 2.4: An illustration of how a `slaml-session` is declared and activated.

In this example *person-name* is passed to the `slaml-session`. Declaration of a `slaml-session` is also shown in Figure 2.4. A `slaml-session` always takes one parameter. Parameters can be send to a `slaml-session` by evaluating `slaml-activate-session` with the `sessionparm` attribute. The attribute value is the parameter passed to the activated session. A `slaml-session` returns the last expression evaluated in the session. The reason for this is, that this is how return values are specified in Scheme. Sessions in the language are first class objects allowing the same possibilities with sessions as with functions.

Implementation `slaml-activate-session` is implemented as a function that extracts session parameters for the optional attributes to `slaml-active-session`. The slaml-session which is required as a parameter to `slaml-activate-session` is executed with the optional session parameters if any where supplied. `slaml-activate-session` is defined as follows:

```
(define (slaml-activate-session sessionfunc . args)
  (let ((parms (slaml-get-sessionparm-parm args)))
    (sessionfunc parms)
    )
  )
```

It always takes one parameter and possible more. First is extracts the session parameters from the optional parameters. Then it activates the *sessionfunc* with the optional parameters.

Client Interaction in SLAML

In order to interact with a client a primitive is needed to show a page to the client and return the data from the client. In Bigwig this is done by the primitive called show. In SLAML this primitive is called `slaml-show` (page 115). What `slaml-show` does is to send a page to the client and receive the data submitted - by the client - from that page. The data returned from the client is passed as a list of key/value pairs. When `slaml-show` returns with a request from the client, the `slaml-session` is continued from the place where the `slaml-show` that showed the page to the client is issued.

A page in SLAML is represented by a primitive called `slaml-page` (page 114). `slaml-page` must take one argument and return the HTML string to be presented to the client. This can be done by writing the HTML string manually. But a more appropriate way is to use the `html` convenience function from the LAML mirror available from [lam01] for generating the HTML string. A `slaml-page` can be written as a `lambda` function from Scheme, since this allows for passing parameters to the page. Parameters are however not passed in this way. Instead parameters to a page are passed as a list. This list is specified with the `pageparm` attribute - with an associated parameter list - to the `slaml-show` primitive. The reason for passing parameters in this way, is that this makes it possible to send more parameters by wrapping them in a list. The reason for using the `slaml-page` rather than a `lambda` is to get a better understanding of a program. A developer seeing a `slaml-page` is less in doubt of the nature of the function than if it was a `lambda` function. For an example consider the following:

```
(slaml-show
 (slaml-page (pageparm)
  (html
   (head (title "The title"))
   (body
    "The page parameter: " (car pageparm)
    (br)
    )
   )
  ) 'pageparm (list "Parameter one" "Parameter two"))
```

Figure 2.5: An example of how a `slaml-page` is declared and shown.

In this example it is seen how a `slaml-page` is shown to the client. Notice the `slaml-page` which takes a list of parameters. How data is returned from the client is discussed in Section 2.1.3.

To allow for defining sessions and pages in SLAML two new primitives are introduced. `slaml-define-session` (page 114) is used to define sessions. Likewise `slaml-define-page` (page 114) is used to define pages. Both of these primitives are similar to the `define` primitive in Scheme. They have been created to allow the developer to better differentiate between the definition of pages or sessions and functions. This is useful if the developer has written

a large program. As soon as the `slaml-define-page` or `slaml-define-session` is seen the developer is not in doubt of what is being defined.

Implementation The `slaml-show` primitive is implemented as a function that activates a primitive in the mod_laml server module. This primitive - named `slaml-display` - handles the communication with the client. Once control is returned from this primitive, `slaml-show` activates a function that generates a key/value pairs list of the form parameters received from the client. The implementation of `slaml-show` is shown in the following. Note that functionality for validation is also included in this implementation. Validation is described in Section 2.3.

```
(define (slaml-show pagefunc . args)
  (slaml-display (pagefunc (slaml-get-pageparm-parm args)))
  (let ((parms (slaml-create-parm-lst (slaml-get-args)))
        (check (slaml-get-checkfunc-parm args)))
    (if check
        (check parms (slaml-get-checkparm-parm args))
        parms
        )))
```

SLAML Sessions Compared With Bigwig

Comparing this idea with the session concept in Bigwig, one difference is that Bigwig has a service layer - represented by the service primitive - that encapsulates one or more sessions. This service layer is not introduced as a primitive in SLAML, because a service in SLAML is represented by the entry point (represented by a single file) to an application initially requested by a client. This means that if sessions wants to share data they must be defined in the file representing the entry point.

Activation of sessions in Bigwig can be done in two different ways. Either by requesting a service containing the session to be executed, or by passing a parameter in the URL indicating the name of the session to be activated. When a service is activated it is explicitly stated in the service which session to start. When the session is passed as URL parameter each session - defined inside a service - can be activated individually. Only the first approach is supported by SLAML, the second is not. The latter approach requires accounting on the server of the sessions available to clients.

In Bigwig the show primitive is used to show an HTML page to a client, and a similar primitive is present in SLAML (`slaml-show`). The session primitive (`slaml-session`) is similar to session in Bigwig. However, the way that values are received from the `input` elements in an HTML form, is different. In Bigwig the developer specifies which variables the form parameters must be bound to. In SLAML all value are returned in a list, which contains key/value pairs of the attribute names and the attribute values entered by the client. The

reason for choosing this solution is that when handling large forms, it is cumbersome to specify all the relations between variables in a program and the `input` elements in an HTML form. Another and more important reason for this solution has to do with the way SLAML handle complex forms. Complex forms are described in Section 2.2.

2.1.3 Flow of a Session in SLAML

In this section the flow of sessions in SLAML is described. This is illustrated in Figure 2.6.

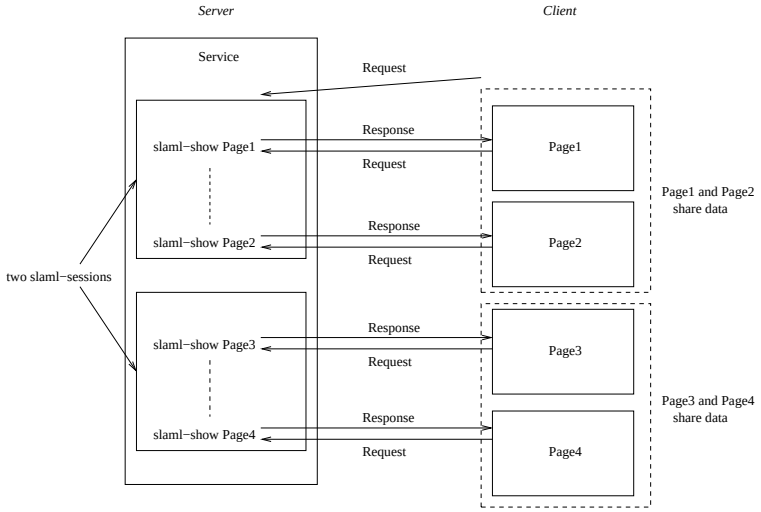


Figure 2.6: An illustration of the flow of two `slaml-session`'s, each containing two `slaml-show`'s.

When the client requests a SLAML program it is activated and evaluated. In the example in Figure 2.6 the SLAML program contains two `slaml-session`s. Here the first session is activated and *Page1* is shown to the client with the `slaml-show` primitive. The client can then send a new request to the server (by submitting the form on the HTML page), and the SLAML program will continue from where it was left when the last response was send to the client (the `slaml-show` primitive returns control to the surrounding program). Following this, *Page2* is shown to the client in a similar manner. Once control is returned from the `slaml-show` the first session is ended. Control is returned to the SLAML program and the second session is activated. This flow of control gives the developer the possibility to view several interactions (a session) with a client as one program rather than small separate programs.

The data - from a submitted HTML form - belongs to the lexical scope that the `slaml-show` was issued in. This gives an overview of the data flow in a program, as the data from a page is returned to the same place in the program flow as the page was send from. This gives the possibility to issue a `slaml-show` as a way to get data from the client. In this way the `slaml-show` function can be compared to other functions in the program. Whenever

the developer needs data from the client to proceed the calculation, the client is asked for data and the evaluation can resume. As the developer builds the HTML page to show to the client, it is known what data is returned from the user.

2.1.4 Example of the SLAML Session Framework

In this section a small example of the session framework is presented and explained. Additional and more complex examples are presented and discussed in Chapter 3. Figure 2.7 shows an example of how the various elements in the session framework are used.

```
(slaml-define-session
 simple-session
 (slaml-session (sessionparm)
  (let* (
         (simple-page1
          (slaml-page
           (pageparm)
           (html
            (head (title "The page 1 title"))
            (body
             "The page parameter: " pageparm (br)
             "The session parameter: " sessionparm (br)
             (form
              (input 'type "TEXT" 'name "inputdata")
              (input 'type "SUBMIT")
              )))))
         (page1-data
          (slaml-show
           simple-page1
           'pageparm "Parameter to page1"))
         )
    (slaml-show
     (slaml-page (pageparm)
      (html
       (head (title "The page 2 title"))
       (body
        "The page parameter: " pageparm (br)
        "The session parameter: " sessionparm (br)
        "Data from page1 before is: "
        (slaml-formparms-key->value 'inputdata page1-data)
        )
       )
      ) 'pageparm "Parameter to page2"))))

(slaml-activate-session
 simple-session
 'sessionparm "Parameter to session")
```

Figure 2.7: An example of using the elements in the session framework.

This example shows a simple session. The session defined is named *simple-session* and is activated in the last expression. Note that parameters are passed to *simple-session* as specified by the `'sessionparm` attribute. When the session is activated, it starts by binding a `slaml-page` expression to *simple-page1*. *simple-page1* represents an HTML page with information about the session parameter, the page parameter and it contains one input element. The next step in the evaluation is to show the *simple-page1* page, and bind the data returned to the variable called *page1-data*. Notice that *simple-page1* is passed a parameter specified by the `'pageparm` attribute to the `slaml-show` function. The last thing done is to show a `slaml-page` that also takes a parameter. On this page the page parameter, the session parameter and the value that is entered on *simple-page1* are shown. Notice the use of the `slaml-formparm-key->value` (page 117), which is used for extracting the value of the *inputdata* attribute from the *page1-data* list.

2.1.5 Solution to the State Handling Problem

In this section it is discussed how sessions in SLAML solve the *State handling* and the *Reusability* problem discussed in Section 1.1.1 and Section 1.1.4. The reason for introducing the session concept in SLAML, is to solve these problems.

Solution to the Control Flow Handling Problem

The solution to the *Control flow handling* problem is inspired by Bigwig and the ideas introduced there, where it is a primitive in the language that presents a page to a client. This primitive is called `slaml-show` in SLAML. Furthermore Bigwig inspired us to let program control return to the place in the program where the `slaml-show` primitive is activated. This results in the developer being able to see the flow of an application in the program code of the application. As a primitive is introduced to show a page and return data from the client, the program continues from the place in the program where the `slaml-show` primitive is activated. A `slaml-show` in the program can be considered as any other function in terms of understanding the program. It is a function that is evaluated and returns the result of the evaluation, which is a list containing the information received from the client.

Solution to the Data Flow Handling Problem

The *Data flow handling* problem is solved along with the *Control flow handling* problem, as parameters already received from the client can remain on the server. The only parameter handling that is necessary is to ask the client for data and receive the parameters. Once the parameters have reached the server they exist when the next request comes from the client. This means that the chosen solution relies on storing state on the server side. This results in easier parameter handling than in CGI. The reason is that the developer does not need explicitly to send all the parameters to the client and receive them on the server to maintain state. There is also problems with this solution as it requires space on the server for storing

the state. Furthermore there is a security concerns to be considered when storing the state on the server. It must not be possible for one client to access the state of another client.

2.1.6 Solution to the Reusability Problem

The problem of being unable to reuse a number of related pages as one unit has been solved by introducing sessions as first class objects. In SLAML it is possible to define a session and later activate it, thereby allowing the developer to activate a session on demand. This means that the developer is able to execute a number of pages following each other and receive data from the session. Being able to receive information from a session on the same level as it is possible from a page means that there are little difference between invoking a session and a page to return some data. As an example the developer can freely choose to develop a session or a page to receive some specific data from the client.

2.2 Complex Forms Framework

As presented in the Analysis (see Section 1.1.3) there is no good solution to maintain data in a complex structure when it has been send to the client. This lead us to our hypothesis:

Hypothesis 2:

> ***It is possible to construct a framework that helps the developer to build, present and update complex structures.***

Three possible ways of representing an HTML form as a complex structure has been identified:

1. A data structure
2. A language
3. A paradigm

A nested lists approach is used to represent a complex structure as a *data structure*. To represent a complex structure as a *language*, an embedded domain specific language is considered. Last, an object oriented approach is used to represent the *paradigm* way of representing a complex structure. Each approach is presented in the following section. Note that the following section serves as a presentation of possible solutions to the *Complex forms* problem. The decisions made in order to design the actual solution to the *Complex forms* problem are presented in Section 2.2.2.

2.2.1 Design Considerations

In this section, solutions to how a complex structure can be created in order to be represented as an HTML form are considered. The solution must fit well in the context of this project, namely Scheme, LAML and mod_laml. Recall that the *Complex forms* problem is split into three steps (see Figure 1.10 on page 17). These steps are:

1. Creating the complex structure
2. Representing the complex structure as an HTML form
3. Updating the complex structure with data from the client

Creating a complex structure means, that the developer creates a complex structure in the programming language. This is the structure, that it is of interest to get filled with data from the client. *Representing* the complex structure as an HTML form is the second step. This is done, in order to receive the data from the client. The third step, *updating*, is where the complex structure is updated with the information from the HTML form. Updating the complex structure is done on behalf of the form parameters, which are placed in a key/value pairs string. Each of the three possible ways of handling an HTML form - nested lists, embedded domain specific language and object orientation - is considered in relation to the three steps, that makes up the *Complex forms* problem. These are the steps just presented, namely *creating*, *representing* and *updating*.

Nested List Approach

This first approach relies on lists in Scheme. The reason for considering a nested list approach, is that both data and program are represented as lists in Scheme. This means, that a developer working with Scheme, is familiar with lists and list syntax. Furthermore, any first-class value can be a list element, so there are only few requirements to list elements.

The first step in the *Complex forms* problem, concerns the *creation* of the complex structure in the programming language used (Scheme in the context of this project). The person structure from Section 1.1.3, is in the following written by the use of nested lists:

```
(person
  (name
    (first-name "")
    (last-name ""))
  (address
    (country "")
    (city
     (city-name "")
     (postal-number ""))
    (street
     (street-name "")
     (house-number "")))
  (email "")
  (phone "")
  (age "")
 )
```

This is an example of a nested list structure, which it is of interest to get filled with information from the client. Three types of elements exists in the list; symbols, other lists and strings. The strings represent the values filled into the HTML form by the client. The strings

are empty (""), since the nested list structure has not been filled with information from the client. The symbols (the first element in each of the lists) are used to specify information about the other elements in the list. E.g. the symbol *name* indicates, that the following elements in the list, makes up a name. Using symbols, strings and lists as elements, a complex structure can be created.

The next step in solving the *Complex forms* problem is to *represent* the nested list structure as an HTML form. To make the task of getting the HTML string representation of a nested list structure simple, a function is used. This function takes the nested list structure as a parameter, and returns the HTML form representation. However, there are no elements in any of the nested lists in the person structure, that specifies how the structure is represented as an HTML form. Two possible ways in handling the HTML layout of a nested list structure have been identified. These are, layout by:

1. attributes
2. a style sheet

Layout specified by attributes, means that the HTML layout information about the different lists in the nested list structure is added as attributes to a list. Consider the list tagged *city-name* (the list, which has the symbol *city-name* as the first element). By adding an attribute named *type*, the HTML representation of *city-name* can be specified. E.g. does *(city-name 'type "TEXT")* specify, that the *city-name* list must be presented as an HTML `input` element of the type `TEXT`. By specifying a *type* attribute to all of the lists, HTML layout information is embedded in the nested list structure. Since HTML layout information is embedded in the nested lists, the nested list structure is mixed with data and information about the HTML layout.

By using a style sheet instead of attributes to specify the HTML layout of a nested list structure, the nested list structure is separated from the HTML layout information. The style sheet is defined external to the nested list structure; e.g. in another list. The style sheet list, can be an association list, where each symbol from the nested list structure is associated HTML layout information. This means, that the list tagged *city-name* has an entry in the association list. This entry looks like *(city-name . "text-input")*. Using a style sheet supports separation between data and layout, since the data is represented in the nested list structure whereas layout is specified in an external style sheet list.

The last step in the *Complex forms* problem is to *update* the nested list structure with the information received from the HTML form. In order to update a nested list structure with data from an HTML form, both the structure and the data from the client must be present at the server. Because of the session framework already designed, the Scheme environment will survive interactions with the client. This means, that the nested list structure does not need to be stored in an hidden HTML element or on the servers file system, in order to be present after a request. By comparing the keys - in the key/value pairs received from the client - with the names in the nested list structure, the nested list structure is updated with

the values. This task is handled by a function, and when given a nested list structure and the corresponding form parameters, the function returns the updated nested list structure.

A great advantage with the nested list approach is, that much functionality for doing list manipulation is present in the Scheme programming language. This covers functionality to get the head and the tail of a list (`car` and `cdr` respectively) together with functionality that supports creating and extending lists (like `list`, `cons`, `append`, `map` and `length`). This functionality helps the developer to create and work with lists.

A problem with the nested lists approach is the way HTML layout is handled. The HTML layout information is specified by type information, which is used by the function that generates the HTML layout. This makes it impossible for the developer to specify a customized HTML layout, e.g. specify that the `input` elements must be placed in an HTML `table`. The reason this is a problem, is that a type does not contain information about the relation between elements. Instead, type information is only related to a single element.

Embedded Domain Specific Language

Instead of using a nested list approach to solve the *Complex forms* problem, an embedded domain specific language can be used. This approach is inspired by the paper, *"Little Languages and their Programming Environment"* [CGKF02]. A *domain specific language*, is a programming language, that is developed to solve problems in an specific domain. In the context of this project, the *domain* is complex structures and HTML forms. The problem in this domain, is the *Complex forms* problem. That a language is *embedded*, means that it is implemented inside another language (a host language). This means, that the interpreter in the embedded language can rely on features in the host language when it is implemented. The embedded domain specific language used to solve the *Complex forms* problem, is named *sfl* (Small Form Language), and the host language is Scheme. *sfl* is in the following considered in relation to *creating*, *representing* and *updating* a complex structure.

To *create* a structure by using *sfl*, means to write a program in *sfl*. A *sfl* program must be interpreted by the *sfl* interpreter. Therefore, a primitive must exist in Scheme, which escapes from the Scheme interpreter into the interpreter for *sfl*. This primitive is named `sfl`, and an example of how to write a program in *sfl* is presented below:

```
(let ((complex-structure
        (sfl
          (sfl-collection "person"
            (sfl-collection "name"
             (sfl-text-input "first-name")
             (sfl-text-input "last-name")
            )
            (sfl-collection "address"
             (sfl-text-input "country")
             (sfl-collection "city"
              (sfl-text-input "city-name")
              (sfl-text-input "postal-number")
```

```
            )
            (sfl-collection "street"
             (sfl-text-input "street-name")
             (sfl-text-input "house-number")
            )
           )
           (sfl-checkbox-input "email")
           (sfl-checkbox-input "phone")
           (sfl-text-input "age")
        ))))
    ;complex-structure can now be used
)
```

The above example is in the following discussed in relation to the syntax, the return value and the primitives in *sfl*. The syntax of a *sfl* program, is similar to the list syntax used in Scheme. Alternatively a syntax with infix notation (instead of Scheme's prefix) and curly brackets (instead of parenthesis) can be used. However, no reason for changing the syntax is seen. It will only be an irritating requirement, that the developer must change syntax in the middle of a Scheme program. However, another syntax indicates that the developer is using *sfl*, but this can easily be seen because of Scheme's prefix notation (the first word encountered when using *sfl* is the `sfl` primitive). Since a *sfl* program is embedded in a Scheme program, the surrounding Scheme program expects to get a return value from *sfl*. In the above example, this value is stored in a variable named *complex-structure*. As the name - *complex-structure* - indicate, a complex structure is returned from a *sfl* program. This complex structure can be a nested list structure or an abstract syntax tree. The primitives in *sfl* are discussed in the following, in relation to the HTML representation.

Specifying the HTML *representation* of a complex structure programmed in *sfl*, is done by using the primitives in `sfl`. In the nested list approach the developer has to specify HTML layout of the nested list structure, by the use of attributes or a style sheet. In *sfl*, the HTML layout information is indicated by the names of the primitives. E.g., the primitive *slt-checkbox-input* indicates, that an *email* (from the example above) is an HTML `input` element of the `CHECK` type. However, the *sfl-collection* primitive, does not specify any information about HTML layout. A solution to this problem, is to extend the interpreter in *sfl*, to recognize LAML like functions. This means, that there is a mapping between *sfl* primitives, and LAML mirror functions. E.g. *sfl-br* maps to the `br` mirror function in LAML. The reason for adding *sfl* to the names of the LAML mirror functions, is to specify that it is not possible to use LAML - and Scheme - functions directly in the embedded language. The following example illustrates how a *sfl-collection* can be presented as an HTML `table`, if the HTML layout specification is embedded:

```
(sfl
 .
 .
 (sfl-collection "street"
  (sfl-table
   (sfl-tr
    (sfl-td
```

```
     (sfl-collection "city"
      (sfl-text-input "city-name")
      (sfl-br)
      (sfl-text-input "postal-number")
      ))))
  )
 .
 .
 )
```

Since HTML layout information is not connected to the way the complex structure is represented in Scheme, the return type from `sfl` is changed. Both the HTML layout information and the complex structure is present in the *sfl* program. Therefore, a *sfl* program returns the complex structure (without HTML layout information) and its HTML representation.

