Makumba och JavaServer Faces 2.0

Utvärdering av möjligheten att implementera
ramverket Makumba på JavaServer Faces 2.0-plattformen

BJÖRN SVEDERUD
och FILIP HINDEMARK

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

Makumba och JavaServer Faces 2.0

Utvärdering av möjligheten att implementera
ramverket Makumba på JavaServer Faces 2.0-plattformen

B J Ö R N S V E D E R U D
o c h F I L I P H I N D E M A R K

Examensarbete i teknik och management om 15 högskolepoäng
vid Programmet för industriell ekonomi
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Christian Bogdan
Examinator var Stefan Arnborg

URL: www.csc.kth.se/utbildning/kandidatexjobb/teknikmanagement/2010/
svederud_bjorn_OCH_hindemark_filip_K10051.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Sammanfattning

Makumba är ett ramverk för webbapplikationsutveckling idag implementerat på en åldrande teknologi, JavaServer Pages. JavaServer Faces har sedan Makumbas implementering tagits fram för att ersätta JavaServer Pages som Javas standardteknologi för webbapplikationsutveckling. Nyligen släpptes JavaServer Faces 2.0 som ämnar förenkla framtagande av webbapplikationer än mer.

Avhandlingen avser, genom ett konceptbevis, visa att en återimplementering av ramverket Makumba på JavaServer Faces 2.0 är möjlig. Tekniken bakom JavaServer Faces 2.0 gås igenom och en analys av dess implikationer för genomförandet av konceptbeviset utförs. Konceptbeviset visar att två taggar från Makumba kan implementeras på JavaServer Faces. Detta bereder väg för en full återimplementering av Makumba. Avslutningsvis diskuteras hur denna implementering kan utföras, vad den skulle innebära för Makumba, samt olika områden som behöver närmare undersökning innan en full implementering slutligen kan genomföras.

Abstract

Makumba Framework is currently implemented on top of an aging technology, JavaServer Pages. JavaServer Faces was developed to replace JavaServer Pages, and has recently received a new major release. One of the main goals of JavaServer Faces 2.0 is to simplify web application development.

This thesis aims to show that a re-implementation of Makumba on JavaServer 2.0 is possible through a proof of concept that aims to implement two of the most important Makumba tags. Certain aspects of JavaServer Faces 2.0 important for development of a proof of concept as well as the proof of concept itself are analyzed. The results show that an implementation is possible, as well as hinting towards other possible ways to perform a complete integration of Makumbas and JavaServer Faces. Drawing from the results a discussion about the implementation as well as the implications of the implementation is held.

Innehåll

Kapitel 1

Introduktion

1.1 Frågeställning

Makumba är ett ramverk för utveckling av webbapplikationer som idag är implementerat på en åldrande teknologi, JavaServer Pages. I avhandlingen skall visas genom ett konceptbevis att ramverket Makumba kan återimplementeras på en nyare teknologi såsom JavaServer Faces, samtidigt som dess enkla frågebaserade språk bibehålls. Genom en implementering av Makumba på en nyare teknologi kan redan existerande webbapplikationer skrivna med Makumba dra nytta av den nya tekniken. Detta samtidigt som utvecklingen av webbapplikationer med Makumbas ramverk på den nya teknologin till stor del hålls intakt.

1.2 Metod

För att svara på frågeställningen har ett konceptbevis genomförts baserat på två taggar i Makumba. De taggar som ansågs lämpliga att återimplementera på JavaServer Faces är **<mak:list>** samt **<mak:value>**. Dessa taggar valdes ut i samråd med ledande Makumba-utvecklare. Deras uppgift är att hämta data från en databas genom frågor och sedan presentera svaren i listform. Dessa två använder sig av en del av Makumbas ramverk kallat *Page Analysis*. *Page Analysis* är en av Makumba tillagd fas i JavaServer Pages livscykel, och ytterst nödvändig för att Makumba ska fungera. En viktig del av konceptbeviset är att visa hur den funktionalitet erbjuden av *Page Analysis* kan tillämpas på JavaServer Faces. Därmed gör det dem till goda kandidater för att visa om det är möjligt och i så fall hur Makumbas funktionalitet kan uppnås även på plattformen JavaServer Faces.

Steg ett var därmed att analysera **<mak:list>** och **<mak:value>** och bestämma vilka av deras aspekter att implementera. Därefter krävdes undersökning av några delar av JSF som ansågs vara relevanta för implementeringen.