Updating a complex structure returned from a *sfl* program, is handled in the same way as with the nested list approach. Recall, that this was done by sending both the complex structure and the form parameters to an update function. This function returns a complex structure, containing the values from the form parameters.

An advantage with an embedded domain specific language, is that a language is created to solve a specific problem. In this project the problem is the *Complex forms* problem. The introduction of specific primitives allows the developer to use special designed functionality, which has the purpose of creating a complex structure. This makes it possibility to specify collections of elements, but also information about the HTML layout of the structure. A problem with the embedded language approach is, that the same embedded program cannot be changed after it has been evaluated. This results in the impossibility to change the complex structure created by the embedded program. Another problem is, that the HTML layout is included in the embedded program. If the embedded program is large, it makes it difficult for developer to maintain the overview of what is HTML layout information and what is structure.

The possibility to specify the HTML layout by using LAML like functions, is an advantage in relation to the nested list approach. In the nested list approach, the developer cannot customize the HTML layout relation between the different elements, since HTML layout is specified by type information. However, a nested list structure can be bound to a variable in the Scheme environment, which allows the developer to manipulate the structure when wanted. This is not possible in the embedded language approach, since a variable bound to an embedded program will result in the variable being set to the return value of the program. Thus the structure (in the form of a program) cannot be manipulated when wanted.

Object Oriented Approach

Instead of creating a structure as nested lists or in an embedded language, an object structure can be created. Just as with the other approaches, an object structure must contain information about the composition of elements. Such a structure can be handled with the

Composite Design Pattern[ERRJ95], as this is used to represent part whole hierarchies. An example of an object structure based on the Composite Design Pattern is seen in Figure 2.8. Two types of classes exist in the Composite Design Pattern, namely the composite and the leaf class.

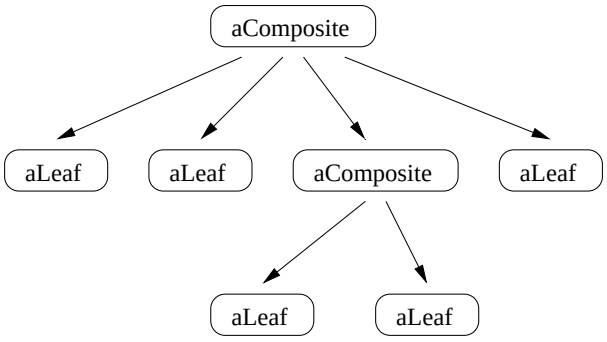


Figure 2.8: An example of an object structure based on the Composite Design Pattern[ERRJ95]. *aComposite* is an instance of the composite class, and *aLeaf* is an instance of the leaf class.

A composite object can represent a collection of composite and leaf objects, like a *sfl-collection* from the embedded language approach can represent other collections or basic elements. A leaf cannot represent other objects, but instead represent the basic entity in the structure. Here a composite structure is used to represent an HTML form, so a leaf must represent a single HTML `input` element.

With the classes introduced, it is possible to *create* an object structure, which must be filled with data from a client. Such an object structure, is created by linking objects together. This is done by using nested constructors, when the objects are created. Each name in the following example, corresponds to the initialization of an object.

```
(aComposite (aLeaf) (aLeaf) (aComposite (aLeaf) (aLeaf)) (aLeaf))
```

The above is an initialization of the object structure presented in Figure 2.8. The root element has four children, where one is a composite object and the others are leafs. The composite child has two children, which are leafs. The relation between the objects in the above object structure is specified at initialization time, namely by the use of constructors. An object structure can also be specified by message passing. This approach is cumbersome, since all the objects must be linked together by passing individual objects as arguments to functions on other objects. An example is presented below.

```
(define root (aComposite))

(add (aLeaf) root)
(add (aLeaf) root)
```

The *add* function takes two objects as argument, and the first argument is added as a child of the second. *aLeaf* returns a leaf object, and *aComposite* returns a composite object. As seen in the above example, it is more comprehensive to create an object structure in this way, than if a constructor is used. To create the same object structure by using a constructor, the developer writes: *(aComposite (aLeaf) (aLeaf))*.

To create an HTML form *representation* of the object structure, a function that recursively traverses the object structure and performs the layout is used. On behalf of the type of an object (a composite or a leaf), it is determined how an object is presented in an HTML form. To represent the various types of HTML `input` elements, specialization is used on the leaf objects. This allows the leaf object to map to an HTML `input` element. By using the object types to determine the way an object is presented as HTML layout, it is not possible to perform HTML layout on composite objects. The reason for this is, that a composite object does not map to an HTML element. A solution to this problem is to specify HTML layout as a property of an object. This allows the developer to specify HTML layout of a composite object. When the HTML representation of the object structure is created, the HTML layout property on each object is considered instead of the type of the object. However, it is only leaf objects, that can be represented as HTML `input` elements in an HTML form.

Updating the object structure is done on behalf of the form parameters. By the introduction of the session framework, the object structure is present on the server after a request has finished. The information from the form parameters are added the object structure by setting it on the individual object. When information from the object structure is needed, the value of the data property can be obtained from the individual objects.

When using the object oriented approach the developer is given two possible ways of creating an object structure. This can be done by using message passing or the constructor. This allows the developer to create some of the structure by using the constructor, and afterward add elements when wanted by using the message passing mechanism. This is e.g. beneficial, when the object structure is extended with more objects after is has been created and used. Furthermore, the developer is not forced to include HTML layout information when specifying the object structure. This information can be created external, and added the individual objects afterward. This allows a separation between HTML layout specification and the creation of the object structure.

In the nested list approach, it is possible to modify the nested list structure by using functions available in the Scheme language. This is not possible in the object oriented approach, since object oriented programming is not supported by standard Scheme. However, by specifying the HTML layout property on the classes, it is possible to customize the HTML layout. This is also possible in the embedded domain specific language approach, but not in the nested list approach.

2.2.2 Design of the Complex Forms Framework in SLAML

It is chosen to use an object oriented approach in the complex form framework. This might seem odd, as Scheme is mainly a functional language. The reason for choosing the object oriented approach is that it allows for division of specification of the layout and specification of the structure. This division allows for easy addition or removal of objects representing elements in the object structure. This means that it is easy to modify and reuse the structure throughout an application.

The reason for not choosing the domain specific language, is that once the embedded interpreter returns a result it is impossible to mutate this result to fit into another page. The flexibility to continue to use the complex structure throughout the application is not present. Another reason is that when the embedded interpreter is constructed, it needs access to the LAML functionality. However LAML functionality is not accessible from the embedded interpreter unless the interpreter is told how to handle it. There are two possibilities to support LAML functionality in an embedded interpreter. The first is to mirror the functions from LAML to the embedded interpreter. This can be done by mapping functions from the LAML library to the embedded language. Thereby, LAML functionality is available in the embedded language. The second possibility is to escape from the embedded interpreter and let the Scheme interpreter handle executing of the LAML functionality. But once the embedded interpreter is left, the Scheme interpreter do not know how to execute the domain specific language. Therefore there is a need for having all the domain specific language functionality outside the embedded interpreter, to enter the embedded interpreter again. This means there is a need for having a mix of functionality both outside the interpreter and inside the interpreter. By choosing an object oriented approach it is not necessary to escape to another interpreter to have functionality executed.

This section starts with a presentation of the three steps - *creation*, *presentation* and *updating* - that are performed, when using the complex forms framework in SLAML. Second is an introduction of the classes needed in the complex forms framework. Third, is a presentation of how object oriented programming is simulated in Scheme. This is included for the Scheme interested reader. Fourth, it is presented how *creation*, *presentation* and *updating* are performed in the complex forms framework in SLAML. Last, is an example of how the complex forms framework is used.

The steps that are taken when using the complex forms framework are presented in Figure 2.9.

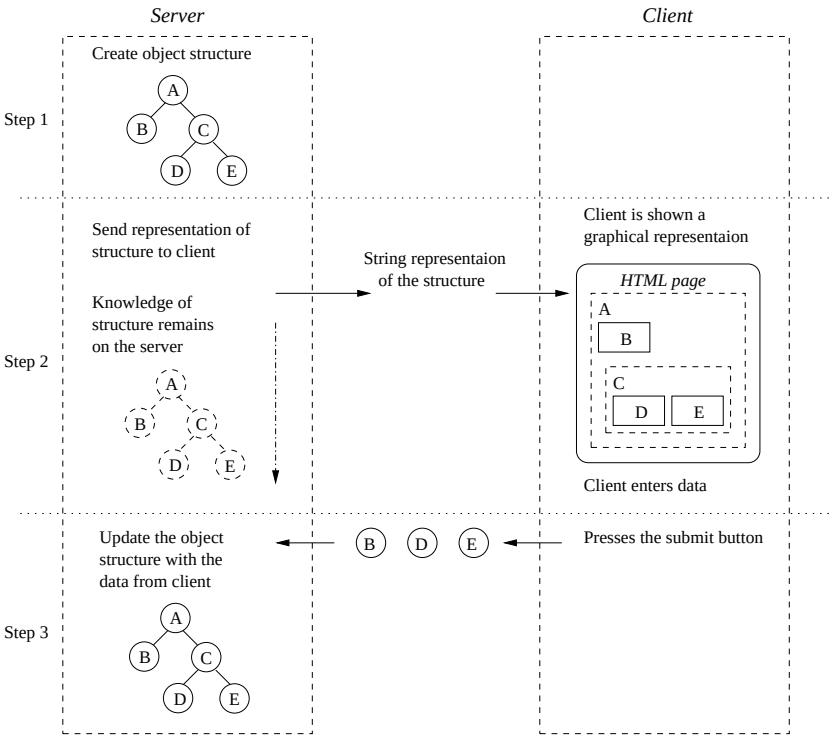


Figure 2.9: An illustration of the actions involved in handling the complex form, when presenting it to the client and updating it with the form parameters from the client.

Step 1 concerns the creation of the object structure. This involves the specification of the relation between the objects and the specification of the HTML layout of the object structure. *Step 2* covers the HTML representation of the object structure as an HTML form. While the HTML form is being presented to the client, the information about the object structure is present on the server. *Step 3* is related to updating the object structure with the form parameters received from the client. Recall, that the form parameters, is a key/value pairs string (an association list in Scheme).

The object model used in the complex forms framework has three classes. The first class is called `slaml-basic-element` (page 118) and represents a leaf in an object structure. The second class is called `slaml-element` (page 117) and represents a composition of objects. The last class is called `slaml-form-element` (page 119) and represents the root object in an object structure. An instance of the `slaml-form-element` class is also a composite object.

A reason for only introducing three classes is the focus on the structure rather than HTML layout, when creating the object structure. This is due to the increased abstraction obtained by separation of concern between the structure and the HTML layout of the structure. The reason for focusing on the structure is that layout is only associated with the structure and not part of the structure. Had focus been equally on layout and structure, the approach used in DOM[WG02] might be more appropriate. The reason is that all elements from HTML is available in DOM and thereby the layout of a page can be created using DOM.

Having separated the layout from the presentation the focus is on the classes needed to create the structure. There are three different expectations to the objects in the object structure. Since the object structure must be able to receive data, its representation must be rooted in an HTML form. This first expectation is a class representing the root of the objects structure. This is represented by the class named `slaml-form-element`. Since the structure is created with an expectation of receiving some data from the client, a class must represent the data from the client. This is the different HTML `input` elements. Therefore a class must represent an expectation of data. This class is called `slaml-basic-element`.

The last expectation regarding the structure is representation of the composite elements. With the above two classes the root and the leaf of the object structure are covered. A composite object as seen in the Composite Design Pattern can be used to represent the composition of several objects. Such an object allow addressing a group of objects as one. This class is called `slaml-element`. No additional expectations are present, and therefore no additional classes are introduced.

Once the structure has been build the layout functionality is created and associated with the individual object in the structure. In this way the structure is in focus. The presentation is a property on the individual objects. Each of the three classes are described in more detail in the following.

2.2.3 Complex Forms Framework in SLAML

To use the Composite Design Pattern to represent the classes in the complex HTML form, SLAML operates with three classes. These are explained in the following.

Slaml Basic Element Class

The leaf element in SLAML, represents the HTML `input` elements (see [W3C02d]). In the complex forms framework, the leaf class is called `slaml-basic-element`. Documentation and default values of this class are found in Appendix A.[1] An object of this type contains the attributes from the HTML `input` element (for a complete list see [W3C02d]). This means, that all attributes of the HTML `input` element are represented by the `slaml-basic-element` class.

As `slaml-basic-element` corresponds to the HTML `input` element, the HTML representation of this element is given. This is the case, since all attributes from the HTML `input` element are present as instance variables in the `slaml-basic-element` class. Therefore, it is possible to construct an HTML `input` element from the instance variables available in an object of the `slaml-basic-element` type.

[1] Please note, that the implementation of `slaml-basic-element` only supports the HTML `input` element, so there is no possibility to e.g. represent a `textarea` with an instance of the `slaml-basic-element` class.

The `slaml-basic-element` class can be subject to specialization. This is possible since a `slaml-basic-element` must represent a number of different HTML elements. Examples includes submit buttons, input elements, check boxes etc. (for a complete list refer to [W3C02d]). Two alternatives are considered. Either the developer is provided with a number of specializations of the `slaml-basic-element` class, or we provide updateable state on the objects representing its HTML type.

Providing a number of specializations is disregarded, as the difference between the specialized objects is the representation (the HTML `input` element represent by the objects). But since presentation is not part of the structure - but rather associated with the individual elements in the structure - there is no difference between the specialized classes. This suggests that specialization is not applicable. If the structure and the representation has not been separated the `slaml-basic-element` class is subject to specialization. An additional argument for not relying on specialization is that the alternative allows for an already instantiated object to change its HTML element type. This mutation makes it easy to change the HTML element type of objects already inserted into an object structure.

Slaml Element Class

The `slaml-element` class is the composite class. Objects of the `slaml-element` type can contain references to other objects (of the types `slaml-basic-element` and `slaml-element`). This makes it possible to make hierarchies of objects, that represents the structure shown as an HTML form to the client. This class is used when grouping objects.

The HTML presentation of an object of the `slaml-element` type, consists of the presentation of its children. However, it is not satisfactory to present the HTML form as a collection of HTML `input` elements. Instead, flexibility is needed to build a specific layout of the HTML representation of the object structure. Therefore, the HTML layout of a `slaml-element` is specified by a template where it is possible to place - in the relation to HTML layout - the children of the `slaml-element` as wanted. To represent the template of an object, a function representing the HTML layout of the `slaml-element` is created. This function is added as a property on the `slaml-element` it represents. The HTML layout function is discussed in Section 2.2.5.

Slaml Form Element Class

The `slaml-form-element` class is the root of an object structure in the complex forms framework. The reason that this class is needed is that the `slaml-element` class contains no information about the data related to the HTML `form` element. This information is present in the `slaml-form-element` class. A `slaml-form-element` cannot be a child of other objects, and must therefore be the root of an object hierarchy. All the attributes from the HTML `form` element are present in the `slaml-form-element` class.

The HTML layout specification of a `slaml-form-element`, is the HTML layout specification of its children together with the specification of the HTML `form` element surrounding them. This means that the `slaml-form-element` also consists of properties that specify the attributes to the HTML `form` element. This is discussed in further details in Section 2.2.5.

The next section (Section 2.2.4) presents underlying Scheme code, that is needed in order to simulate object oriented programming in Scheme. This is included for the Scheme interested reader and can be ignored if simulation of object oriented principles in Scheme is not of interest.

2.2.4 Object Oriented Programming in Scheme

It is possible to program object oriented in Scheme and different frameworks with support for object oriented programming in Scheme exists. Some of these frameworks are general and works with most Scheme systems. Others are written to a specific Scheme system.

Examples of object oriented frameworks that are used with a specific Scheme systems are Goops (The Guile Object Oriented Programming System)[LDJ02] and MzScheme's object[Fla02]. Goops is a framework for the Guile interpreter and is an extension to the basic Guile interpreter. MzScheme also contains an object framework which is part of the interpreter. An example of a more general object oriented framework is Meroon [Que02], which can be used in various Scheme systems.

Another possibility to program object oriented in Scheme is to use functions to represent classes and objects in Scheme (shown by Kurt Nørmark [Nør90]). This approach is the most portable as it is supported in all Scheme systems. The reason for this is, that it is build on `lambda` expressions.

It is chosen to simulate classes and objects in SLAML with functions, as this gives the best portability. The reason that portability is important, is that LAML is usable in many different Scheme systems and by making SLAML interpreter independent it is possible to use the SLAML framework on the same Scheme systems that LAML can be used. However, this approach might not be as efficient as an object system written for a specific Scheme system. This is not considered a problem as the complex form framework is more a proof of concept than it is a framework used for production. Another concern is the readability of the programs written in the SLAML framework. The programs written with pre-made object systems have a higher syntactical abstraction and therefore the programs are easier to read. When simulating object orientation this can be achieved by making syntactical abstractions on top of the simulation. The last problem considered with the chosen approach, is that all the mechanisms used in object oriented programming (inheritance, message passing, etc.) has to be implemented when needed. This is not a big problem in the SLAML framework, as only classes, message passing and a constructor are needed.

In the following it is explained how to define classes and create objects in Scheme. Furthermore it is explained how to use message passing to change the state of objects and how a

constructor is used to instantiate objects with others than the default values.

Classes and Objects in Scheme

Classes can be simulated in Scheme, by defining a function from which objects can be instantiated. When an object is instantiated it returns a function object, that serves as an interface to the object. An example of a simple class is the following:

```
(define (test-class)
  (letrec ((x 0)
           (get-x (lambda () x))
           (set-x (lambda (new-x) (set! x new-x)))
           (type-of (lambda () 'test-class))
           )
    (lambda (message)
      (cond ((eq? message 'get-x) get-x)
            ((eq? message 'set-x) set-x)
            ((eq? message 'type-of) type-of)
            (else (error "Message not found"))))))
```

Here the function *test-class* represents a class. An instance of the class is instantiated by evaluating the function. Evaluation of this function makes it return a function object. This function object serves as an interface to the object. The interface is a dispatcher that can call methods on the object. This dispatcher is activated by message passing.

Message Passing in Scheme

Once an object is instantiated, its state can be changed by sending messages to it. The messages that can be send to the object are specified in the dispatcher function, which serves as an interface to the object. Messages are send to the object in the following way:

```
(define new-object (test-class))

((new-object 'set-x) 10)
```

The first thing that happens in this example is that the *test-class* is instantiated and the resulting dispatcher function is bound to the name *new-object*. Next, the dispatcher is invoked with the message *set-x*. This results in the function associated with the *set-x* property of *new-object* to be returned. This function is evaluated with the value *10* as input. All this results in creation of an object and setting the x property to the value *10*.

It is preferable to create a function (often named *send*) to send messages to the objects. The reason this is preferred is that it provides a syntactical abstraction. This results in the following way to send messages to objects:

```
(send 'get-x new-object)
```

Here the *get-x* message is send to the object *new-object*. The result is the same; the function associated with the *get-x* property on *new-object* is returned. It is cumbersome to rely on message passing to change values on the individual objects. Therefore a constructor mechanism is created which allows for specifying values at instantiation time.

Instantiation of Objects in Scheme

By creating a constructor it is possible to specify arguments when an object is instantiated. In SLAML, XML syntax is used when instantiating the objects with others than the default values. An example is the following:

```
(define new-object (test-class 'x 10
                               'y 11))
```

In this example an object is instantiated and bound to the name *new-object*. *new-object* has two instance variables. One is called x which value is set to *10* at instantiation time. The other is called y and is assigned the value *11*. The reason for choosing this syntax, is that this is a XML like syntax, which is similar to LAML syntax. Another possible syntax is to rely on positional parameters, but this is not a good solution if there are many instance variables in the objects, especially if it is possible to specify all of them at instantiation time.

In this section it is shown that a constructor mechanism is created to allow a flexible instantiation of objects from classes. Message passing is introduced as a way to modify the individual objects. Furthermore it is illustrated how a function object - returned from a function representing a class - acts as an object.

2.2.5 Creating, Presenting and Updating Object Structures

In this section it is explained how an object structure from the three basic classes (explained in Section 2.2.3) is created. Next, is a description of how to add HTML layout to the objects in the object structure. Last, is described how the object structure is updated with the data received from the HTML form shown to the client.

Creating an Object Structure

To create an object structure to represent an HTML form, the first thing to do is linking the objects together. This can be done in two ways. The first is to create all the object in the object structure and afterward link them together by using message passing. This can be done by invoking the `add` method on an object. The `add` method takes one parameter, which is the object to add as a child. Another way to create the structure, is to do it at instantiation time by using the constructor. This is done in the following way:

```
(define person-form
  (slaml-form-element
   'name "person-composite"
   'elements
   (slaml-create-obj-lst
    (slaml-element
     'name "name-composite"
     'elements
     (slaml-create-obj-lst
      (slaml-basic-element
       'name "first-name-leaf")
      (slaml-basic-element
       'name "last-name-leaf")))
    (slaml-basic-element
     'name "have-car-leaf")
    (slaml-basic-element
     'name "submit-button-leaf"))))
```

This expression creates the structure in Figure 2.10 (see page 60). In this example three things are added to the SLAML elements. The first is the `slaml-create-obj-lst` (page 121) primitive. This primitive is used to create a list of objects to be added to another object. The second thing added in this example is the `elements` attribute. This attribute is used to specify the children of an object. The list of objects - to be added as children - must be created with the `slaml-create-obj-lst` primitive. In this way it is possible to create an object structure.

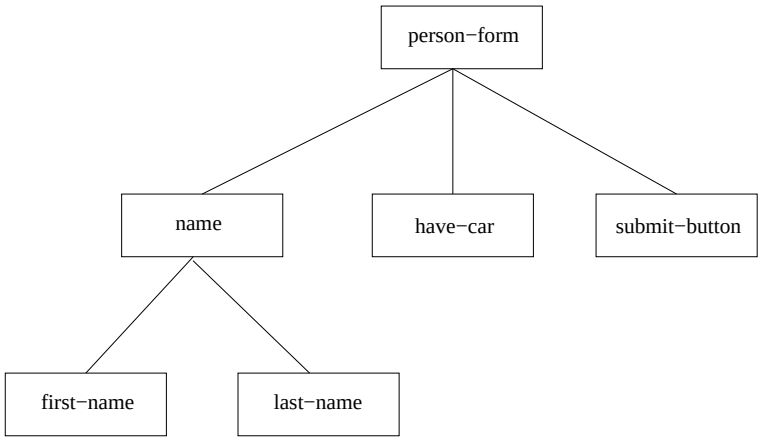


Figure 2.10: The person-form hierarchy.

The third thing added to the example is the name attribute. This attribute sets the name instance variable of the object. This serves to access an object once it is added as a child to another object. If the name attribute is not present, there is no way of identifying the object and discriminate it from other children with the same parent. Therefore, by giving

all objects of the same parent different names, this property is used to refer to a specific object. This is used when the HTML layout information is added to the objects. If the child objects have no names it is impossible to tell which element is to be placed where in the layout. This is conducted later in this section.

The last thing described in this section is the `id` of an object. An `id` is set on the objects of the type `slaml-basic-element`. The `id` on an object serves as the `name` attribute of the HTML `input` element when the object is presented - in an HTML form - to the client. The reason for having the `id` is to associate each HTML `input` element with an object of the type `slaml-basic-element`. The difference between the `id` property and the `name` property of an object is that the `name` is used when the programmer refers to an given child of a parent in the program. The `id` is used when the input from the user is placed in the object structure in order to update it. By letting each of the leaf objects in the object structure have an unique id and let the same id represent the name of an input element in HTML, it is possible to traverse the object structure and place the client data in the object structure. This is done by comparing the name of the HTML `input` element with the `id` attribute on the object. If they match, the data from the HTML `input` element is placed as data in the matching leaf object.

The `id` can be set by hand when the object is created. If no `id` attribute is specified when the object is created, an object is given a unique `id`. The reason for letting the developer change the unique `id` is that sometimes it is convenient to access the form parameters directly by name instead of using the objects to access them. This is not possible if the `name` attribute in the HTML element cannot be changed. And as the `id` property of an object corresponds to the `name` attribute on an HTML `input` element, it is necessary to allow the developer to change the `id`.

Adding HTML Layout to Object Structures

After the object structure is build, the HTML layout is added. This is done by associating a template to each object. The template represents the HTML layout of the object it is associated to. This means that a way to specify HTML layout information on each of the three classes in the SLAML framework must be present. This is discussed in the following.

Templates for the composite classes (`slaml-element` and `slaml-form-element`) are specified as a function. The reason for using functions to represent templates is that functions can return an HTML string representing the HTML layout of a given node in the object structure. Furthermore, it is possible to send parameters to the function telling which node in the structure to do layout on. This node serves as the parent node and by referring to the names of the children it is possible to place them in the layout. Beside this a function can contain additional HTML information and thereby represent more than the layout of the children. It can also contain HTML elements used for formatting the HTML form layout.

Another possibility is to let the HTML layout be represented by a list of children and let the order of the list be the order in which the elements are presented in the HTML form. In

this approach there can be no additional formatting of the children in the composite nodes of the structure. Therefore the first approach is chosen.

For the leaf class (`slaml-basic-element`) the layout is represented by setting attributes on leaf objects. The reason is that the leaf objects represents an HTML `input` element and there can be no HTML layout inside an HTML `input` element as it is a single tag. Therefore, all formatting information is the surrounding of these elements and this layout information is present in the composite elements.

Another possibility is to let each `slaml-basic-element` have a template associated to it as done with `slaml-element` and `slaml-form-element`. Then the developer will need to specify a template for the leaf elements and associate the HTML layout with a leaf node in the object structure. The first approach is chosen since it requires less associations between templates and nodes. The reason is that the leafs are self contained as the instance variables of the objects specifies the HTML layout.

A layout function for the *name-composite* object from the object structure on page 60 is presented in the following.

```
(define name-layout
  (slaml-layout (self args)
    (table
     (tr (td "Name" 'colspan "2"))
     (tr (td "First name")
         (td  (slaml-do-layout-child self "first-name-leaf")))
     (tr (td "Last name")
         (td (slaml-do-layout-child self "last-name-leaf")))
     'border "1")))
```

Figure 2.11: An example of a `slaml-layout` function.

In this example the `slaml-layout` (page 123) primitive is introduced. `slaml-layout` can be thought of as a `lambda` expression. A `slaml-layout` takes two arguments. The first parameter is a reference to the object to which this HTML layout function is associated. The second parameter can be send to the layout function when it is activated (when the `slaml-do-layout-child` (page 122) or `slaml-do-layout` (page 122) is called). To associate a layout function with an object the `layout` attribute is used at instantiation time.

```
(slaml-element
     'name "name-composite"
     'elements
     (slaml-create-obj-lst
      (slaml-basic-element
        'name "first-name-leaf")
      (slaml-basic-element
        'name "last-name-leaf"))
```

```
     'layout name-layout
     'layoutparm "simple string")
```

In this way the *name-composite* is specified to be presented by the `slaml-layout` function bound to the name *name-layout*. When the *name-layout* is called it is passed the string *"simple string"* as its second parameter. This is done by including the attribute `layoutparm` in the instantiation of the object. This parameter is not used in the `slaml-layout` function in Figure 2.11.

In Figure 2.11 a function called `slaml-do-layout-child` - performing the HTML layout - is introduced. Since a composite element like *name-composite* is responsible for layout of its children, it needs a primitive to express this. This is done by the function called `slaml-do-layout-child`. This function takes two arguments. The first is a reference to the object in which the child is located. In the example in Figure 2.11 this is *self*. The second parameter is a string representing the name of the child to layout. The return value of `slaml-do-layout-child` is a string representing the HTML layout of a given child.

As a `slaml-form-element` must represent an HTML form it needs more than a `slaml-layout` function to present itself. Besides a `slaml-layout` function, a `slaml-form-element` also needs attributes specifying the various properties (see [W3C02c]), to be able to represent itself. These attributes can be set on the `slaml-form-element` object at instantiation time.

To present an object of the `slaml-basic-element` type, the value of its `type` variable must be set. The reason is that this attribute specifies what type of HTML `input` element the object is representing (examples include hidden, text and password. For a complete list refer to [W3C02c]). In Figure 2.10 the *submit-button* is of type `submit` and the *have-car* is of type `checkbox`. This can be specified in the following way.

```
(slaml-basic-element
     'name "have-car-leaf"
     'type "CHECKBOX")

(slaml-basic-element
     'name "submit-button-leaf"
     'type "SUBMIT")
```

When the HTML layout has been specified for each element the HTML layout on the root object is activated in order to generate the HTML layout for the children. The HTML layout functionality of the root object is activated by the function called `slaml-do-layout`. This function takes one argument, which is the root object of the object structure to present. `slaml-do-layout` returns a string representation of the HTML form, which can be a part of an HTML page, as shown in the following.

```
(slaml-show
  (slaml-page (parm)
    (html
```

```
(head (title "A title"))
(body
  (slaml-do-layout person-form)))))
```

Here a `slaml-page` is presented where the *person-form* is included in the body of the page.

In this section it is described how a `slaml-layout` function is associated with each object of the types; `slaml-element` and `slaml-form-element`. This function allow generation of a flexible HTML representation of the individual objects. It is furthermore explained how the functions `slaml-do-layout` and `slaml-do-layout-child` is used in a `slaml-layout` function to activate the layout functionality on child objects.

Updating Object Structures

When the object structure is presented to the client as an HTML form the client can fill data in the HTML `input` elements. As explained, the HTML `input` elements in the HTML form represents the `slaml-basic-elements` at the server. When the client submits the HTML form, key/value pairs are returned to the server. These key/value pairs are used to update the object structure. As each of the HTML `input` elements has a unique id as name - and the `slaml-basic-elements` has the same ids - the object structure can be traversed and the values entered by the client can be assigned to the structure. This is done by using the function called `slaml-update-object!` (page 123). This function takes two parameters. The first is the root of the object structure, presented to the client. The second parameter is the key/value pairs returned from the client. Given these two parameters, the object structure is updated with the values from the request and the developer can then query the objects for data.

To allow operation on list structures rather than object structures, the `slaml-update-object!` returns a list representing the object structure as a tagged list. The reason for returning a list when the object structure is updated is that lists are the general data structure in Scheme. The resulting list structure is shown here.

```
(slaml-form-element
 (obj-name . person-composite)
 (slaml-basic-element
  (submit-button-leaf . ""))
 (slaml-basic-element
  (have-car-leaf . ""))
 (slaml-element
  (obj-name . name-composite)
  (slaml-basic-element
   (last-name-leaf . "my last name"))
  (slaml-basic-element
   (first-name-leaf . "my first name"))))
```

This example shows the list structure returned from `slaml-update-object!` given the *person-composite* object as first parameter and the key/value pairs - returned from `slaml-show`

- as second parameter.

The list structure returned from `slaml-update-object!` represents the object structure with *person-composite* as root element. The list is tagged with the name of the type of each of the three classes in the SLAML framework. The `slaml-form-element` and `slaml-element` tagged list structures consist of a key/value pair list as the first element. The key is *obj-name* and the value is the name of the object that is represented by this list. The rest of the list is the children of this object. These can be of the types `slaml-basic-element` and `slaml-element`.

The `slaml-basic-element` list structure consist of a key/value pair, where the key is the name of the object that the list represents. The value of the list is the string entered into the HTML `input` element by the client.

The list is tagged with the type of object that the list structure represents. It is not always satisfactory to use the type of the object as the tag. The reason is that the first symbol in a tagged list specifies the type of the content of the list. And as the content of the list is more than an element in SLAML, it is beneficial to allow the developer to specify a custom tag. This is done by introducing a property named `tagtype` to the object structure. This is illustrated in the following.

```
(define person-form
  (slaml-form-element 'name "person-composite"
   'tagtype "person"
   'action ""
   'elements
   (slaml-create-obj-lst
    (slaml-element 'name "name-composite"
     'tagtype "name"
     'elements
     (slaml-create-obj-lst
      (slaml-basic-element 'name "first-name-leaf"
       'tagtype "first-name")
      (slaml-basic-element 'name "last-name-leaf"
       'tagtype "last-name"))
     'layout name-layout)
    (slaml-basic-element 'name "have-car-leaf"
     'tagtype "have-car"
     'type "CHECKBOX")
    (slaml-basic-element 'name "submit-button-leaf"
     'tagtype "submit-button"
     'type "SUBMIT"))
   'layout person-layout))
```

Figure 2.12: Creation of the *person-composite* object structure with a tagtype for each object.

In Figure 2.12 it is specified that the list representing the *person-composite* object must be tagged with *person*. The *name-composite* must be tagged with *name* and so forth. The

resulting list is seen in the following.

```
(person
 (obj-name . person-composite)
 (submit-button
  (submit-button-leaf . ""))
 (have-car
  (have-car-leaf . ""))
 (name
  (obj-name . name-composite)
  (last-name
   (last-name-leaf . "my last name"))
  (first-name
   (first-name-leaf . "my first name"))))
```

This list is tagged in the way specified, and thereby the tagging is more specific than in the case where no custom tagging is used.

2.2.6 Example of the Complex Forms Framework

In this section an example of how to use the complex forms framework in SLAML is shown and explained. The example presents the use of the principles already introduced and will therefore not be described in all details. The example is split into three steps. First is the binding of `slaml-layout` functions to names. Second is the creation of the object structure and third is the presentation and updating. A larger example of the complex forms framework can be seen in Chapter 3.

The first thing done is to define two layout functions, which represents the HTML layout of *person-form* and *name-composite*.

```
(define person-layout
  (slaml-layout (self args)
      (table
      (tr (td "Person Information" 'colspan "2"))
      (tr (td (slaml-do-layout-child self "name-composite")))
      (tr (td (string-append "Do you have a car?"
                (slaml-do-layout-child self "have-car-leaf"))))
      (tr (td (slaml-do-layout-child self "submit-button-leaf")))
      'border "1")))

(define name-layout
  (slaml-layout (self args)
     (table
      (tr (td "Name" 'colspan "2"))
      (tr (td "First name")
          (td  (slaml-do-layout-child self "first-name-leaf")))
      (tr (td "Last name")
```

```
        (td (slaml-do-layout-child self "last-name-leaf")))
    'border "1")))
```

The HTML layout of the *person-form* is the HTML layout of the two composite objects in the HTML `form` (*person-composite* and *name-composite*). The first layout is the HTML layout of the *person-form*. This layout is bound to the name *person-layout* in the above example. The *person-form* object is responsible for doing HTML layout of the *name-composite*, *have-car-leaf* and *submit-button-leaf* objects. The second layout function is the layout function for the *name-composite* object. This layout is bound to the name *name-layout*. The *name-composite* object is responsible for doing HTML layout on the *first-name-leaf* and *last-name-leaf*.

The HTML presentation of the *person-form* is shown in Figure 2.13.

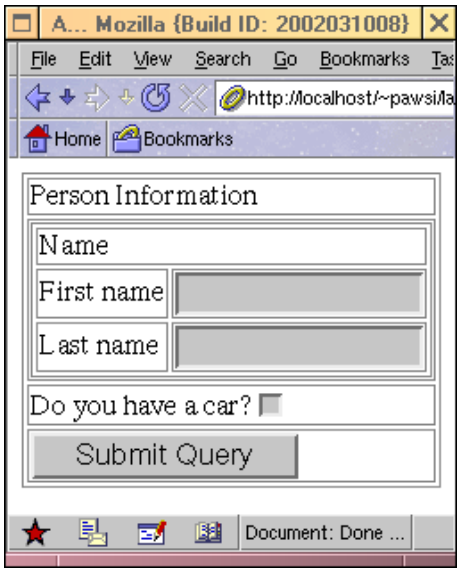


Figure 2.13: Screen shot of the complex HTML form example

The next step is to build the *person-form* object structure to present to the client. The object structure is seen in Figure 2.12 (on page 65). This figure shows how the object structure is build and that it is bound to the name *person-form*. The layout is also added to the object structure, by the use of the `layout` attribute. Next step is to present the object structure to the client. This is shown in the following.

```
(define page-data
  (slaml-update-object!
   person-form
   (slaml-show
    (slaml-page (parm)
       (html
        (head (title "A title"))
```

```
        (body
         (slaml-do-layout person-form)))))))

(slaml-show
  (slaml-page (parm)
    (html
     (head (title "A title"))
     (body
      (table
       (tr (td "Name" 'colspan "2"))
       (tr (td "First-name")
           (td (get-first-name-from-form page-data)))
       (tr (td "Last-name")
           (td (get-last-name-from-form page-data)))
       (tr (td "Car ?")
           (td (get-have-car-from-form page-data)))))))))
```

The first thing that happens in the above is that the HTML page with *person-form* is shown to the client. When the client submits the HTML form, the *person-form* is updated by the function `slaml-update-object!`, which returns a tagged list representing the object structure. This list is bound to the name *page-data*. The last thing that happens is that *page-data* is shown on a new HTML page. The declaration of the selector functions called: *get-first-name-from-form*, *get-last-name-from-form*, and *get-have-car-from-form* are not shown in this example. They are used to get data from the tagged list returned from `slaml-update-object!`.

2.2.7 Solution to Complex Forms Problem

In this section it is discussed how the SLAML complex form framework solves the *Complex forms* problem discussed in Section 1.1.3. The *Complex forms* problem is divided in three parts. The first concerns building the complex structure the second concerns presenting the complex structure and the last concerns updating the complex structure with data from the client.

Building the complex structure is done by creating objects using a constructor mechanism. By adding objects to other objects an object structure is build. The individual object in this structure contains a reference to its children. This object structure represents the HTML form that is to be presented to the client. A message passing mechanism is developed which allow activation of methods on the individual objects. This function is named `slaml-send`.

Presenting the complex structure is done by assigning the individual object a layout function that is responsible for generating the HTML representation of that particular object. These layout functions are created with the `slaml-layout` function. A function is introduced, which can activate the layout functions on its children. This is named `slaml-do-layout-child`. A special function named `slaml-do-layout`, is used to activate the layout function on the root element.

Updating the complex structure is done by invoking `slaml-update-object!` with the root object and the form parameters received for the client as parameters. This inserts the data entered by the client into the appropriate objects in the object structure. As an additional feature a nested list representation of the object structure is returned from the `slaml-update-object` function.

This gives the developer possibility to perform the three steps in the *Complex forms* problem.

2.3 Validation Framework

As shown in the analysis, input validation is not supported directly in HTML/CGI. It is of interest to build validation into the SLAML framework. Giving the developer the possibility to write validating functions in the same language as the HTML is generated, makes it possible to remove the need of external technologies. The design of validation is made on behalf of hypothesis three, from the problems definition:

Hypothesis 3:

> ***It is possible to construct a validation framework that helps the developer to validate data from the client.***

The goal is to give the developer the possibility to use a validation framework together with the SLAML session framework. In this validation framework, two possible levels of validation have been identified.

1. Page level
2. Object level

Page level validation is related to validation of a `slaml-page`, which is presented to a client by using the `slaml-show` function. Object level validation is related to validation of the composite and non-composite objects presented in the solution to the *Complex forms* problem. The reason for introducing validation on both the object level as well as the page level is the independent nature of the two frameworks. A developer relying on the complex form framework does not have to use the session framework and vice versa. If validation is not supported on both levels, then validation is impossible in some situations. In the following the design of the validation framework is presented. The considererations regarding the design are presented as alternatives to the actual solutions. The reason for not including a considerations section as in the rest of the design is, that much of the design is given, since it is almost dictated by the session framework and the complex forms framework, how the validation framework must be designed. This is the case, since we aim for consistent approaches among the different frameworks.

2.3.1 Design of the Validation Framework in SLAML

The following presents the choices made, regarding the design of the validation framework. Each of the two levels is handled individually. However, a general decision concerning client side or server side validation has been taken. The validation on both levels is handled on the server. Two reasons exist for this choice. First, the needed technology (Scheme) is not available as a scripting language that can be executed by the browser. Second, client side validation does not ensure HTML form input to be validated. We are aware, that server side validation can become a problem regarding bandwidth usage if many users are interested in validating information at the same time. A solution to this problem can be to validate input on both the server and client as it is done with Powerforms in Bigwig.

Page Level Design

It has been decided, that validation of a page, is done by extending the `slaml-show` function. The reason for not using a declarative manner as done by Powerforms, is that Powerforms relies on already defined "types". A declarative fashion of specifying valid types does not fit well in a weakly typed functional oriented programming language as Scheme. Powerforms also rely on JavaScript for evaluation, since a format is translated to JavaScript. Translating Scheme code to JavaScript seems to comprehensive, to allow client side input validation.

Extending `slaml-show` means, that if a `slaml-page` must be validated according to a check function, `slaml-show` must be called in the following way:

```
(slaml-show a-page 'check check-page)
```

The *check* attribute specifies, that the *check-page* function must be used to validate the page *a-page*. Here the syntax differs from XML like syntax, since *check-page* is a function and not a string. All HTML form information from the presented page (*a-page*) must be validated with the check function called *check-page*. For the check function to validate the data returned from *a-page* it must take the data as parameter. Before the form parameters are used as an argument to the check function, they are converted to a key/value pairs list. The `slaml-formparms-key->value` (page 117) can be used to extract the value of a key given the form parameters.