Plattformen JSF 2.0 levereras med en mängd redan implementerade standardkomponenter. Några av dessa har ett beteende likt det som **<mak:list>-** och **<mak:value>-**taggarna innehar. Den använda metoden för framtagande av **<mak:list>** har varit att utgå ifrån en existerande komponents källkod, modifiera befintliga metoder samt implementera nya metoder för att uppnå det önskade beteendet. För **<mak:value>** skapades en ny komponent som skrev över ärvda metoder.

1.3 Avgränsningar

Avhandlingen är utarbetad till de som ämnar genomföra en fullskalig integrering av Makumba på plattformen JavaServer Faces.

<mak:list> har ett stort omfång av metoder. Fokus ligger på implementering av dess huvudfunktionaliteten. Genomförandet av detta påvisar möjligheten att utelämnad funktionalitet kan implementeras via liknande tillvägagångssätt. De metoder som har valts att implementera är de som skall ställa frågor till databasen utifrån angivna attribut;

- FROM - talar om vilken databas som data ska hämtas ifrån
- WHERE - begränsar den data som ska hämtas ur databasen enligt det villkor som anges

Svaren på dessa frågor skall skrivas ut. Ytterligare ett krav är att utföra en korrekt iteration som resulterar i rendering i HTML. I avhandlingen behandlas den källkod som har modifierats och den kod som har lagts till i källkoden. Den källkod som behållits intakt har visat sig vara av mindre vikt för detta konceptbevis och kommer därmed inte diskuteras i samma utsträckning.

I vanliga fall arbetar **<mak:list>** mot en databas för att hämta och lagra data. För konceptbeviset valdes att simulera en databas genom att använda en **Managed Bean**. Den är enklare att arbeta med då problematiken med databaskopplingar helt försvinner, all data hämtas istället från en instans av den simulerade databasen.

Önskat beteende för konceptbeviset är att samla ihop alla frågor ifrån nästade barn, ställa dessa, spara resultaten, och göra en korrekt rendering av sidan, vilket kräver en korrekt iterering över resultaten. För att möjliggöra utveckling av ett konceptbevis

behövdes närmare undersökning inom delar av JavaServer Faces. Därför togs fyra frågor fram som behövdes svara på innan ett konceptbevis kunde genomföras:

- Hur fungerar sidor i JSF (hur utvecklas de, vad består de av, och hur renderas de)?
- Hur utvecklas nya komponenter i JSF?
- Hur fungerar JSF:s livscykeln?
- Hur lagras information i JSF över sidor?

Valet av avgränsningar utformades efter de krav som ställs på **<mak:list>** och **<mak:value>**. Det resultat som presenteras i denna avhandling är endast ett konceptbevis, vars syfte är att visa om och i så fall hur en Makumba Tag kan implementeras på plattformen JavaServer Faces.

1.4 Källdiskussion

JavaServer Faces 2.0 är en ny modern teknologi för utvecklingen av webbapplikationer. Det har medfört att mängden tillgänglig information är något begränsad jämfört med äldre teknologier. Speciellt finns det lite information om utveckling av avancerade komponenter för ramverk. För att skapa en djupare förståelse för JavaServer Faces 2.0 har böcker skrivna av bland annat utvecklarna använts.

För exempel på kod och lösningar på olika problem har flertalet webbforum dedicerad till webbutveckling använts, där användare själva delar med sig av sina lösningar. Källor som kodexempel och lösningar på kodproblem hämtats ifrån behöver ej någon djupgående granskning eftersom koden testas genom implementering. Dock behöver den lånade koden ej vara den optimala lösningen.

Gällande teori och funktionalitet hos JavaServer Faces är det viktigt att förhålla sig kritisk till källan. Därmed har information hämtats från litteratur rekommenderad av oberoende part, information ifrån JavaServer Faces utvecklare, samt Sun Microsystems egen dokumentation.