If a check function is specified with the `check` attribute to `slaml-show`, it returns the value returned by the check function. This gives the developer the flexibility to define the return value from `slaml-show`, which means, that a check function is not necessary limited to return `#t/#f` (true or false). The following is an example of a check function, which verifies that a number is between two values (*iname* is the name of the input element) which are specified by the developer.

```
(define (is-between? form-parms limits)
  (let ((num (slaml-formparms-key->value 'iname form-parms))
        (min (get-min limits))
        (max (get-max limits)))
```

```
(cond
 ((not-a-num? num) #f)
 ((and (< (string->number num) min) (> (string->number num) max)) num)
 (else #f))))
```

As seen the check function takes two parameters, the form parameters and an additional parameter (a list with two numbers, named *limits*). The check function first extracts the value entered in the HTML `input` element named *iname* with the `slaml-formparms-key->value` function and stores it in the *num* variable. Furthermore, the minimum and maximum values are needed. These are extract from the *limits* parameter (the parameter specified by the developer), by using the *get-min* and *get-max* functions. The values returned from these functions are stored in the variables *min* and *max*, respectively. The actual validation is handled in the `cond` special form: if *num* is not a number or not between *min* and *max*, `#f` is returned. Otherwise the value from the HTML `input` element is returned.

Parameters to check functions are specified in a similar way, as parameters to a `slaml-page` or a `slaml-session`, namely by specifying an attribute with the parameter as value. In the page level validation framework, the attribute is named `checkparm`. To activate the *is-between?* check function with 50 as the minimum value, and 100 as maximum, the values are wrappend in a list and specified as the value of the `checkparm` attribute:

```
(slaml-show a-page 'check is-between? 'checkparm (list 50 100))
```

This example illustrates how a check function take parameters. An example of how to use check functions on the page level is presented in Section 2.3.3. This example also presents how validation of dependencies is performed.

Object Level Design

Concerning validation of a complex form created by using the complex forms framework, two alternatives are considered. The first possible solution is to give the whole object structure to a function and let this function traverse the object structure to validate the object structure. In this approach the developer need to traverse the objects structure explicitly in order to validate the structure. The second, and the chosen solution is to add a check property to all the classes. This allows performing validation on each object in the structure and thereby the object level validation framework fits well with the complex structure represented by the objects. In this solution the traversing of the object structure is done implicitly by letting each node in the object structure be responsible for validating itself and the subtree it consists of.

The decision to extend the objects with information about validation, means that all types of objects must have functionality to set and get a check function. Furthermore, each class must contain a property (named `valid`) specifying if the object is valid. The attribute value used for the valid property is either `#t` or `#f`. The `valid` property value is determined by the check function assigned to the object. The default value is `#t`. Functionality to set (`slaml-set-valid` (page 125)) and get (`slaml-get-valid` (page 125)) the `valid` property

also exist. This allows the developer to query each object for the status of the validation performed on the object. Instead of using message passing when assigning information about validation to objects, the constructor is extended to allow this.

A check function can be assigned to an object by using message passing or by using the constructor. By using the constructor it is done as follows:

```
(slaml-element 'name "obj-name" 'check check-func)
```

In this example it is stated that the `slaml-element` named *obj-name* must be checked with the function bound to the name *check-func*. When this object is returned and has been updated the `valid` instance variable is set to `#t` if the validation went well and `#f` if the object data was invalid according to *check-func*.

Since check functions are used to set the `valid` property on objects, there are requirements to the check functions. They must always return either `#t` of `#f`, whereas a check function on the page level does not have requirements to the return value. Likewise, a check function used on the object level, must always take exactly one parameter. If the object is of the type `slaml-basic-element`, the parameter will be the string entered in the corresponding input element in the HTML form. If the object is either a `slaml-form-element` or a `slaml-element`, the parameter is a list containing the children. `slaml-get-element-from-list` is a function that can extract an object from the list on behalf of a name.

To see an example of validation on the object level, consult Section 2.3.3.

2.3.2 Flow of Validation

This section will in turn present flow of both the page level validation as well as objects level validation.

Page Level Flow

The flow of the page level validation contains two steps. First, get the form parameters from the client. Second, validate the parameters according to the check function. This flow is presented in Figure 2.14.

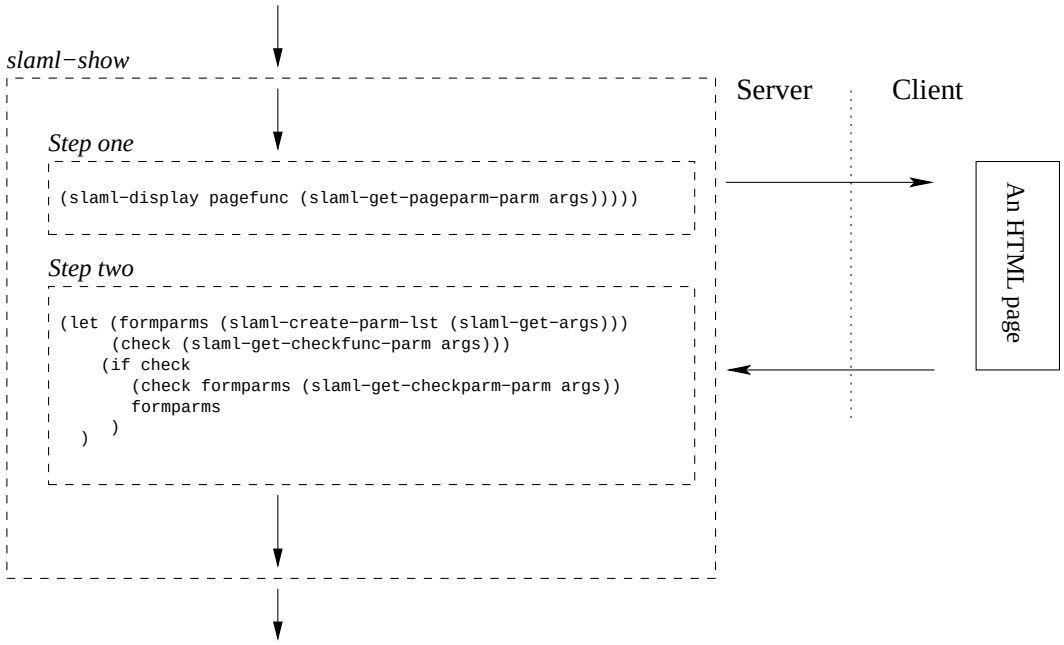


Figure 2.14: This figure presents two steps from the `slaml-show` function. `slaml-display` presents the string representation of a page to the client. The call to `slaml-display` will cause the evaluation to pause, until a new request is issued by the client.

Showing a page to the client, means that the application pauses on the server. The application will wait until the client submits information (*Step one* on Figure 2.14). The first thing done in *Step two*, is to create a key/value pairs list containing the form parameters (done with the call: *(slaml-create-parm-lst (slaml-get-args))*). The form parameters are stored in the variable *formparms*. Next, the check function is extracted from the optional parameters to `slaml-show` and stored in the variable *check*. This is done, by using the function *slaml-get-checkfunc-parm*, which returns the value of the `check` attribute and `#f` if none was specified. If *check* is false (the false branch in the `if` expression), *formparms* is returned. If a check function is specified (the true branch in the `if` expression), the check function is called with the return value of *(slaml-get-checkparm-parm args)* as a parameter. `slaml-get-checkparm-parm` returns the value of the `checkparm` attribute, and the empty list if none is specified. Recall, that the `checkparm` attribute is used to specify parameters to the check function, when `slaml-show` is called. All this means, that if a check function is specified when `slaml-show` is called, `slaml-show` returns the return value from the check function. Otherwise, `slaml-show` returns the form parameters.

Object Level Flow

Before an object structure can be validated, the objects in the structure must be assigned validation functions and the related form parameters must be present. Assigning validation functions to objects is done by message passing or by the constructor mechanism when the object structure is instantiated. In order to get the form parameters related to an object

structure, a page where the structure is represented as an HTML form must be presented to a client. This is done by calling `slaml-show` with the page containing the related HTML form as a parameter. When the client submits the HTML form, the form parameters are returned from `slaml-show`. These form parameters can then be used to update the object structure. This is done, by calling `slaml-update-object!` with the object structure and the form parameters as parameters. `slaml-update-object!` activates the validation functions in relation to the object structure. This means, that a check function on an object of the `slaml-basic-element` type is activated with the value entered in the related `input` element. Furthermore, a check function on a composite object (`slaml-form-element` or `slaml-element`) is activated with the objects children. This means, that a composite object is responsible for validating the subtree it represents. By setting the `valid` property of each object in the object structure to the return value of its associated validation function, the entire object structure is validated. Furthermore, information about the validation is added to the individual elements in the list returned from `slaml-update-object!`. This means, that it can be determined if an object is valid, by either searching the list returned from `slaml-update-object!` or by querying the individual object for the value of its `valid` property.

2.3.3 Example of Validation Framework

This section gives a small example on how to use validation on the page level and the object level. The scenario used in the examples is a single HTML page containing two input elements. The value entered in both elements must be numbers, and the value entered in the second input element must be the double of the value entered in the first. This yield for validation on single input elements, but also validation on the dependencies between the input elements. Additional examples can be found in Chapter 3.

Page Level Validation Example

The first thing done, is to create the HTML page (named *enter-double*) with the input elements. There are three `input` elements (two textual `input` elements and one submit `input` element) inside a form:

```
(slaml-define-page enter-double
  (slaml-page (parms)
    (html
      (head (title "Number test"))
      (body
        (form 'action ""
          (p "Enter a value:")
          (input 'type "text" 'name "value1")
          (p "Enter the double:")
          (input 'type "text" 'name "value2")
          (input 'type "submit")
        ))) ; end html
  ))
```

To check, if the values entered in the input elements are numbers, a function called *is-number?* is created. This function is called from the validation function (named *double-value?*), that is assigned to the *enter-double* page.

```
(define (is-number? str)
  (integer? (string->number str)))

(define (double-value? form-parms extra)
  (let ((val1 (slaml-formparms-key->value 'value1 form-parms))
        (val2 (slaml-formparms-key->value 'value2 form-parms)))
   (if (and (is-number? val1) (is-number? val2))
     (let ((num1 (string->number val1))
           (num2 (string->number val2)))
       (if (equal? num2 (+ num1 num1))
         form-parms
         #f ;not the double size
        )
       )
     #f ;not both numbers)))

(slaml-show enter-double 'check double-value?)
```

The *double-value?* function takes the form parameters as input (and the additional second input, which is not used here). The values entered in the two input elements are extracted, and stored in local variables (*val1* and *val2*). It is verified if each of the two values are numbers. If not, `#f` is returned. Otherwise, the dependency is checked. If the number entered in the second input element is not the double of the number entered in the first, `#f` is returned. If the dependency is fulfilled, the form parameters are returned.

Object Level Validation Example

This section will describe how validation is handled on the object level. The *is-number?* function from the previous example is used in this example also. The object structure consists of a `slaml-form-element` which has three `slaml-basic-elements` as children. Two of the basic elements are text `input` elements, and the third is a submit button.

```
(define double-form
  (slaml-form-element
    'layout (slaml-layout (parm)
      (string-append
        (p "Enter a value: ") (slaml-do-layout-child parm "value1")
        (p "Enter the double: ") (slaml-do-layout-child parm "value2")
        (slaml-do-layout-child "send-button")))
    'check check-structure-double?
    'action ""
    'elements (slaml-create-obj-lst
      (slaml-basic-element 'name "value1"
        'check is-number?)
      (slaml-basic-element 'name "value2"
        'check is-number?)
      (slaml-basic-element
        'type "SUBMIT"))))
```

Figure 2.15: Creating the object structure. Notice, that a `check` attribute is used to specify the check function - here *is-number?* - when an object is created.

As shown in Figure 2.15, two of the `slaml-basic-elements` are assigned a check function. This is done by the *is-number?* function from the page level validation example. There is also assigned a check function to the `slaml-form-element`, namely *check-structure-double?*. This function has the responsibility to validate the dependency between the children. The creation of the last check function and the presentation of the page is presented below:

```
(define (check-structure-double? children)
 (let ((obj1 (slaml-get-element-from-list children "value1"))
       (obj2 (slaml-get-element-from-list children "value2")))
  (if (not (and (slaml-get-valid obj1) (slaml-get-valid obj2)))
   #f ;they are not both numbers!
   (let ((num1 (string->number (slaml-get-data obj1)))
         (num2 (string->number (slaml-get-data obj2)))
     (equal? num2 (+ num1 num1)))))))

(slaml-update-object! double-form
 (slaml-show
  (slaml-page (parms)
   (html
     (head (title "Number test"))
     (body
       (slaml-do-layout double-form))))))
```

Figure 2.16: The definition of the *check-structure-double?* function and the presentation of the structure to the client. Recall, that `slaml-show` returns the form parameters, and that `slaml-update-object!` takes the object to update and the form parameters as input.

The *check-structure-double?* function takes the children element list as argument. It first extracts the object representation of the two input elements. These objects are stored

in the local variables *obj1* and *obj2*. Next, it is checked if the data entered in the input elements (validated by the *is-number?* function) are numbers. This is done, by using the `slaml-get-valid` function, which returns the value of the `valid` property from an object (recall, that the value of the valid property is determined by the check function on the object, and it is set when `slaml-update-object!` is called). If either is invalid, `#f` is returned. Otherwise the data (the values entered in the input elements) from the `slaml-basic-elements` are extracted with the `slaml-get-data` (page 127) function, and it is checked if the value from input element two is the double of the value from input element one.

The `slaml-update-object!` function updates the *double-form* object, and activates all the check functions assigned to the objects. To validate the HTML form according to the dependencies, `slaml-get-valid` is called with *double-form* as the argument. This will return the `valid` attribute set by *check-structure-double?*.

2.3.4 Solution to Input Validation Problem

In this section it is discussed how the validation framework solves the *Input validation* problem described in Section 1.1.2. The *Input validation* problem is considered on two levels, the page level and the object level.

Page level validation is done by supplying a validation function as parameter to a `slaml-show`. The validation function must be supplied as the attribute value to the attribute named `check`. The validation function must always accept two parameters. The first is the form parameters entered by the client. The second is an additional parameter which can be supplied by using the `checkparm` attribute with the `slaml-show` function. The value of the additional parameter is the value of the `checkparm` attribute, and the empty list if none is specified. The return value of the validation function on this level can be freely decided by the developer and is returned by `slaml-show`.

Object level validation is done by adding validation functionality to each object. A validation function on the object level must take one parameter. A valid property containing the status of the validation is introduced on the classes in the complex forms framework. The validation is performed once the object structure is updated with the form parameters. This is done with the `slaml-update-object!` function. It is a requirement that the validation function on this level returns a `#t` or `#f` value indicating if the data is valid.

2.4 Summary

In this chapter the design of the solutions to the problems mentioned in Section 1.1 is conducted. Three sections are presented which each corresponds to a hypothesis.

During the design of the SLAML session framework it is decided to base the SLAML session framework on the session concept from Bigwig (sessions as lexical scope), as this makes the

developer able to see the entire flow of a session. A primitive, `slaml-session`, is introduced to create a session. Another primitive, `slaml-show`, is introduced to show a `slaml-page` to the client. The control flow of the application is maintained as the data entered by the client is returned from the `slaml-show` primitive. Activation of a session is done with the `slaml-activate-session` primitive.

During the design of the complex forms framework it is decided to rely on object orientation, as this allows flexible mutation of the complex structure. Three different classes are introduced to represent objects in the object structure. These are `slaml-form-element`, `slaml-element` and `slaml-basic-element`. A constructor mechanism is created to allow a flexible instantiation of the classes. Layout concerns of the object structure is handled by assigning layout functionality to the individual objects in the object structure. This layout functionality must be created with the `slaml-layout` primitive. Representing the object structure to the client is done by activating the assigned layout functionality on each object in the structure. Once the client submits the form - representing the object structure - the data are associated with the individual objects in the object structure. This association is done with the function `slaml-update-object!`. The individual object can be queried for the value of its data instance variable.

Design of the validation framework is divided into two levels, page level and object level. This makes validation available on both the complex forms framework and the session framework. On the page level a validation function is passed to the `slaml-show` primitive as the value of the `check` attribute. A validation function must take two parameters. The first is the form parameters entered by the client. The second is an additional parameter which can be supplied by passing the `checkparm` attribute to the `slaml-show` primitive. On the object level validation functions are written using the `slaml-check` function. They are associated with the individual objects in the object structure. A `slaml-check` function on the object level must always take one parameter, namely the string value entered by the client associated with the particular object. The return value must always be `#t` or `#f`. Validation is performed when `slaml-update-object!` is used to update the object structure.

This has resulted in three different frameworks solving the problems mentioned in Section 1.1.

3

Example Applications

Contents

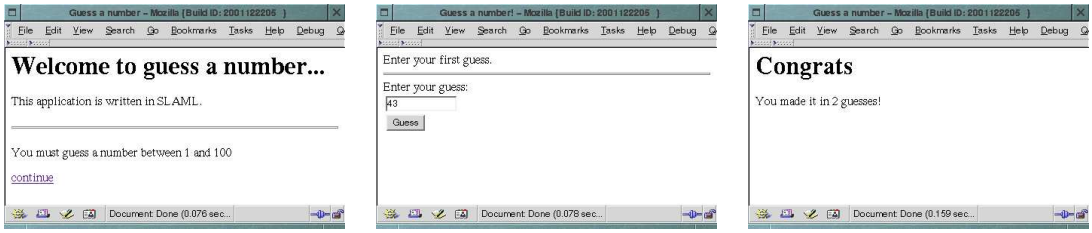
This chapter introduces two applications, which use the SLAML framework (designed and implemented to solve the problems identified during the analysis see Chapter 1). Reflections are made after the presentation of each application. The first application is a "Guess a number" application. The second example presented, is a "Student class" application.

3.1 Guess a Number Application

The first application implemented in order to show, how the SLAML framework can be used, is "Guess a number". The idea behind the application is that a client must guess a random number. When a client enters a guess that is invalid, a hint is shown to help the client. Since validation is done on the server, a client/server loop is maintained, until the client enters the correct number. The application is divided into two parts. The first part presents the way, that objects, layout, check functions and pages are defined and handled. The second presents the definition of a session, and the flow of the application is described. The code presented in this section is almost complete; the definition of two - almost static - pages and a single validation function is not included. The entire implementation of the application is found in Appendix B. For readability, all functions, objects and pages are defined globally.

3.1.1 Objects, Layout, Check Functions and Pages

The "Guess a number" application consists of three pages. First a welcome page, second the game loop page and third the end page. The three pages are presented in Figure 3.1.



(A) (B) (C)

Figure 3.1: The three pages in the "Guess a number" application. Page *(A)* is the welcome page. Page *(B)* is the page, where the client can enter a guess. The top of the page, presents a hint, or the text shown in *(B)* (if it is the first time the page is shown). The last page - *(C)* - presents the number of guesses used, when the correct number has been guessed.

The first and last page are simple and does not use objects to represent structure. The game loop page (*Page B*) consists of two parts. One part that gives the client a hint to help perform the next guess, and one part that contains the HTML `input` element and a `submit` button. The definition of the guess loop page, is seen in Figure 3.2.

```
(slaml-define-page guess-page
  (slaml-page (guess-information)
     (html
      (head (title "Guess a number!"))
      (body
       (get-hint guess-information)
       (hr)
       (slaml-do-layout guess-form)))))
```

Figure 3.2: The definition of the guess loop page. The page takes a single parameter - *guess-information*- which is used when getting a hint (the *get-hint* function). *guess-information* is a list, which contains the guess made by the client, the number of guesses used and the right number to guess.

This page builds up the two parts needed. The first part (the hint) is created with a call to *get-hint* and the second part (the `input` elements) is done by doing the layout of *guess-form*. *guess-form* is an object of the type `slaml-form-element`, which consists of a `slaml-element`, which again consists of two `slaml-basic-elements`. The creation of the object structure is seen in Figure 3.3. General for all the objects used, is that they are assigned a tagtype. The tagtype is used when the objects in the structure are updated with the data entered by the client (after a request). Similar, all objects are given a name. The

reason for giving the objects a name, is to identify them when the HTML layout functionality is specified. This is shown later.

```
(define guess-input
  (slaml-basic-element
   'check slaml-is-integer?
   'name "input-field"
   'tagtype "input-field-guess"))

(define submit-guess-button
  (slaml-basic-element
   'name "submit-button"
   'tagtype "submit-button-guess"
   'type "SUBMIT"
   'value "Guess"))

(define guess-composite
  (slaml-element
   'layout guess-composite-layout
   'elements (slaml-create-obj-lst guess-input submit-guess-button)
   'name "guess-composite"
   'tagtype "guess-composite"))

(define guess-form
  (slaml-form-element
   'layout guess-form-layout
   'name "guessform"
   'action ""
   'method "GET"
   'tagtype "guess-form"
   'elements (slaml-create-obj-lst guess-composite)))
```

Figure 3.3: The creation of the object structure. Notice, that the `slaml-basic-element` named *input-field* is assigned a check function named *slaml-is-integer?*. Recall, that functions, objects and pages are defined global for readability.

The first object created - *guess-input* - represents the HTML *input* element on the second page in the application (see Figure 3.1 *(B)*). The default value of the `type` property on the `slaml-basic-element` class is `TEXT`. Since the type of the HTML `input` element must be `TEXT`, the type is not specified when *guess-input* is instantiated, as `TEXT` is the default value. *guess-input* is assigned a check function named *slaml-is-integer?* (see Appendix B for its definition), which verifies if the value entered in the HTML `input` element is an integer. Next, the submit button is created. Like the *guess-input* object, the *submit-guess-button* is of the type `slaml-basic-element`. A composite object of the type `slaml-element` is created, and it is used to represent the two basic elements. A layout function - *guess-composite-layout* - is assigned to the object (the layout functions are shown later). The two basic elements are added to the *guess-composite* object. The elements must be gathered, by using the `slaml-create-obj-lst` function. The *guess-composite* object is added

to the root object, which is named *guess-form*. A layout function (*guess-form-layout*) is also added to the *guess-form* object. Even though the *guess-form* object is only added a single child (*guess-composite*), `slaml-create-obj-lst` is used. The reason for this, is that `slaml-create-obj-lst` adds a special tag to the list of elements it returns.

In order to present the object structure in an HTML form, it is needed to specify the layout of the individual objects in the structure. Layout functions have been created to the *guess-form* and the *guess-composite* object. The following presents the layout functions *guess-composite-layout* and *guess-form-layout*, which are the layout of the *guess-composite* and *guess-form* objects respectively:

```
(define guess-form-layout
  (slaml-layout (self parms)
     (slaml-do-layout-child self "guess-composite")))

(define guess-composite-layout
 (slaml-layout (self parm)
  (string-append
    "Enter your guess:"
    (table
     (tr (td (slaml-do-layout-child self "input-field")))
     (tr (td (slaml-do-layout-child self "submit-button")))))))
```

The purpose of the *guess-form-layout*, is to activate the HTML layout function on its child. As seen, the name of the child object (set in Figure 3.3) is used to specify which child to layout. The `guess-composite-layout` layout function, specifies that the `input` element and the `submit` button are placed in an HTML table.

The following section presents the second part of the "Guess a number" application. This includes the definition of the session used in the application.

3.1.2 Flow and the Session Definition

The flow of the application is modeled as a session containing three steps. One step handling the presentation of each of the pages defined. The definition of the *guess session* is seen in Figure 3.4.

```
(slaml-define-session guess-session
 (slaml-session (session-param)
  (slaml-show start-game) ;say hello - step one
  (letrec ((guess-loop
     (lambda (guess guesses right-number)
       (if (equal? guess right-number)
           guesses ; Return the number of guesses used
           (let
              ((obj-struct
                (slaml-update-object! guess-form
                  (slaml-show
                    guess-page 'pageparm (list guess guesses right-number))))
               )
              (if (slaml-get-valid guess-input)
                (guess-loop
                  (string->number
                    (slaml-get-data guess-input)) (+ 1 guesses) right-number)
                (guess-loop
                   NaN (+ 1 guesses) right-number)
                )
            )))))
    (let* ((right-number (get-random-number))
           (guesses (guess-loop 0 0 right-number)) ;do loop - step two
           )
      (slaml-show end-game 'pageparm guesses) ;say bye - step three
      guesses ; return the number of guesses used
      )
    );end letrec
  ))

(slaml-activate-session guess-session) ; it starts
```

Figure 3.4: The definition of the *guess-session*. The name *guess-session* is bound to a session that is created with the `slaml-session` primitive.

The *guess-session* is activated with the primitive `slaml-activate-session`, and as seen, the session takes a single parameter (it must always take exactly one parameter). The parameter to a session must be specified when the session is activated (with `slaml-activate-session`). This is not done in this example, so *session-param* is equal to the empty list. The first thing done in the session, is to call `slaml-show` with *start-game* as parameter. The page shown is the hello page (bound to *start-game*) and as seen, the page is given no parameters. Next, is the definition of the *guess-loop* function. The *guess-loop* takes three arguments: the guess, the number of guesses and the right number the client must guess. The first thing done in the loop, is to check if the client has made the right guess. If this is the case, the number of guesses is returned. If the client has not made the right guess, the *guess-form* object structure is updated with the information gained from showing the *guess-page* (already presented in Figure 3.2) and the object structure is stored in a variable named *obj-struct*. All information needed to present the page (create the hint to the client), are send as parameters to the page. The page can only take one parameter (specified with the `pageparm` attribute),

so the information is wrapped inside a list. None of the information used to maintain the loop is needed to be send to the client in hidden fields, but are instead handled as variables and parameters as shown.

A check function is assigned to the *guess-input* object, and after the object structure has been updated (by the `slaml-update-object!` function), it is asked if an object is valid according to its check function. This is done on the *guess-input* object, to see if the client entered a number. If this is the case, the game loop is called, with the entered number, the number of guesses increased by one and the right number. Else, the game loop is called with *NaN* - Not a Number - as a guess (*NaN* is bound to the value *-1*), the number of guesses increased by one and the right number, as arguments.

After the definition of the loop, a local variable named *right-number*, holding the random generated number (in the example it is generated by the function *get-random-number*), is created. Another variable - *guesses* - is set to represent the return value of a call to the *guess-loop*. The call to the loop means, that a number of interactions with the client is carried out. After the right number is guessed (the loop returns), the final page (*end-game*) is presented. The number of guesses used is send as a parameter to the page. The session returns the number of guesses used.

As shown in the example, information and functions needed to handle the flow of the session is maintained and defined locally to the session. This allows for encapsulation (in the form of lexical scope) of functionality and information. If needed, all the pages, objects and help functions (seen in Appendix B) can be created locally to the session (in a similar way as it is done with the *guess-loop*). The session can be loaded when the server is started (as it can be done with all libraries), and can be activated whenever wanted, by performing the call: `(slaml-activate-session guess-session)`.

In the example, a validation function is added to the *guess-input* object. It is somewhat comprehensive to first create a validation function, add it to the object and ask the object if it is valid, when only a single HTML `input` element exist on the HTML page. Since the only information from the *guess-page* is the value entered in the input element, page level validation can be used. This eliminates the need to query an object to determine if the input is a number. However, it is still our opinion that object level validation is beneficial when more than a single input element exist on the HTML page. The reason for this is, that a page level validation function gets complex (many `if` and `cond` statements), if it must validate many input values.

3.2 Student Class Example

To show that the SLAML framework is applicable in real world applications, a large example application (about 1000 lines of code) is developed. This application is described in this section. The application is not described in every detail but instead an overview is given of the system. The reason is that the software is large and giving a detailed description of

the whole application, is not necessary to understand how the SLAML framework is used in this application. The places where the SLAML framework is used, is described in details to give an indication of the usability of the SLAML framework in this application. The entire source code of the program is found in the folder *ExampleApplications/StudentClass* on the CD distributed with this report.

3.2.1 Overview of the Application

The application is a student class registration application. In this application it is possible for teachers to add new courses to a list of courses. Furthermore, students are able to select the courses they want to attend from the list of all courses. Each student have a profile, where details about the presentation of the pages for the student is set. Last, a calendar is available where a course can be scheduled.

Flow of the Application

The flow of the Student class application is seen in Figure 3.5.

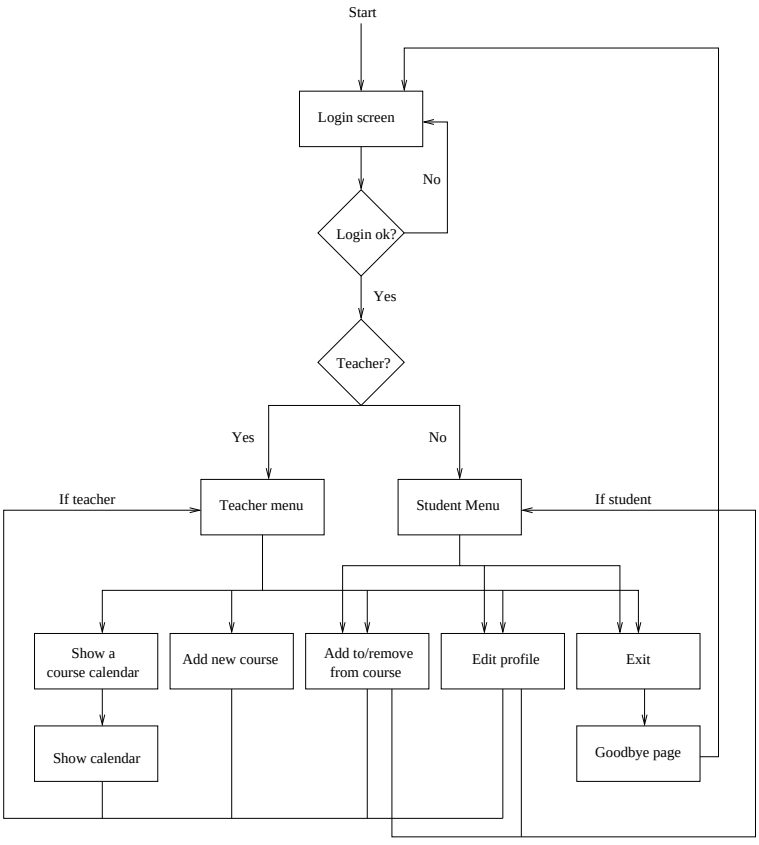


Figure 3.5: The flow of the Student class application.

The application starts with a login screen where the user is asked for username and password. If the username and password is incorrect, the login screen is shown again. If the username and password is correct, the type of user is examined. There exist two types of users in the application, students and teachers. The difference between these two types lays in what they are allowed to do. The next step in the application is to see if the user who is logged in, is a teacher. If this is the case, the teachers menu is shown, otherwise the students menu is shown. Teachers are shown a menu with five items:

- Edit profile
- Add to/remove from course
- Add new course
- Show a course calendar
- Exit

Students are shown a menu with three items:

- Edit profile
- Add to/remove from course
- Exit

As it is seen the two menus are the same, except that a teacher has two additional menu items (*Add new course* and *Show a course calendar*). Depending on which item is chosen in the menus, a new page is shown where an HTML form is present. In the following each of the menus are described.

The first menu item is *Edit profile*. Here the user can change his profile, this includes the background color, the welcome message and the title. The *Edit profile* page has three `input` elements in the HTML form, and the page is shown in Figure 3.6.

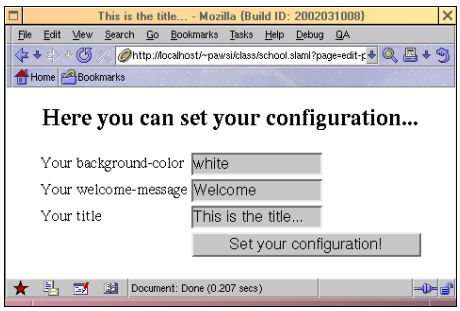


Figure 3.6: *Edit profile* page from Student class application.

The next menu item is *Add to/remove from course*. Here the user can specify which courses to attend. All the courses in the application are available and a checked check box indicates if the user wishes to attend a given course. The *Add to/remove from course* page is shown in Figure 3.7.

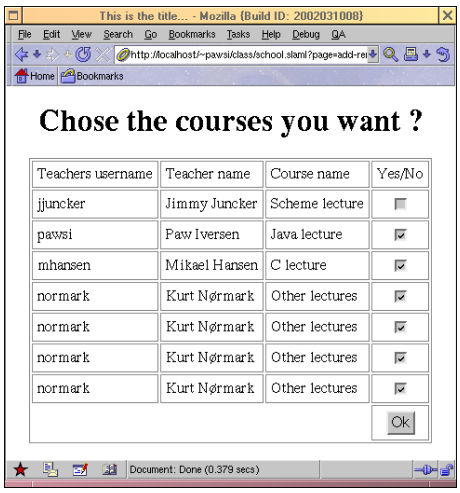


Figure 3.7: *Add to/remove from course* page from Student class application.

The third menu item is only available to teachers and is called *Add new course*. Here teachers can add a course to the list of already existing courses. The details that are needed to create a new course, is shown in an HTML form on this page. The teacher then enters information about the new course. When the submit button is pressed, the course is added to the list of courses. This page is shown in Figure 3.8.

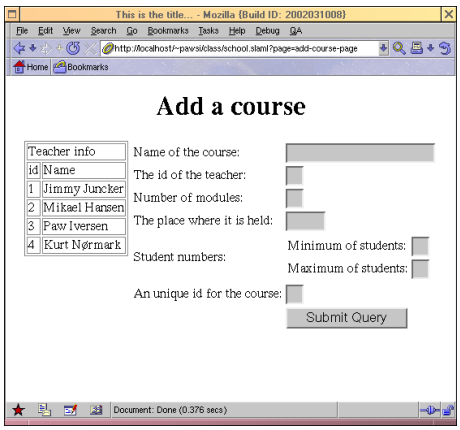


Figure 3.8: *Add new course* page from Student class application.

The fourth menu item is also available only to teachers and is called *Show a course calendar*. The only purpose for this calendar is as a demonstration of the use of complex HTML forms, where more than one item of the same type is shown on a page. Therefore this calendar do

not update the global state in the program, neither is the calendar associated to a specific course. This interaction consists of two pages. The first page shows a calendar consisting of check boxes. Here it is possible to mark days in a calendar. This page is shown in Figure 3.9.

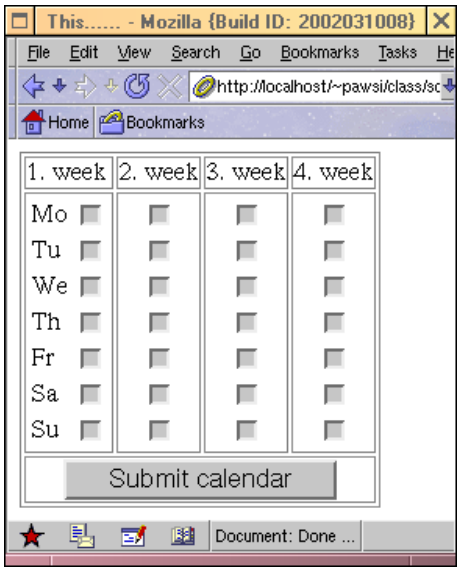


Figure 3.9: *Show course calendar* page from Student class application.

When the submit button on this page is pressed a new page is shown where the days - that was marked on the previous - is marked in a new calendar. This page is shown in Figure 3.10.

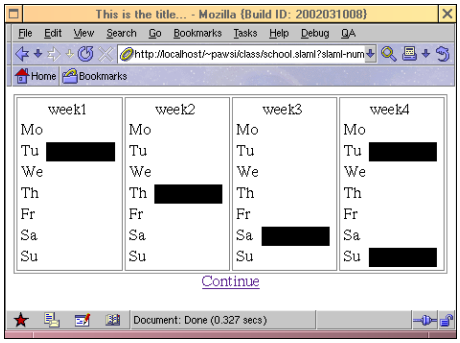


Figure 3.10: *Show calendar* page from Student class application.

3.2.2 Use of the Session Concept

In this section it is explained how the session concept is used in the Student Class application. This application contains five sessions:

- main-session

- login-session
- student-session
- teacher-session
- exit-session

The flow of the sessions is illustrated in Figure 3.11. *main-session* is the first session activated when the application is started. *main-session* is responsible for activating the *login-session*. *login-session* is the session asking for the username and password and returning a record with the data representing this user. *student-session* and *teacher-session* are almost identical. Depending on the type of user (teacher or student) one of the sessions is started once the user has logged in. The reason for having two sessions that are nearly identical, is to show that it is possible to have two sessions and on behalf of the type of the user, choose which session to activate. The last session is called *exit-session* and is activated when a user logs out. It shows a goodbye page and activates *main-session* again.

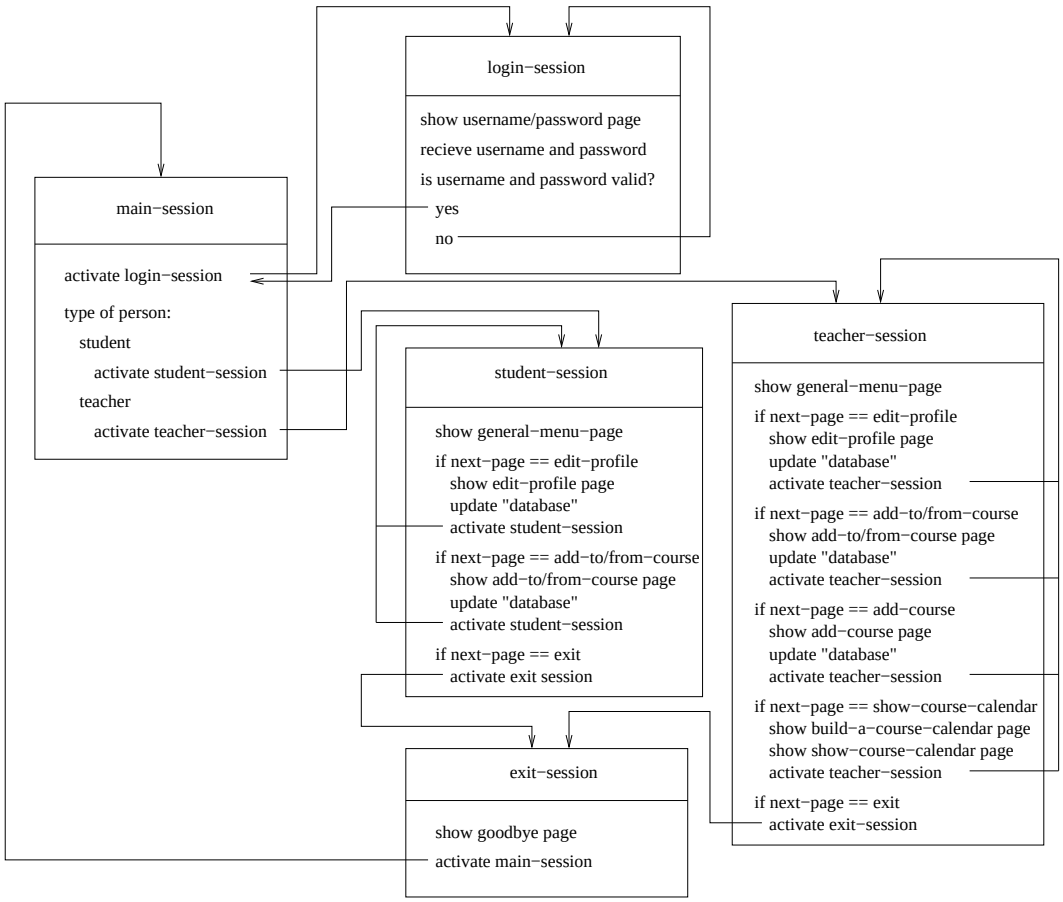


Figure 3.11: The session flow of the Student class application.

In the following *login-session* and *student-session* are explained.

Login-session

The first session that is discussed is the *login-session*. The *login-session* illustrates the usefulness of using recursive sessions. Furthermore it shows how a session can be used to return a value.

```
(slaml-define-session
 login-session
 (slaml-session (lst)
   (let ((app-user
          (slaml-show
           login-page
           'check login-check
           'checkparm people-lst)))                ; check if the person is valid
                                                   ; this is page level validation
     (if app-user                                  ; is the user valid ?
         app-user                                  ; yes : return the users information
         (slaml-activate-session login-session))))) ; no : reactivate the login-session
```

The *login-session* shows *login-page* to the client. If the user enters an non-valid username or password, the *login-page* return `#f` else it return the record structure - named *app-user* - for the person with the currently entered username and password. This is done with a check function, that is explained later. The last `if` expression in the *login-session*, checks if a valid username and password is entered. If it is a valid user, the users record is returned from the session. If it is not a valid user (the *login-page* returned `#f`) the *login-session* is activated again. This illustrated the usefulness of recursion of sessions.

Student-session

Other sessions that make use of recursion are *student-session* and *teacher-session*. Furthermore these sessions rely on session parameters. The session parameter is used to send a person record to the session and in this way make it possible to customize the layout of the pages as specified in the users profile. The *student-session* and the *teacher-session* are also used to show the right page based on the link in the *teacher-menu* or *student-menu*. Each of the links in the menu is created as follows:

```
(a "Edit profile" 'href "?page=edit-profile-page")
```

The *href* attribute sets the *page* url parameter to the value of the next page to display. As the *student-session* and *teacher-session* are alike, only the *student-session* is shown.

```
(slaml-define-session
 student-session ; the student-session
 (slaml-session (lst)
  (let* ((app-person (car lst))
         (next-page (slaml-formparms-key->value ; show general-menu-page
                     'page                      ; and get the "page" parameter
                     (slaml-show
                      general-menu-page
                      'pageparm (list app-person))))) ; show the menu page for the student
    (cond ; based on the page parameter submitted choose a page
     ((string=? "edit-profile-page" next-page) ; edit profile
        ;"show the edit-profile-page"
        ;"when the form is submitted, update the configuration"
        ;"activate the student session again"
      )

     ((string=? "add-remove-page" next-page) ; add to or remove from course page
        ;"show the add-remove-page"
        ;"when the form is submitted, update the course list"
        ;"activate the student session again"
      )
     ((string=? "exit" next-page) ; the exit session
        ;"activate the exit session")))))
```

Figure 3.12: The definition of *student-session*.

In the above example some of the code has been replaced with text. This is done to make it easier to read. This example shows the definition of the *student-session*. In this example next-page is bound to the value of the *page* form parameter (from the link in the menu), which is set by the link in the *general-menu-page*. Based on the *page* parameter the wanted action is performed (*edit profile*, *add to/remove from courses* or *exit*). After each action is performed, the *student-session* is activated again. This approach looks like the approach taken in CGI, but this is necessary to branch to the right action. This approach differs however from the CGI approach as here it is explicitly stated to restart the session. This is discussed in details in Section 4.3.