1.5 Struktur

Avhandlingen är uppdelad i fyra avsnitt. I det första avsnittet beskrivs hur Javateknologin för utvecklingen av webbapplikationer har skett. Här följs de olika stegen

i utvecklingen från Servlets till JavaServer Faces. Detta avsnitt introducerar även det generella begreppet ramverk samt Makumba. Makumba är det ramverk som denna avhandling behandlar. Det andra avsnittet redovisar den undersökning som gjordes för att möjliggöra implementering av konceptbeviset. Undersökningen består av två delar. Först görs en analys av de taggar som ska implementeras på JSF. Därefter redovisas svaren på de frågor som ställdes om JSF. Dessa delar av JSF är viktiga att ha insyn i för att förstå de ändringar som genomförts i konceptbeviset. Det är även av vikt att förstå de olika delarna av JSF som tas upp för den som ämnar att utveckla egna komponenter på plattformen JSF.

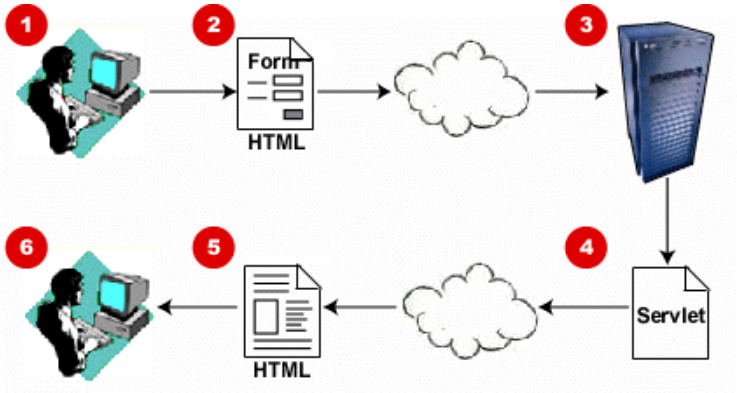
Genom förgående avsnitt förbereds för konceptbeviset, som gås igenom grundligt i avhandlingens tredje del. Detta avsnitt är tämligen kodtung, och är främst en redovisning av de lösningar som tillämpades för att lösa avhandlingens problem. Slutligen avslutas avhandlingen med en diskussion om implikationerna av konceptbeviset, andra möjliga sätt att lösa problemet, och hur en full implementering av Makumba skulle kunna genomföras. Eftersom detta är ett konceptbevis har de komponenter som utvecklats något annorlunda funktionalitet än de taggar som finns implementerade på Makumba idag. Därmed diskuteras även hur taggarna bör utvecklas vidare för att uppnå den funktionalitet som en full implementering kräver.

Kapitel 2

Bakgrund

2.1 Servlets

Servlets var Javas lösning på problemet med HTML:s uteslutande statiska karaktär. Servlettekniken utvecklades för att skapa dynamiska webbapplikationer. I kontrast till HTML genererar Servlets dynamiska svar på förfrågningar. Servlets är Java-klassfiler som hanterar förfrågningar dynamiskt och genererar svar oberoende av protokoll[2]. De är program som ligger på serversidan och utökar serverns funktionalitet. Servlets är oberoende av protokoll och används oftast för att hantera förfrågningar i HTTP och generera svar i HTML[2]. En *servlet* fungerar som en webbapplikations logik, den bestämmer vad som skall göras och hur.



Figur 2.1: Servlet livscykeln
url: http://java.sun.com/products/servlet/articles/tutorial/index.html

En förfrågan skickas till en server som skapar objekten **HttpServletRequest** och

HttpServletResponse, och skickar dessa vidare till en instans av *servleten*, om en sådan finns[2]. Finns ingen skapar servern en instans av *servleten* och skickar objekten till den nya instansen. Objekten som skickas till *servleten* innehåller informationen som ska behandlas och metoder för hur det skall skickas tillbaka. När informationen behandlats av *servleten*, genereras ett svar som skickas tillbaka till servern, som vidarebefordrar informationen till klienten.

Som koden nedan visar måste programmeraren skriva en **out.println()**-anrop per HTML- rad. Det skapar problem av separationskaraktär eftersom HTML måste skrivas inuti applikationslogiken[1]. Samma person behöver ej vara ansvarig för innehållet och funktionaliteten i webbapplikationen. Därmed är möjligheten att påverka layout utan att påverka logik och logik utan att påverka layout eftersträvansvärt. *Servlets* problematiserar detta i och med att HTML blandas med applikationslogik. Att arbeta på detta sätt är inte effektivt för stora webbprojekt. Det blir även svårt att skriva och tyda koden för logiken när den blandas med massvis av HTML[1].