3.2.3 Use of Complex Forms

Complex HTML forms are used in four places in the application (*Edit profile*, *Add to/remove from course*, *Add new course* and *Show a course calendar*). In this section it is shown how complex HTML forms are used on *Add new course* page and *Show a course calendar* page. The reason for choosing these two is that these are the two most complex HTML forms in the application.

Add New Course

When adding a course to the list of courses, it is necessary to specify all the entries in a course record. A course record structure look as follows:

```
(course
 (name . "Scheme lecture")
 (teacher . "1")
 (modules . "5")
 (place . "E0-001")
 (student-info (min-students . "2")  (max-students . "10"))
 (id . "a1"))
```

The first entry is the *name* of the course. This is the string shown on the *Add to/remove from course* page. The next entry is the *teacher* entry, which specifies the id of the teacher that will be teaching the course. *modules* specifies how many modules (lessons) the course consists of. *place*, is the name of the place where the course is held. *student-info* is a new record structure, specifying what the minimum and the maximum number of students are for this course. Last is the *id* of the course, this is a unique id used to relate courses to students.

To present a form where these informations can be entered, a complex form is build. The reason for building a form to handle the new course is that the course record structure is build to match the list returned from the `slaml-update-object!` function. This makes it easy to mutate the global list - where all courses are present - to include the new course. When the list is returned from `slaml-update-object!` it is made to fit the course record shown above and it is then added to the global list of courses. This makes it easy to add new courses to the list of courses.

Show a Course Calendar

Another place where a complex HTML form is used, is in the *Show a course calendar* page. This HTML form consists of four weeks where each week consists of seven days. To generate a week object with seven days a function is used.

```
(define (create-course-calendar-days-objects) ; creates a week
  (slaml-create-obj-lst
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "Mo" 'type "CHECKBOX")
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "Tu" 'type "CHECKBOX")
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "We" 'type "CHECKBOX")
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "Th" 'type "CHECKBOX")
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "Fr" 'type "CHECKBOX")
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "Sa" 'type "CHECKBOX")
```

```
   (slaml-basic-element
    'value "present" 'tagtype "day" 'name "Su" 'type "CHECKBOX")))
```

This function takes no parameters, but returns an *object-list* (which is a tagged list used inside the objects to represent references to other objects) of seven `slaml-basic-elements` representing a week. The reason for using a function instead of binding the list to a name, is that a new instances of the week is needed. As it is seen each day is tagged with `day` and the name of the objects are the names of the days of a week. The reason for tagging the list with *day* is that thereby it is possible to specify that each of the lists returned from `slaml-update-object!` represents a day. The list representing one week is then added to each of the four weeks in the calendar HTML form.

```
(define create-course-calendar-weeks-object ; a month of objects
  (slaml-create-obj-lst
   (slaml-element
    'name "week1"
    'tagtype "week"
    'elements (create-course-calendar-days-objects)
    'layout week-layout
    )
   (slaml-element
    'name "week2"
    'tagtype "week"
    'elements (create-course-calendar-days-objects)
    'layout week-layout
    )
   (slaml-element
    'name "week3"
    'tagtype "week"
    'elements (create-course-calendar-days-objects)
    'layout week-layout
    )
   (slaml-element
    'name "week4"
    'tagtype "week"
    'elements (create-course-calendar-days-objects)
    'layout week-layout)))
```

Here the weeks are tagged with *week* and the name of the objects are the name of the weeks (here *week1*, *week2*, *week3* and *week4*). To get the elements (days) for each week, the *create-course-calendar-days-objects* function is called. The list representing four weeks is bound to the name *create-course-calendar-weeks-object* as only four weeks are needed in the application. The list representing four weeks can now be added as elements to the calendar HTML form.

```
(define course-calendar-form ; calendar form
  (slaml-form-element
   'action ""
   'elements (append
              create-course-calendar-weeks-object
```

```
              (list (slaml-basic-element
                     'name "submit-button"
                     'tagtype "submit-button"
                     'type "SUBMIT"
                     'value "Submit calendar")))
   'layout month-layout))
```

Here it is seen that the new `slaml-form-element` is bound to the name *course-calendar-form*. Furthermore a submit button is added to the HTML form. The *course-calendar-form* is then shown to the client which checks the check boxes that represents the days where a course is held. When the HTML form is submitted, the data returned from the HTML form is used to update the *course-calendar-form* object structure. How this is done is illustrated in the following.

```
(let ((page-data
       (slaml-show
        show-course-calendar-page
        'pageparm (list app-person))))
  (slaml-show
   show-course-calendar-result-page
   'pageparm (list app-person page-data))
  (slaml-activate-session teacher-session 'sessionparm lst))
```

First is the *show-course-calendar-page* shown to the client. The resulting form parameter list is then bound to the name *page-data*. The record representing the current user is bound to the name *app-person*. The reason for sending this record to all pages is that it contains the profile of the person. This information is used to e.g. set the background color of the page. Next step in the application is to show the page bound to the name *show-course-calendar-result-page*. This page is given a list containing *app-person* and *page-data* as parameter, as this data is used to build the resulting calendar. Last the *teacher-session* is activated again. The *lst* parameter, which is passed as parameter to the *teacher-session* is passed as parameter to the session from which the above example is taken.

3.2.4 Use of Validation

Both page level validation and object level validation is used in the application. Page level validation is used in the *login-session* when the *login-page* is called. Object level validation is used to check the background color on the *Edit profile* page. Both of these are explained in this section.

Login Check

When the *login-page* is shown, a check function is used to check the username and password against a list of persons (which each has a username and a password).

```
(define login-check ; function used to do page level check on the login form
  (lambda (form-parms all-persons-lst)
    (let ((username (slaml-formparms-key->value 'username form-parms))
          (password (slaml-formparms-key->value 'password form-parms)))
      (check-user-and-password username password (get-persons all-persons-lst)))))
```

In the above the check function used with the *login-page* is seen. It is seen that the *form-parms* parameter is asked for the username and password. This is done with the function `slaml-formparms-key->value`. The username and the password is bound to the variables called *username* and *password*, respectively. These two values are passed to the function *check-user-and-password*, which takes a username, a password and a list of persons as parameter. Based on these parameters the person that matches the username and password is returned. If no person matches the username and password, `#f` if returned. The second parameter to the *login-check* function is the list of all persons in the system. *login-check* returns what the *check-user-and-password* returns.

Background Color Check

An example of object level validation is in the *Edit profile* menu, where the submitted background color is checked. The reason for checking the background color is that if the user submits black as the background color the user cannot see the text on the screen as this is black too. The following check function is used for checking the background color:

```
(define (check-background-color str)
  (not (string=? str "black")))
```

This check function is simple, but it is useful since it makes it impossible to select the same background color as the text color.

Another possibility to this problem is to add a menu to the page where the colors can be selected from. Thereby, it is unnecessary to do check on this value as it is impossible to choose a wrong color.

Check on coherence of two `input` elements is useful in the *student-info* record structure in the *Add new course* page. Here it must always be the case that the minimum number of students is lower than the maximum number of students. But as validation on objects of the type `slaml-elements` is not implemented it is not used in this application.

3.3 Summary

Two applications based on the SLAML framework is discussed in this chapter. First a small application - “Guess a number” - is discussed. Second, a larger application - “Student class” - is discussed. Based on the experiences gained during the implementation of these example

applications, the following concludes on usability of the frameworks.

The session framework in SLAML gives the developer the possibility to think of a Web application as one program. The result is that the flow of the Web application is like the flow of a non-Web application. This means that the flow of the program is gathered in a single file. This gives an overview of the flow of the application. That a session encapsulates interactions with a client means that responsibility can be delegated on a higher level than a single page. By this we mean that interactions sharing the same responsibility can be gathered in a session. An example is a login session where more than one page is responsible for ensuring a user is logged in. By doing this it is possible to access all the pages responsible for logging the user in, as a unit.

The complex forms framework, gives the developer possibility to create an object structure on the server and rely on this object structure to query for data returned from the client. Therefore, the object structure send to the client is also the object structure that is queried for data. Building the object structure and placing the layout on the objects in the structure is a considerable amount of work, but once this has been done it is straight forward to update the object structure and query the objects in the structure for data.

The validation framework is designed to work on both the page level and the object level. This means that it is possible to do validation when using the complex forms framework as well as the session framework. The validation framework is not implemented on `slaml-element` and `slaml-form-element`. On the page level the form parameters from the client is send as parameter to the check function. Thereby, all data from the submitted HTML form can be checked. This is an advantage since it thereby is possible to build HTML forms without the complex forms framework and still get the data validated. However, if both the complex forms framework and the session framework is used it is redundant to check on both the page level and the object level.

4

Reflection

Contents

In this chapter the reflection of the designed and implemented frameworks are given. This chapter consists of four sections.

The first section gives an overview of the problems that were encountered during the implementation of the SLAML framework. These problems are related to the Apache server.

The second section is about the limitations to the implemented framework in relation to the design. The limitations are concerned with the implementation done in Scheme as well as limitations by the problems encountered when implementing the session framework in Apache.

The third section is reflections on the experiences gained when implementing the example applications (from Chapter 3) with help from the SLAML framework.

Last is a summary where the limitations and reflections are summarized.

4.1 Encountered Problems

During the implementation problems were encountered. These are described in the following two sections. The nature of the encountered problems is mainly on a low level i.e. involving mod_laml and the server. The reason for not solving the problems is the decision to place our focus on Scheme level implementation.c

4.1.1 New Apache Module

During the implementation of the designed solutions a problem concerning mod_laml occurred. The reason for this problem is the intended implementation of the `slaml-show` primitive. We decided to let the `slaml-show` primitive halt the evaluation of the SLAML application and display a page to the client. Upon submission of the page from the client, the program control is returned to the `slaml-show` primitive in the SLAML application.

The solution is based on a signal/wait situation in the Apache server. By a signal/wait situation we mean that once the first request - for a Web application - is handled by the server, the server will spawn a new thread (named *session thread*) to run the Scheme program in. This means that two threads are present in the server after the first request (the Apache process that handles the request is also seen as a thread, the *main thread*). This is seen in Figure 4.1.

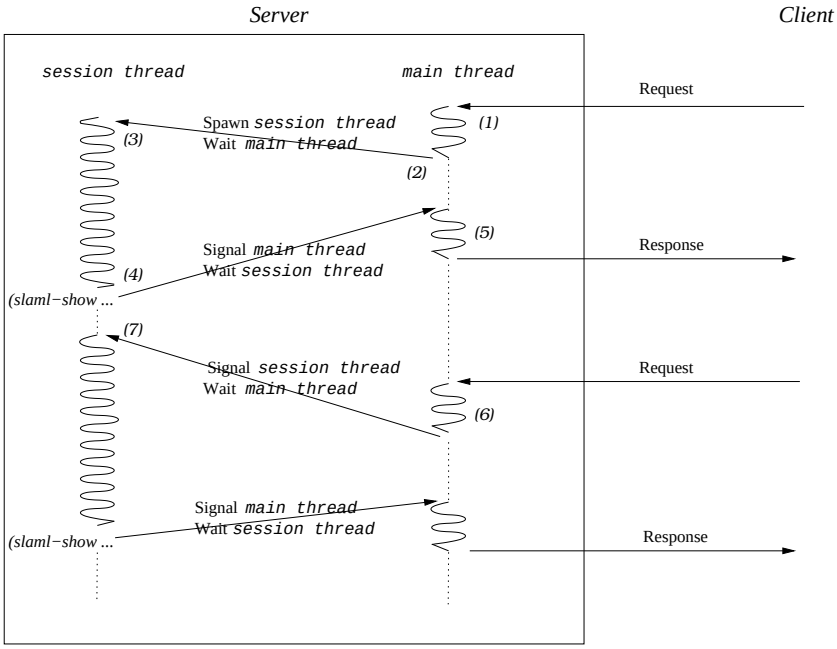


Figure 4.1: An illustration of the two threads running in the server and the communication between them.

To handle requests by using threads, the server module must handle seven steps, which are explained below. Each number in the following, corresponds to the same number in the figure:

1. When a client requests a SLAML application, it is handled by the *main thread*. The *main thread* represents the Apache child that receives the request.

2. The *main thread* checks if the *session thread* is present (checked by a global state in the module). Since this is not the case at the first request, the *session thread* is spawned and the *main thread* waits.

3. The *session thread* starts evaluating the Scheme program requested by the client, by using the embedded interpreter.

4. When the *session thread* reaches a `slaml-show` in the application, it signals the *main thread* and waits.

5. The *main thread* sends the page - specified as a parameter to `slaml-show` - to the client and ends the request. This results in the *main thread* going idle.

6. When the client submits the HTML form from the page presented, the *main thread* is resumed. Like step 2, the *main thread* checks if the *session thread* is present (by checking the global state in the module). Since this is not the first request to the application, the *session thread* is already present. Therefore it is signaled (instead of spawned) and the *main thread* waits.

7. When receiving the signal, the *session thread* resumes its computation. This results in `slaml-show` to return the data entered by the client. The steps from step 4 and forward are continued until the application is ended.

The problem related to this approach is that mod_laml is unable to use threads as it uses the Apache server version 1.3 [apa01]. We tested the pthread library with mod_laml, but were unable to create threads in mod_laml. Additional small modules indicated that threads are not allowed in a module for the Apache server version 1.3. Version 2.0 of the Apache server is created to allow better portability and better support for threads, as the process model in this version is changed. This motivated us to implement a module using threads under Apache 2.0, and it worked. Therefore we replaced Apache version 1.3 with Apache version 2.0 and created mod_laml for this version of the server.

By using Apache version 2.0 it is possible to use threads in a server module. But it is not possible to use Apache version 2.0 together with MzScheme. The reason for this is unknown, but small tests indicates that it is impossible to start a MzScheme interpreter inside Apache 2.0. Therefore we changed the Scheme interpreter to Guile[gui01]. This resulted in creation of a module containing only the necessary features to perform a proof of concept implementation of the designed solution to the problems presented during the analysis.

4.1.2 Handling Data on the Server

During the beginning of this semester experiments with handling data sharing on the server were conducted. This was motivated by two requirements.

First a way to share data between the individual server processes are needed. The reason is that Apache starts several processes to handle requests from the client. The processes cannot share data, so if a client sends a request to one process this process must also serve the following requests for the data to be accessible. This is not the case in Apache, since an arbitrary process can respond to a request. But if the processes can share data it is unnecessary to ensure that a given process always handles requests from a given client. Secondly it

is needed to make sessions persistent, so in case the server restarts it can access the data related to the sessions. This is impossible if data related to the session is kept in main memory. Another reason for considering persistence of sessions is the memory usage on the server. If all sessions are kept in main memory and not moved to disk, the memory requirement will increase continuously. These two problems are discussed and possible solutions are given.

We found a library that is used by other Apache modules (e.g. mod_ssl [Eng02b]) which solves the problem of sharing data between processes in an Unix system. The library is called MM [Eng02a] and is a flexible way to create memory that can be shared between the various server processes. This is done by creating a memory segment and give each server process a reference to it. This memory segment is - dependent of the platform - located in a file on the hard-disk or in main memory, so using this memory segment is not always as efficient as main memory.

The problem with persistence of sessions is to store the contents of a Scheme environment. The contents of the embedded Scheme environment is all the name bindings in the current instance of the interpreter. In general the content is all the information needed to reactivate the Scheme interpreter, as if it has never been deactivated. Making sessions persistent has been done by Queinnec in [Que]. Queinnec has implemented a server and a Scheme interpreter in Java, which allows continuations to be stored on the disk. By implementing both the server and the Scheme interpreter in Java, Queinnec relies on the possibility for serialization in Java [Inc02a] to store continuations. Serialization allows writing object to files on disk, and to recreate the objects from the files. Unfortunately this is not possible when relying on Apache and Guile as these are written in C and serialization is not as easy in C as it is in Java.

The implementational task of solving data sharing requires a substantial amount of work, and will place focus of the project on the C level instead of the Scheme level. As focus is on the Scheme level in this project, it is chosen to rely on a server module where data is kept in main memory and not written to disk. Therefore it is not considered any further. This results in a number of limitations which are stated in the following section.

4.2 Current Limitations

In the implemented framework there are limitations in relation to the designed framework. Some of these limitations are related to the implementation of mod_laml. Others are limitations made during work at the Scheme level. Limitations in mod_laml are related to implementation of the `slaml-show`, sharing data between processes and making Scheme environments persistent. Limitations in the SLAML framework are made when no new insight is expected by implementing the facility.

4.2.1 Mod_laml Limitations

Due to the problems described in Section 4.1 a new implementation of mod_laml is made. It is decided that a basic implementation is sufficient for the purpose of this project. The module need to be able to evaluate Scheme programs - with `slaml-show` primitives - on the server. Since this cannot be done with the old mod_laml the solution was to discarding the module implemented during the preparatory work, and create a new module for the Apache server version 2.0, using the Guile Scheme interpreter. In the following it is presented what is lost and what is still present from the old mod_laml when using the new mod_laml.

In the old module it is possible to load files into mod_laml when the server is started and thereby decrease the execution time spend on the request. This is still possible in the new mod_laml, but in the new version a Scheme file is used for loading the libraries, whereas the old version relies on a special configuration file.

In the old version of mod_laml it is possible to have as many client connected as the hardware allowed. The reason is that the Scheme environment is cleaned after each request. This is not possible in the new version of mod_laml as information about the sessions are needed at a later time. Furthermore, since it is not possible to share data between Apache processes, each client needs its own Apache process. It is impossible to ensure that a client gets the same Apache process at the next process so only one Apache process can be present at the server. This is done because it must be ensured the client gets the same Apache process at the next request. As the Scheme environment in the Apache process is dedicated to one client, only one client can access the server.

The old version of mod_laml has a possibility to use different interpreters. This is not possible in the new version of mod_laml. The reason is that the implementation of this requires much work on the implementation of Scheme in Apache, and this is not the focus of this project. But implementing support for more than one interpreter is designed in our Dat5 report and can be implemented according to this design.

In order to make the new version of mod_laml work with more than one client it is necessary to make data sharing and data persistence available on the server. This is the first thing to be done in order make this module usable as a production server. By production server we mean a server that is stable enough to be used for other than proof of concept applications.

4.2.2 HTML Elements

In the design of the complex forms framework the `slaml-basic-element` type is created to represent basic HTML elements, such as the HTML `input` element. In the implementation the possible HTML elements are limited to the HTML `input` element. This is chosen as the implementation of the additional HTML elements does not bring new insight. To implement the additional HTML elements a new property on objects of the `slaml-basic-element` type indicating the element type of the element, must be added. Possible values for this are:

`input`

`select`

`textarea`

`button`

Besides introducing the element type property, the functionality of the `slaml-basic-element` function must be extended to include possible attributes and default values of the properties from the new HTML elements. For a complete reference of the elements and the attributes of the elements please refer to [W3C02b].

The above list contains only a subset of the HTML elements that can appear as part of an HTML form. The above list is chosen as these are the ones that allows input to be entered by the client. As an example the HTML `label` element is also a valid HTML element inside an HTML form. But it must not be represented as a `slaml-basic-element`, as it cannot represent data entered by the client. Instead the HTML `label` element is part of the layout functions written on objects of the `slaml-form-element` and `slaml-element` types.

We suggest to implement the new elements as a property on the `slaml-basic-element` class, indicating the type of the HTML element. An example of a property name can be *element-type*. As part of this, functionality on the `slaml-basic-element` must be extended to support the attributes present with the new element type.

4.2.3 Error Messages

The initial goal of this project is to help the developer in the development process. Part of this help, is to provide suitable and precise error messages when an error is encountered. The task of writing descriptive error messages has not been a design goal for the SLAML framework. Writing error messages is something that must be done, before software is released. No special considerations to the quality of the error messages in the current implementation has been given. This is chosen since the implementation is not intended as a product for release, but rather as a proof of concept.

4.2.4 Validation on `slaml-element` and `slaml-form-element`

A validation framework has been developed for both the session framework and the complex forms framework, but the implementation of the validation is not complete in the complex forms framework. Validation on objects of the `slaml-element` and `slaml-form-element` type is not implemented, as it is not necessary to show that validation on the individual objects is possible. Furthermore, by proving the idea of a validation function on the objects of the `slaml-basic-element` type we expect this to work on `slaml-element` and `slaml-form-element` too. Validation on objects of the `slaml-element` and `slaml-form-element` type must be implemented for the validation in the complex forms framework to be complete.

4.3 SLAML Framework

In this section the experiences with the SLAML framework are considered. The session framework, complex forms framework and validation framework are reflected upon in turn. The reflections are based on the experiences gained during the implementation of the "Guess a number" example and the "Student class" example.

4.3.1 Session Framework

By using the session framework we are able to share data between interactions with a client. This is seen as a strength compared to a CGI approach, where data sharing is usually done by sending data between client and server. In the session framework, data is present on the server when the next request is received. This ensures a decreased use of bandwidth, since client input are not send between server and client in order to be available at a later point. However in CGI there is also a possibility to store data on the server, but this has to be done explicitly by the programmer. Furthermore, CGI applications uses files or databases for sharing data since data in main memory are lost after a request. Using files to share data means that the state of the program is accessible from all other scripts in the CGI application. In the session framework the data received from an interaction is part of a lexical scope and thereby protected from other unrelated interactions. Furthermore, the data is kept in main memory, which means that the programmer does not need to explicitly take care of storing the client data to disk.