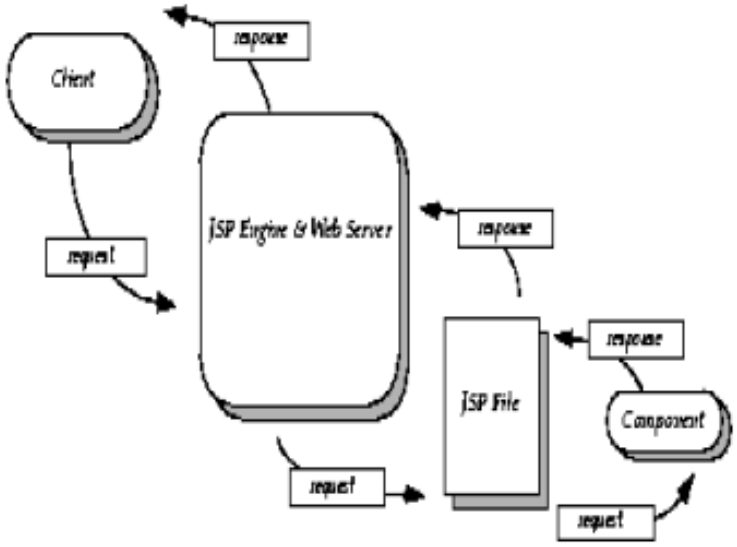
```
public class HelloWorld extends HttpServlet
{

      public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws IOException,
      ServletException
      {

            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<HTML>");
            out.println("<HEAD>");
            out.println("<TITLE>Hello World!!</TITLE>");
            out.println("</HEAD>");
            out.println("<BODY>");
            out.println("Hello World! Your name is: "
            + response.getParameter("name"));
            out.println("</BODY>");
            out.println("</HTML>");

      }

}
```

Figur 2.2: Servlet kodexempel
url: http://devcentral.iftech.com/articles/Java/intro_Servlet_JSP_Engine/default.php

2.2 JavaServer Pages

Lösningen på problemet att behöva skriva mängder av HTML i applikationslogiken kom i form av JavaServer Pages (JSP). Målet med JSP var att minska arbetet och svårigheten involverade i skapandet och underhållet av webbapplikationer[1]. JSP

är en vidareutveckling av Servlettekniken, där logiken skrivs i en JSP-fil istället för i en Javaklass. En JSP-fil består utav HTML-kod för det statiska innehållet samt små delar av Java-kod för dynamisk funktionalitet[1]. Mängden Javakod minskades samtidigt som Servletfunktionaliteten bevarades. Servern kompilerar automatiskt om Javakod inuti en JSP-fil till en *servlet* efter ett anrop genom en JSP-Server. Svaret till klienten renderas av en JSP-kompilator i HTML-kod[1].



Figur 2.3: JSP livscykeln
url: http://java.sun.com/products/jsp/html/jspbasics.fm2.html

Målet med JSP var att separera presentationen av innehållet och logiken. Vad som egentligen har skett är att HTML-koden har förflyttats ur Java-koden och istället har Java-koden förflyttats till HTML-koden. För små applikationer är JSP-tekniken ändock mer effektiv och enklare att arbeta med än Servlettekniken. För större applikationer kvarstår dock problemen. Tekniken JavaBeans utvecklades för att råda bot på dessa problem.

JavaBeans tillåter utvecklaren att skapa separata klassfiler för olika delar av logiken[17]. Dessa kan anropas ifrån JSP-sidor istället för att upprepa kod på flera olika sidor med samma funktionalitetskrav. Tekniken ökar separationen av presentation och logik samt minskar redundans. Koden ovan visar hur en JSP-sida kan skrivas för att utföra samma arbete som Servletexemplet. Resultatet blir detsamma fast mängden kod är mycket mindre och mer lättöverskådlig.

För att råda bot på de problem associerade med de tidiga versionerna av JSP, bland annat att större webbprojekt och webbsidor blev väldigt komplexa och svåröverskådliga, skapades JavaServer Pages Standard Tag Library (JSTL). JSTL består

```
<HTML>
   <HEAD>
      <TITLE>Hello World!!</TITLE>
   </HEAD>
   <BODY>
      Hello World! Your name is:<%
out.println(response.getParameter("name"));%>
   </BODY>
</HTML>
```

Figur 2.4: JSP kodexempel
url: http://devcentral.iftech.com/articles/Java/intro_Servlet_JSP_Engine/default.php

av Java-kod som anropas med XML-taggar[15]. Detta minskar både mängden kod och koduppprepningar. Dock kvartstår problemet att programlogik beblandas med presentationens HTML-kod.