The session framework introduces a new way to view a Web application. As the `slaml-show` primitive is used to ask the client for data, the view of an application is turned around. By this we mean, that in a CGI approach the Web application is mostly programmed in a way where the client asks the server to generate a page based on input. In the session framework the server asks the client for data and based on this data, it can continue the evaluation of the application. By requesting data from the server, the client still has the control of the application. However, the opposite seems the case from the developers point of view, since it seems as if the client is asked for data.

From the "Student class" example it is experienced that the flow of the application is more like non-Web applications (compared to a CGI approach). The reason is, that the next action to be performed when the client submits data, can be seen in the program. An example is the menu in the "Student class" example. In the example the menu page is presented to the client, and based on the link chosen in the menu a branch of the program is taken. At first sight this looks like the control needed in the CGI approach, where a `cond` special form is used to determine the page to show. The difference between the two approaches is, that it is not possible to maintain an overview of various interactions with the client in a CGI script. This is possible in the session framework since the `slaml-show` returns like a normal function and each `slaml-show` represents an interaction with the client.

In Bigwig it is possible to activate a session from the URL. This means that a menu like

the one in the "Student class" example, can be constructed of links to new sessions. But the "Student class" example shows, that when activating a link only one or two pages are shown in sequence. To make sessions that only consists of one or two pages spreads the flow of the application. The reason for this is, that it results in a similar approach as CGI, where a single page in an application corresponds to a single script. In the session approach, each page then corresponds to a session, which is activated by a link. This approach seems much like the approach used in WASH/CGI and PACKS/HTML, where event handlers are associated to submit buttons on an HTML page. Pressing a submit button results in the presentation of a single page from a Web application. Since an event handler is equal to a page, which is equal to a single script, the entire application is split into a number of scripts.

4.3.2 Complex Forms Framework

In this section the complex forms framework is compared to a solution based on CGI.

One way to come around the *Complex forms* problem in CGI, is to use data structures in the CGI program. Based on these data structures, functions that can take a data structure as input and return the HTML representation of the structure, must be created by the developer. In this way each structure needs an associated function to generate the HTML representation. When data are submitted from the client, a function is needed to recreate the structure based on a specific key/value pairs string. Therefore it is necessary to send information to the client specifying what type of structure is submitted (see Section 1.1.3). This approach means that two functions are needed for each structure. The first is responsible for presenting the structure and the second is responsible for rebuilding the structure.

We can think of two optimizations to this approach. The first is to make a general function that can take any given structure and create the HTML presentation. Since the function must contain information about the presentation of each individual structure, it is difficult - if not impossible - to achieve such general functionality.

Another solution is to build substructures and let two functions handle presentation and recreation of each substructure. Large complex structures can then be created based on these substructures. This can be done by combining functions that generate HTML presentation of substructures. By aggregating HTML presentation of substructures, the HTML presentation of a larger structure is achieved. Recreating a large and complex structure is done by combining the appropriate recreation functions used on substructures.

In contrast to the CGI solution, the complex forms framework consists of three phases. The first is to build the object structure. This is similar to building a structure in CGI, except when programming CGI in Scheme a list structure is more appropriate than the object oriented approach used in the framework. The reason for this is, that in CGI it is comprehensive to rebuild an object structure after each request, whereas an object structure created in the framework is persistent (in main memory) and survives interactions with the client. The next step is to present the structure. In the complex forms framework this is done by adding

HTML layout to the individual objects and call `slaml-do-layout` to generate the HTML representation. In CGI this is done by writing functionality, that based on a specific structure generates the HTML representation. The last step is to update the object structure. In the complex forms framework this is done by calling `slaml-update-object!` with the received data and the object structure to update. In CGI, this is done by creating a function that creates a complex structure on behalf of the substructures.

Comparing the CGI approach with the complex forms framework two large differences are seen. The first is, that in the CGI approach a new function has to be created for each structure to present. Since the function that presents a structure also contains the HTML layout, a new function has to be created for each representation of a structure. In the complex forms framework the layout of the object structure can be changed since it is a property on the object. Furthermore there are general functionality to generate the layout of an object structure (`slaml-do-layout`). The second difference when comparing the complex forms framework with a CGI approach is recreation of a data structure. In CGI, this is done by writing a recreation function for each structure. To recreate a structure, the server needs to know which structure is submitted. This is needed in order to call the right function that based on data from the client recreates the structure. In the complex forms framework all this is done by passing the object structure - present on the server - and the data received from the client to `slaml-update-object!`. This results in the object structure being updated with the values received from the client. However, if two HTML forms are present on the same page, information about each structure must also be present. The reason for this is, that both structures send to the client are present on the server. The problem here is, that the server does not know which of the two HTML forms is submitted. This information can be placed on the submit button in the HTML form and based on which submit button is pressed, the appropriate structure can be send as parameter to `slaml-update-object!`.

4.3.3 Validation Framework

In this section the experiences with the validation framework are reflected upon. The validation framework works on both the object level and the page level. Both of these levels are reflected upon in the following.

On the object level, validation on `slaml-basic-element` is implemented. This means that it is possible to add a check function to objects of the `slaml-basic-element` type and after the object structure is updated the objects can be queried for their status (valid or invalid). This gives the developer the possibility to validate the data from the `input` elements in an HTML form.

Validation on composite objects has also been designed, but not implemented. Our opinion is that validation on composite elements is useful since it gives the developer a way to validate the dependencies between elements in the structure. An example is a page where two lists of `input` elements are presented. One consists of person names and the other consists of person emails. In this example it must be ensured, that all persons have an email, i.e.

when data is successful validated, there are an equal amount of names and emails.

Validation on the page level gives the developer possibility to verify that the HTML form submitted from a page is valid according to a check function. This can also be done in CGI, since the form parameters from the client can be passed to a validation function defined in the script. The difference between the CGI approach and the approach in the validation framework is, that validation of the client data is done as one action in the validation framework. This is possible since the server has knowledge of which function to use for validating the data returned from the client (it is specified when `slaml-show` is called).

The approach used in the validation framework, changes the semantics of the `slaml-show`, since the returned value from this function indicates if the data received from the client is valid or not. Thereby `slaml-show` has a semantic that states: If data is valid according to the check function return the return value of the check function else return false.

4.4 Summary

This chapter presents the problems encountered during the implementation of the design. A number of limitations has been presented and described. These are the mod_laml module only being able to handle one client. The `slaml-basic-element` only representing the HTML `input` element. The error messages, not being considered. And finally the lack of validation on objects of the types `slaml-form-element` and `slaml-element`. Based on the implementation of the two proof of concept applications - described in Chapter 3 - reflections on the designed solutions to the problems are given. The reflections also includes comparison with the CGI approach.

5

Conclusion

The purpose of this project was to continue work made during the preparatory project (Dat5). In the previous project, mod_laml was developed, which decreased the evaluation time of the average LAML script by 45%. Furthermore, existing work in the Web world was analyzed in order to identify ideas and principles that could be used together with mod_laml. The Dat5 project concluded, that we wanted to make Web development in SLAML (Server side LAML in mod_laml) "easier". Therefore, the first task in this project, was to specify how to do Web development "easier". To make Web development "easier" we needed to identify often encountered problems in Web development. Therefore the focus of this project has been to identify problems in Web development and to design and implement solutions to these problems. Four problems were identified (see Section 1.1):

1. State handling
2. Input validation
3. Complex forms
4. Reusability

The *State handling* problem had two different aspects, namely *Control flow handling* and *Data flow handling*. During the analysis, the four problems were explained in detail, and possible solutions to the problems were presented. After the specification of the problems, the *session concept* was introduced as a possible solution. Three possible ways of using sessions were identified, and they were presented through an analysis of Bigwig, WASH/CGI and PACKS/HTML. The Analysis ended in a problem definition, which presented three hypotheses. In the following, the results of this project is related to each of these hypotheses.

The first hypothesis is related to two of the identified problems in Web development, namely *State handling* and *Reusability*. The first hypothesis is presented below:

Hypothesis 1:

> ***A session-centered approach to Web development in SLAML solves the State handling problem of a Web application. Furthermore, a session concept makes access to several HTML pages as a single unit possible.***

The first hypothesis was split into sub-hypotheses as it includes two problems, namely the *State handling* and *Reusability* problems. Recall, that the *State handling* problem had two aspects: *Control flow handling* and *Data flow handling*. This resulted in three hypotheses. The first of the three sub-hypotheses regards the *Control flow handling* problem and is presented below:

Hypothesis 1.1:

> ***The Control flow handling problem is solved by introducing a session concept, where a primitive in the language displays an HTML page to a client and returns as a regular function.***

In this hypothesis we state that the *Control flow* problem is solved by introducing a primitive in the language that can show a page to the client and return control to the program as a regular function call. This primitive was designed and implemented and is called `slaml-show`. This solution is inspired by Bigwig and the primitive in Bigwig called *show*. In relation to CGI, this primitive solves the problem that the developer has to take care of the control flow explicitly (by linking between files or use selection statements) as described in Section 1.1.1. This helps the developer to see a Web application as one application rather than small "applications" linked together.

In relation to the *Control flow* problem, The `slaml-show` primitive is designed and implemented like *show* is in Bigwig. Therefore `slaml-show` will have much of the same effect on SLAML programs as *show* has on Bigwig programs. This results in simplifying the interaction with a client. This was much as expected since Bigwig states that:

> *"...the session concept greatly simplifies the programming of complicated control flow with multiple client interactions ."* [CAM02]

In our opinion we have solved the *Control flow* problem, since `slaml-show` works as a regular function call in a non-Web application. `slaml-show` is called when data is needed from the client and it returns the form parameters to the surrounding program.

Hypothesis 1.2:

> ***The Data flow handling problem is solved by introducing a session concept to SLAML, where interactions inside the same lexical scope (session) can share data.***

The *Data flow handling* problem is solved along with the solution to the *Control flow* problem. This is so, since the `slaml-show` primitive ensures that sequential interactions with a client are performed without the Scheme environment on the server is lost after each interaction. `slaml-show` was designed and implemented to return the data received from a

client, and this data can be stored in variables in the Scheme environment. This means that the data are present at a later time, and can therefore be used without the developer having to handle data explicitly (e.g. store it in hidden `input` elements or on the servers filesystem).

The introduction of the `slaml-show`, resulted in the developer being able to see a Web application as one program and thereby as a non-Web application. This has the effect that all data received in an application can be bound in the Scheme environment and be available at a later time. This was expected as this is the case in Bigwig. Since it is possible to program a SLAML application as a non-Web application, data is present once it is bound in the Scheme environment.

We mean that the *Data flow handling* problem is solved, by introducing the `slaml-session` primitive. The reason is that this primitive encapsulates interactions with a client and allows the interactions to share data.

Hypothesis 1.3:

> ***The Reusability problem is solved by introducing a session primitive that can activate a series of interactions with a client and rely on parameters at call time.***

Reusability has been obtained by implementing a session primitive that relies on parameters. A `slaml-session` in the SLAML framework, is a first class object in the Scheme environment. This fits well with the Scheme language. The `slaml-session` primitive encapsulates a number of interactions with the client, represented by `slaml-show`s. In order to fully evaluate the level of reusability gained by introducing sessions, a number of general sessions must be created and evaluated in accordance to the reusability.

We have implemented relatively few applications with the SLAML framework and can therefore not conclude if the *Reusability* problem is solved. However, a `slaml-session` encapsulates more than one interaction with a client and can therefore be used as a module that can activate interactions with a client. Since a session can take arguments, it is possible to activate a session in different contexts.

The second hypothesis is related to the *Complex forms* problem. It includes the three steps present in the problem:

Hypothesis 2:

> ***It is possible to construct a framework that helps the developer to build, present and update complex structures.***

The *Complex forms* problem was inspired by the need to:

- Build complex data structures on the server.
- Send the data structures to the client as an HTML form and get it filled with data.

- Receive the data and maintain the data structures.

This is not possible in plain CGI as there is no solution to build an HTML form from a structure and receive the data in the same structure as it was presented to the client. No solutions were found that solves this problem. Motivated by this we designed and implemented a framework as part of the SLAML framework, to handle complex forms.

It was chosen to rely on objects to represent data structures. Other alternatives were presented. These were a embedded domain specific language and a nested list approach. The reason for choosing an object oriented approach is that it gives a flexibility to easily change the structure.

Since the session framework is part of the SLAML framework, it was possible to rely on features from the session framework when the complex forms framework was designed and implemented. The reason for relying on the session framework in the design and implementation of the complex forms framework is that sessions makes it possible to store the object structure on the server and update it with data submitted from the client. However, since the session framework cannot be used as a production framework (is not stable enough to run with many clients), there are two possibilities to make the complex forms framework ready for production. First, the underlying problems of the session framework can be solved and thereby use the complex forms framework as it is now. Another possibility is to base the complex forms framework on CGI. Implementing the complex forms framework in a CGI environment means that it is possible to use the framework without having to install a new Apache server module.

In our opinion the object oriented approach solves the *Complex forms* problem. This is so, since the complex forms framework supports the developer to create, present and update a complex structure.

The third and final hypothesis, specifies how the *Input validation* problem was to be solved:

Hypothesis 3:

> ***It is possible to construct a validation framework that helps the developer to validate data from the client.***

This hypothesis stated that it is possible to solve the *Input validation* problem by constructing a validation framework. Since two frameworks were created (the session framework and the complex forms framework) to handle data, two approaches to validation existed. In the session framework the data handling consists of asking the client for data and return the data to the server. In the complex forms framework the data handling consists of updating the objects with the data from the client. By supporting validation on both the session framework and the complex forms framework it is possible to use validation on the two frameworks independent of each other.

If the validation framework must work under CGI, the validation functions must instead be handled explicitly by the developer. The reason for this is, that a check function must be defined in the script where it is used. A solution to this problem, is to create a "validation library" which is included in all scripts. This library can then consist of collections of validation functions available.

By extending the objects in the complex forms framework to contain validation functionality, we have made validation on the object structure possible. This is done by allowing the developer to define a check function to each of the nodes in an object structure. These check functions are then activated when the object structure is updated, thus setting the `valid` property on the objects in the object structure. Validation in the session framework is obtained by validating information received from individual interactions with a client. This is done, by specifying a check function as a parameter to each `slaml-show` where validation is wanted.

5.1 Future Work

This last part of the conclusion presents possible areas where focus for future work with the SLAML framework can be set. In order to use the SLAML framework in a broader context, it is needed to solve the problems related to the Apache module. If these problems are not solved, it is not possible to serve more than a single client at a time (see Section 4.2). This is not adequate for production use.

An aspect of the session concept, has not been considered in detail in this project. This is related to the possibility to step back in a session. A session consists of an amount of interactions with a client and we find it beneficial to allow, that a client can go back in a session to change information entered. Stepping back in a session means that the server will need to do accounting of how far the individual clients has reached in their sessions. Furthermore, it is necessary to undo actions performed by the client when the back button is pressed. This is a subject that can be investigated further.

A problem not considered in details in this project, is the need to make sessions persistent. This must be considered, since a client can pause the session (stop sending requests) for an amount of time. It is not known when - or even if - the client returns to continue the session. The problem with persistent sessions have two aspects. The first is to share the data between the Apache processes. This is not possible in the current implementation, because of the Apache process model. This problem can be solved by the MM library, which allows data to be shared between Apache processes. The other aspect of the persistence problem is to store sessions to disk. A possible solutions is to make the Scheme environment persistent, as e.g. done by Christian Queinnec [Que].

In Bigwig it is possible to access sessions directly by specifying their name as an URL parameter. This is done by letting the server have knowledge of all the sessions in a given service. In the current session framework this is only supported if handled explicit by the developer.

An example of this is shown in the *student-session* in the "Student Class" application. The solution in Bigwig is better since this approach requires no explicit control of the flow by the developer. Thereby it is possible to access a session by a link on a page.

To use the functionality from the old mod_laml it is necessary to implement all the features again. This includes support for more than one interpreter, registration of which libraries are loaded and logging facilities etc. This is a suggestion to future work in which the design from our Dat5 project can be used.

The last aspect that is considered, is how the use of a session concept changes the developers view on developing Web applications. This can be examined by performing an analysis of the difference between an application written in CGI and the same application based on the session concept. Aspects such as efficiency, lines of code, readability, reusability, development time and execution time can also be included here.

A

SLAML Reference

This appendix presents the primitives from the SLAML library. For each primitive the various characteristics are presented. The characteristics on each primitive is:

Name	The name of the primitive
Description	A description of the primitive
Form	The form in which the procedure is activated
Returns	The return value of the procedure
Required Parameters	The required parameters to the procedure
Optional Parameters	Optional parameters in form of named parameters

First functionality associated with the session framework are presented. Next functionality associated with the object framework are presented.

A.1 Session Framework

This section presents the primitives used in the session framework. This includes primitives for defining pages and sessions as well as primitives for activating sessions and showing pages.

Name:
`slaml-session` [Special form]
Description:
A function used to represent a session in SLAML. It is similar to a `lambda` function, taking one parameter. The *body* of this function contains the various pages and interactions with the client.
Form:
(slaml-session (*args*) *body*)
Returns:
A function representing the body of the slaml-session.

Required Parameters:

Name	Description
args	A parameter to be used during the activation of the session.

Name:
`slaml-page` [Special form]
Description:
A function used to represent a page in SLAML. It is similar to a `lambda` function with one parameter. The *body* of this function must evaluate to a string representation of an HTML page.
Form:
(slaml-page (*args*) *body*)
Returns:
A function representing a HTML page.
Required Parameters:

Name	Description
args	A parameter to the page.

Name:
`slaml-define-page` [Special form]
Description:
A function used to define a `slaml-page` in the Scheme environment.
Form:
(slaml-define-page *name slaml-page*)
Returns:
unspecified
Required Parameters:

Name	Description
name	The name to bind the *slaml-page* to.
slaml-page	The `slaml-page` to bind to *name*.

Name:
`slaml-define-session` [Special form]
Description:
A function used to define a `slaml-session` in the Scheme environment.
Form:
(slaml-define-session *name slaml-session*)
Returns:

unspecified

Required Parameters:

Name	Description
name	The name to bind the *slaml-session* to.
slaml-session	The `slaml-session` to bind to *name*.

Name:

`slaml-show` [Procedure]

Description:

Shows a page to the client and returns form parameters entered by the client. Used in the programs to ask or query a client for data. Control flow of the applications will return to the point just after the activation of `slaml-show`. If a *check* attribute is supplied the check function passed as attribute value is activated on the form parameters and the return value of the validation function becomes the return value of the check function.

Form:

(slaml-show *slaml-page* . *attributes*)

Returns:

The data entered by the client, or the return value of the optional validation function.

Required Parameters:

Name	Description
slaml-page	The page to be shown to the client. The page is created with the `slaml-page` primitive. It must be a `slaml-page` function.

Attributes:

Name	Description
check	An attribute indicating the validation function to be executed on the data returned from the client.
pageparm	An attribute indicating that parameters are passed to the page. The attribute value is the parameters to be passed. If more than one parameter is required, the attribute value is send as a list containing the parameters.

Name:

`slaml-activate-session` [Procedure]

Description:

Activates a session on the current location of the program. The value of the last expression is returned.

Form:

(slaml-activate-session *slaml-session* . *attributes*)

Returns:

The value of the last expression in the *slaml-session*

Required Parameters:

Name	Description
slaml-session	The session to be activated. The session must be specified with the `slaml-session` primitive.

Attributes:

Name	Description
sessionparm	An attribute indicating that a parameter is passed to the *slaml-session*. If more than one parameter is required a list of parameters is the attribute value.

Name:

`slaml-create-parm-lst` [Procedure]

Description:

A function to create the appropriate representation of the data received from the client, in an url encoded string. The string containing the keys and values - representing the contents of the HTML form presented to the client - are processed and a association list is created.

Form:

(slaml-create-parm-lst *form-parameter-string*)

Returns:

A list of key/value pairs, tagged with the `formparms` symbol.

Required Parameters:

Name	Description
form-parameter-string	A string in url encoded format

Name:

`slaml-key->value` [Procedure]

Description:

A function for searching association lists. Based on a key it extract the associated value.

Form:

(slaml-key->value *key a-lst*)

Returns:

The value that corresponds to *key* from *a-lst* or `#f` is *key* is not found.

Required Parameters:

Name	Description
key	The key to search for in *a-lst*. *key* must be a symbol.
a-lst	The list to search for *key*. The list must be an association list.

Name:
`slaml-formparms-key->value` [Procedure]
Description:
A function for extracting values associates with a key from a list tagged with `formparms`. The list of data entered into an HTML form by a client is returned tagged with the `formparms` symbol. If the key is not found, `#f` is returned.
Form:
(slaml-formparms-key->value *key lst*)
Returns:
The value that corresponds to *key* from *lst* or `#f` if *key* is not found.
Required Parameters:

Name	Description
key	The key to search for in *lst*. *key* must be a symbol.
lst	The association list tagged with `formparms` to search for *key*.

A.2 Object Framework

This part of the appendix presents functionality associated with the complex structure framework. First primitives associated with the classes are presented. Next various convenience functionality are presented. Then functionality for presentational tasks are presented. Finally a message passing primitive are presented.

A.2.1 Classes

This section describes the functions used to represent the different classes in the complex forms framework. The complex forms framework consists of functionality for building, presenting and updating complex HTML forms.

Name:
`slaml-element` [Class]
Description:

The function representing the `slaml-element` class. `slaml-element` is used to represent a composite object. Activating this function will create an object and return a reference to it.

Form:
(slaml-element . *attributes*)

Returns:
A reference to the newly created object.

Required Parameters:
None.

Instance Variable:

Name	Default Value	Description
name	"unique name"	The name of this object. Must be a string.
layout	""	The layout function of this object. Must be a string.
check	(lambda (str) #t)	The check function associated with this object.
elements	()	A list of objects rooted in this object.
tagtype	"slaml-element"	The tag identifying this object. It is used when the object structure is returned in list format from the `slaml-update-object!` function.

Name:
`slaml-basic-element` [Class]

Description:
The function represents the `slaml-basic-element` class, which represents basic HTML elements. It is used to represent HTML elements, e.g. input[W3C02b]. Activating this function will create an object and return a reference to it. Information present on the objects that is not related to the particular type, is ignored when `dolayout` is called. As an example the *maxlength* attribute is not used if the *type* is checkbox.

Form:
(slaml-basic-element . *attributes*)

Returns:
A reference to the newly created object.

Required Parameters:
None.

Instance Variables:

Name	Default Value	Description
name	"unique name"	The name of this object. Must be a string.
type	"TEXT"	The type of input field from HTML [W3C02b]. Possible values are TEXT, PASSWORD, CHECKBOX, RADIO, SUBMIT, RESET, FILE, HIDDEN, IMAGE, BUTTON. Must be a capitalized string.
size	"15"	The size of an textfield. Must be a string.
maxlength	""	The maximum length of an textfield. Must be a string.
checked	"false"	Indicates whether or not a check box is checked ("true" or "false"). Must be a string.
value	""	The default contents of this textfield. Must be a string.
check	(lambda (str) #t)	The check function associated with this object.
tagtype	"slaml-basic-element"	The tag identifying this object. It is used when the object structure is returned in list format from the `slaml-update-object!` function. Must be a string.

Name:
`slaml-form-element` [Class]
Description:
The function represents the `slaml-form-element` class, which represent an HTML form. Activating this function will create an object and return a reference to it. Objects of this type represents the root element in an object structure.
Form:
(slaml-form-element . *attributes*)
Returns:
A reference to the newly created object.
Required parameters:
None
Instance Variables:

Name	Default Value	Description
name	"unique name"	The name of this object. Must be a string.
action	"http://localhost"	The action associated with the form. Must be a string.
method	"GET"	Method of the action (either GET or POST). Must be a string.
enctype	"application/x-www-form-urlencoded"	The type of the HTML form encoding. Must be a string.
accept-charset	"UNKNOWN"	The character-set accepted in the form. Must be a string.
accept	"text/html"	Accepted content type. Must be a string.
layout	""	The layout function associated with this object. Must be a string.
elements	()	A list of references to other objects, rooted in this object. It must be a obj-lst.
tagtype	"slaml-form-element"	The tagtype identifying this object. It is used when the object structure is returned in list format from the `slaml-update-object!` function, represent the type of this object. Must be a string.

A.2.2 Convenience Functionality

In this section convenience functionality used to create object structures are described. This includes functionality for creating objects as well as functionality for creating a list of objects.

Name:
`slaml-create-basic-element` [Procedure]
Description:
A convenience function used to create an object of the class `slaml-basic-element`. `slaml-basic-element` represents a basic HTML `input` element [W3C02d].
Form:
(slaml-create-basic-element *name*)
Returns:
A reference to the newly created object.
Required Parameters:

Name	Description
name	The name of the object. It must be a string.

Name:
`slaml-create-element` [Procedure]
Description:
Used to create an object of the type `slaml-element`. `slaml-element` represents a composite objects used to address a group of objects as one.
Form:
(slaml-create-element *name*)
Returns:
A reference to the newly created object.
Required Parameters:

Name	Description
name	The name of the object. It must be a string.

Name:
`slaml-create-form-element` [Procedure]
Description:
Used to create objects of the type `slaml-form-element`. `slaml-form-element` represents an HTML form. An object of this type must be the top level object in the object structure.
Form:
(slaml-create-form-element *name*)
Returns:
A reference to the newly created object.
Required Parameters:

Name	Description
name	The name of the object. It must be a string.

Name:
`slaml-create-obj-lst` [Procedure]
Description:
A function that given a list of objects returns a list in the format required as the attribute value to the `elements` attribute. Objects of the types `slaml-element` and `slaml-form-element` has the `elements` instance variable. The list is tagged with the `slaml-obj-lst` symbol.
Form:
(slaml-create-obj-lst . *lst*)
Returns:
A specially formatted object list that is used as attribute value to the `elements` attribute name when creating SLAML objects.
Required Parameters:
None
Optional Parameters:

Name	Description
lst	A list of objects that can be included in the special formatted object list which is returned.

A.2.3 Functionality for Generating HTML

This section includes functionality for presenting the complex structure to the client. It also includes functionality used for working with the structure once data has been received from the client.

Name:
`slaml-do-layout` [Procedure]
Description:
A function that can be used to activate the layout of the object whose reference is passed as parameter. This function is used when generating the representation of an object. This function is used when the layout of a `slaml-form-element` is needed. In contrast to `slaml-do-layout-child`, which is used in the `slaml-layout` functions to do layout on the specified child.
Form:
(slaml-do-layout *obj*)
Returns:
A string representing the intended representation of the object in HTML terms.
Required Parameters:

Name	Description
obj	A reference to an object of the `slaml-form-element` class, on which the layout function is to be activated.

Name:
`slaml-do-layout-child` [Procedure]
Description:
Activate the layout function of a child object to a given parent object. Is used in the `slaml-layout` function to call the layout function of other objects. This allows for recursively generating the layout of all objects in the object structure. This function is used inside `slaml-layout` functions to call the layout on a specific child. In contrast `slaml-do-layout` is used when a `slaml-form-element` is presented.
Form:
(slaml-do-layout-child *parent childname*)

Returns:
An HTML string of the object with *childname* present in the *parent* object.
Required Parameters:

Name	Description
parent	A reference to the parent object of the object to layout.
childname	The name of the child to layout. It must be a string.

Name:
`slaml-update-object!` [Procedure]
Description:
A function to update the object structure - rooted in *obj* - with the data entered by the client. This function must be used explicitly to update the object structure. The format of the *parms* parameter must be the same format as the data returned by the `slaml-show` function. Besides updating the object structure it returns a nested list representation of the object structure. This list includes the following instance variables from the various objects as attributes; *tagtype*, *data*, *valid*.
Form:
(slaml-update-object! *obj parms*)
Returns:
A list representation of the object structure rooted in *obj*.
Required Parameters:

Name	Description
obj	A reference to the object that is the root of the object structure which is updated with the data from the client.
parms	The `form` parameters to be inserted into the object structure. Must be created by `slaml-show`.

Name:
`slaml-layout` [Special form]
Description:
A function used to activate layout functionality in the *self* object.
This function is equivalent to `lambda`. **Form**:
(slaml-layout (*self parm*) *body*)
Returns:
A reference to a function representing the layout.
Required Parameters:

Name	Description
self	The objects that contains this layout function.
parm	A parameter to be supplied to the activation of the layout function on *self*.

A.2.4 Message Parsing Functions

This section presents the get and set methods implemented for easy access to the objects instance variables and methods. Common to all of these are that the same functionality can be achieved by message passing with the `slaml-send` primitive.

Name:
`slaml-send` [Procedure]
Description:
A function used to activate functionality in the various objects.
Form:
(slaml-send *method obj . parm*)
Returns:
The result of evaluating *method* on *obj*.
Required Parameters:

Name	Description
method	The functionality that must be invoked on *obj*.
obj	The objects that contains the *method* to be activated.

Optional Parameters:

Name	Description
parm	A parameter to be supplied to the activation of *method* on *obj*. Must be a symbol.

Name:
`slaml-get-elements` [Procedure]
Description:
A function for extracting the elements lists from the object passed as parameter. The list returned is tagged with the `slaml-obj-lst` tag.
Form:
(slaml-get-elements *obj*)
Returns:
The element list of objects rooted in this object.
Required Parameters:

Name	Description
obj	The object to extract the elements list from. Must be a `slaml-form-element` or `slaml-element`.

Name:
`slaml-get-name` [Procedure]
Description:
A function used to extract the value of the name instance variable from the object passed as parameter.
Form:
(slaml-get-name *obj*)
Returns:
The value of the name instance variable of *obj*.
Required Parameters:

Name	Description
obj	The object to extract the value of the `name` instance variable from as a string.

Name:
`slaml-get-valid` [Procedure]
Description:
A function used to extract the value of the valid instance variable. The result is either `#t` or `#f`, indicating whether the data entered by the client and validated with a supplied validation function is valid. The valid instance variable is used to indicate whether an eventual validation on the objects failed.
Form:
(slaml-get-valid *obj*)
Returns:
A boolean value
Required Parameters:

Name	Description
obj	The object to extract the value of the valid instance variable from.

Name:
`slaml-set-valid!` [Procedure]
Description:
A function for setting the valid instance variable on an object. The valid instance variable is used to indicate whether an eventual validation on the objects failed.

Form:
(slaml-set-valid! *obj valid*)
Returns:
Nothing
Required Parameters:

Name	Description
obj	The object to set the *valid* instance variable on.
valid	The value to be set on *obj*. It must be a Scheme true or false (`#t`, `#f`).

Name:
`slaml-get-type` [Procedure]
Description:
A function for extracting the type of an object. The return value is a string representing the type of the object. Possible value are check, text, radio etc.
Form:
(slaml-get-type *obj*)
Returns:
The type of the object as a string.
Required Parameters:

Name	Description
obj	The object to extract the value of the type instance variable from. *obj* must be of the type `slaml-basic-element`.

Name:
`slaml-get-tagtype` [Procedure]
Description:
A function used to extract the values of the `tagtype` instance variable of an object. The tagtype is used as identification of the object, when a nested list representation of an object structure is returned from `slaml-update-object!`.
Form:
(slaml-get-tagtype *obj*)
Returns:
The tag type of an object, as a string.
Required Parameters:

Name	Description
obj	The object to extract the tag type from.

Name:

`slaml-get-check` [Procedure]
Description:
A function used to retrieve the validation function associated with the object passed as parameter.
Form:
(slaml-get-check *obj*)
Returns:
A reference to a function object.
Required Parameters:

Name	Description
obj	The object who's check function is wanted.

Name:
`slaml-set-data!` [Procedure]
Description:
A function to set the `data` instance variable of the object passed as parameter. The value set on the `data` instance variable is the supplied value.
Form:
(slaml-set-data! *obj value*)
Returns:
Nothing
Required Parameters:

Name	Description
obj	The objects who's `data` instance variable is set to *value*.
value	The value to be set on the `data` instance variable of *obj*. It must be a string value.

Name:
`slaml-get-data` [Procedure]
Description:
A function used to retrieve the value of the `data` instance variable.
Form:
(slaml-get-data *obj*)
Returns:
The value of the data instance variable in *obj*.
Required Parameters:

Name	Description
obj	The object who's data value is wanted.

B

Small Example

This appendix contains the functionality of the "Guess a number" application, presented in Section 3.1. First various functionality is defined. Then the various objects and their associated layout functions used in the example are created. This also includes the definition of three HTML pages. Last is the session in the example.

Utils

The different utilities used in the "Guess a number" application.

```
;======== Utils to the Guess a number application ============;
(define NaN -1) ;Not a Number

;return 42, since no function exist for generating a random number in Scheme
(define (get-random-number) 42)

(define (slaml-is-integer? str)
  (integer? (string->number str))
  )

(define (get-guess lst)
 (car lst)
  )

(define (get-guesses lst)
  (cadr lst)
  )

(define (get-right-number lst)
  (caddr lst)
  )


(define (get-hint guess-info) ;;checks information in order to set the correct hint
  (let  ((guess (get-guess guess-info))
         (guesses (get-guesses guess-info))
```

```
        (right-number (get-right-number guess-info)))
    (string-append
     (cond
      ((equal? 0 guesses) "Enter your first guess.") ;first time, no guess yet
      ((not (slaml-get-valid guess-input)) "HINT: Try a number next time.")
      (else (string-append "HINT: Your guess was: "
                           (cond
                            ((> guess right-number) "too high... try a lower.")
                            ((< guess right-number) "too low... try a higher.")
                            )))
     )
     )
    )
  )
```

Layout

The definition of the objects and their associate layout functions. This section also presents the definition of the HTML pages.

```
(load "utils.slaml")
;;====== The objects needed in the Guess a number application ==========;
(define guess-form-layout
  (lambda (self parms)
     (slaml-do-layout-child self "guess-composite")
    )
  )

(define guess-composite-layout
  (lambda (self parms)
    (string-append
     "Enter your guess:"
     (table
      (tr (td (slaml-do-layout-child self "input-field")))
      (tr (td (slaml-do-layout-child self "submit-button")))
      )
     )
    )
  )

(define guess-input
  (slaml-basic-element
   'check slaml-is-integer?
   'name "input-field"
   'tagtype "input-field-guess")
  )

(define submit-guess-button
  (slaml-basic-element
   'name "submit-button"
   'tagtype "submit-button-guess"
   'type "SUBMIT"
   'value "Guess")
```

```
  )

(define guess-composite
  (slaml-element
   'layout guess-composite-layout
   'elements (slaml-create-obj-lst guess-input submit-guess-button)
   'name "guess-composite"
   'tagtype "composite-guess")
  )

(define guess-form
  (slaml-form-element
   'layout guess-form-layout
   'name "guessform"
   'action ""
   'method "GET"
   'tagtype "guess-form"
   'elements (slaml-create-obj-lst guess-composite)
   )
  )

;;======================== Simple pages ============================;

;say hello
(slaml-define-page start-game
  (slaml-page (lst)
        (html
         (head (title "Guess a number"))
         (body (h1 "Welcome to guess a number... ")
               (p "This application is written in SLAML.")
               (hr)
               (p "You must guess a number between 1 and 100")
               (a "continue" 'href "http://localhost/laml/guess-app/guess-number.slaml")
               )
         )
        )
  )

; say goodbye
(slaml-define-page end-game
  (slaml-page (guesses)
        (html
         (head (title "Guess a number"))
         (body (h1 "Congratulations")
               (p "You made it in " (number->string guesses) "guesses!"))
         )
        )
  )

(slaml-define-page guess-page
  (slaml-page (parameter-list)
        (html
         (head (title "Guess a number!"))
         (body
```

```
          (get-hint parameter-list)
          (hr)
          (slaml-do-layout guess-form)
          )
         )
        )
  )
```

Main

The definition of the session in the example. The final line activates the session.

```
(load "layout.slaml")
;;============================================================
;; Guess a number application
;;============================================================

(slaml-define-session guess-session
 (slaml-session (session-param)
  (slaml-show start-game) ;say hello - step one
  (letrec ((guess-loop
        (lambda (guess guesses right-number)
          (if (equal? guess right-number)
              guesses ; Return the number of guesses used
              (let
                  ((obj-struct
                    (slaml-update-object! guess-form
                      (slaml-show
                        guess-page 'pageparm (list guess guesses right-number))))
                   )
                (if (slaml-get-valid guess-input)
                    (guess-loop
                     (string->number
                       (slaml-get-data guess-input)) (+ 1 guesses) right-number)
                    (guess-loop
                     NaN (+ 1 guesses) right-number)
                    )
                )))))
    (let* ((right-number (get-random-number))
           (guesses (guess-loop 0 0 right-number)) ;do loop - step two
           )
      (slaml-show end-game 'pageparm guesses) ;say bye - step three
      guesses ; return the number of guesses used
      ) ;end letrec
    )))

(slaml-activate-session guess-session) ; it starts
```

Bibliography

[AB84] Andrew D. Birrel and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Conmputer Systems*, 2(1):39–59, 1984.

[apa01] Apache Homepage. `http://www.apache.org/`, September 2001.

[asp01] Introduction to Active Server Pages. `http://msdn.microsoft.com/library/en-us/iisref/html/psdk/asp/iiwaabt.asp`, September 2001.

[BMRS01] Claus Brabrand, Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach. PowerForms: Declarative Client-Side Form Field Validation. `http://www.brics.dk/bigwig/research/publications/powerform.ps`, October 2001.

[CAM02] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> Project. *ACM Transactions on Internet Technology*, 2002. It is to appear in the journal.

[cgi01] The CGI Specification. `http://hoohoo.ncsa.uiuc.edu/cgi/interface.html`, November 2001.

[CGKF02] John Clements, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Little Languages and their Programming Environments. `http://www.cs.rice.edu/CS/PLT/Publications/mw01-cgkf.pdf`, May 2002.

[DJ01] David A. Ladd and J. Christopher Ramming. MAWL. `http://www.bell-labs.com/project/MAWL/mawl.html`, December 2001.

[DL02] D. Kristol and L. Montulli. HTTP State Management Mechanism. `http:/www.ietf.org/rfc/rfc2109.txt/`, Feburary 2002.

[ECH02] ECHMA. ECHMAScript Language Specification. `http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM`, may 2002.

[Eng02a] Ralf S. Engelschall. MM Shared Memory Library. `http://www.engelschall.com/sw/mm/`, May 2002.

[Eng02b] Ralf S. Engelschall. mod_ssl. `http://www.modssl.org/`, June 2002.

[ERRJ95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Fla02] Matthew Flatt. PLT MzScheme: Language Manual. http://download.plt-scheme.org/doc/200alpha12/html/mzscheme/, June 2002.

[gui01] Guile Homepage. http://www.gnu.org/software/guile/guile.html, September 2001.

[Han01] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[Han02] Michael Hanus. The Portland Aachen Kiel Curry System. http://www.informatik.uni-kiel.de/~pakcs/, May 2002.

[Inc01] Sun Microsystems Inc. Java 2 Platform Enterprise Edition. http://java.sun.com/j2ee, November 2001.

[Inc02a] Sun Microsystems Inc. Java Object Serialization Specification. http://java.sun.com/j2se/1.4/docs/guide/serialization/spec/serialTOC.doc.html, June 2002.

[Inc02b] Sun Microsystems Inc. The Source for Java Technology. http://java.sun.com/, June 2002.

[JO02] John Peterson and Olaf Chitil. The Haskell Home Page. http://www.haskell.org/, Feburary 2002.

[KCR+98] Richard Kelsey, William Clinger, Jonathan Rees, H. Abelson, H. I Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele Jr., G. J. Sussman, and M. Wand. *Revised Report on the Algorithmic Language Scheme*. 1998.

[lam01] The LAML Home Page. http://www.cs.auc.dk/~normark/laml/, August 2001.

[LDJ02] Christian Lynbech, Mikael Djurfeldt, and Niel Jerram. Goops manual. http://www.gnu.org/software/goops/goops.html, June 2002.

[Mic02a] Michael Hanus. The Functional Logic Language Curry. http://www.informatik.uni-kiel.de/~curry/, May 2002.

[Mic02b] Microsoft. JScript. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/js56jsoriJScript.asp, June 2002.

[Mic02c] Microsoft Corporation. VBScript. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vtoriVBScript.asp, May 2002.

[MPJ02] Mikael Hansen, Paw Iversen, and Jimmy Juncker. SLAML - Server side LAML. Technical report, Aalborg University, 2002.

[Net02] Netscape. JavaScript Developer Central. http://developer.netscape.com/tech/javascript/index.html, June 2002.

[Nør90] Kurt Nørmark. Simulation of Object-oriented Concepts and Mechanisms in Scheme. Technical Report R 90-01, Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, January 1990.

[Nør00] Kurt Nørmark. A Programmatic Approach to WWW Authoring Using Functional Programming. http://www.cs.auc.dk/~normark/laml/papers/old-programmatic-approach.pdf, November 2000.

[Pet] Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions, Compositional Forms, and Graphics. http://www.informatik.uni-freiburg.de/~thiemann/papers/cgi-in-haskell.ps.gz.

[Pra02] Vipul Ved Prakash. Cgi::persistent. http://search.cpan.org/doc/VIPUL/CGI-Persistent-0.22/lib/CGI/Persistent.pm, May 2002.

[Que] Christian Queinnec. The Influence of Browsers on Evaluators or, Continuations to Program Web Servers.

[Que02] Christian Queinnec. Meroon: an Object System in Scheme. http://youpou.lip6.fr/queinnec/WWW/Meroon.html, June 2002.

[SM02] Inc. Sun Microsystems. JavaServer Pages(TM) Technology. http://java.sun.com/products/jsp/, Feburary 2002.

[The02] The PHP Group. Session handling functions. http://www.php.net/manual/en/ref.session.php, Feburary 2002.

[W3C02a] W3C. Extensible Markup Language (XML). http://www.w3.org/XML, June 2002.

[W3C02b] W3C. The HTML 4.01 specification. http://www.w3.org/TR/html401/, may 2002.

[W3C02c] W3C. W3C Recommendation - The form element. http://www.w3.org/TR/html4/interact/forms.html#h-17.3, June 2002.

[W3C02d] W3C. W3C Recommendation - The input element. http://www.w3.org/TR/html4/interact/forms.html#h-17.4, June 2002.

[WG02] W3C DOM WG. Document Object Model (DOM). http://www.w3.org/DOM/, May 2002.