2.3 Ramverk

För att förenkla och göra utvecklingen av webbapplikationer effektivare finns det mängder av tillgängliga ramverk för webbutveckling. Ett ramverk definieras som ett set av klasser som skapar en återanvändbar design för en specifik klass av mjukvara. Ramverk ger arkitektoniskt stöd genom att partitionera designen i abstrakta klasser och definierar deras ansvar[7]. Ramverk kan även ses som ett pussel, nästan färdigt, vars slutgiltiga bild varierar beroende på vilka bitar som stoppas in[8]. Kodåteranvändning är en viktig aspekt för ramverk. Makumba är en instans av det abstrakta begreppet ramverk. Ett bra ramverk behöver balansera mellan standardstruktur och flexibilitet. Det är väsentligt att delar av arkitekturen lämnas tom. Detta för att användaren ska kunna anpassa arkitekturen till problemet. Möjligheten måste också finnas för vidareutvecklingen av ramverk[10]. Praktiskt sett är ett ramverk en samling färdigskrivna taggar i ett taggbibliotek.

De ramverk som finns idag har olika sätt att arbeta och utföra uppgifter på samtidigt som de har olika mål att uppfylla för slutanvändaren. En del ramverk förenklar och har kraftfullt stöd för presentation av data och information, andra bidrar med effektivare metoder för lagring och hantering av olika datatyper.

Ett exempel på ett ramverk är Struts. Struts var bland det första ramverket att använda sig av en standardiserade livscykel, något som sedan anammats av efterföljande ramverk. Att JavaServer Faces marknadsfördes som en blandning av Struts och Java Swing när det kom är ett bevis på Struts genomslag[12]. Struts blev så pass populärt och använt bland utvecklare att personer involverade i utvecklingen av Struts har varit delaktiga i utvecklingen av JSF.

Det finns populära ramverk som redan gjort en övergång till att basera sig på JSF. Ett utav dessa är just Struts, men även Spring Framework har gjort denna övergång även om de inte är helt färdiga är grunden färdig och utvecklingen pågår[11] [18].

Ramverket Makumba

Makumba är ett ramverk byggt på JSP-plattformen vars syfte är att förenkla utvecklingen och underhållet av dynamiska webbsidor. Makumba är designat med utgångspunkten att erbjuda endast ett fåtal funktioner samt att ha en minimalistisk design. Det är med anledningen att en sådan utvecklingsmiljö skall vara lättare för en ny användare att lära sig använda[5]. Huvudfunktionaliteten som Makumba erbjuder användare är att hantera data på olika sätt, till exempel att redigera data, ta bort data, lägga till data och hämta data. För att utnyttja Makumba krävs ingen bred kunskap om *servlets*, JSP, eller Java. Istället behöver utvecklaren endast lära sig Makumbas egna språk baserat på Makumbas taggbibliotek för att snabbt och enkelt utveckla dynamiska webbapplikationer. Språket i Makumba liknar mycket SQL-språket. Det är ingen slump, då SQL-språket är baserat på talspråk för enkelhetens skull och Makumba följer samma idé där ett frågebaserat språk liknande talspråket är enkelt att lära och använda och känns intuitivt[5].

För att förstå den effekt ett ramverk har för utvecklaren av en webbapplikation är det viktigt att förstå vad som sker i koden nedan. I exemplet importeras Makumba taggbibliotek och tilldelas prefixet mak. Den kopplar ihop taggen med taggbiblioteket. De taggar som används nedan är **<mak: input>** samt **<mak: newForm>** där **<mak: newForm>** skapar ett formulär och varje **<mak: input>** skapar ett textinmatningsfält.

När sedan JSP-kontrollern läser igenom webbsidan, hittar den Mak-taggarna, och letar efter de filer som innehåller taggarnas logik. Detta görs på den plats som specificerats i *headern*. Programmet returnerar sedan data sedan JSP-motorn hanterar och renderar för klienten.

2.4 JavaServer Faces

Genom att flera ramverk existerande på den äldre JSP-teknologin fått stor användarkrets och ökad popularitet insåg utvecklarna av JavaServer Faces att det saknades flera nödvändiga aspekter i JSP. De aspekter som ansågs saknas och sedan blev grunden till JSF-teknologin var utnyttjandet av designmönstret MVC och ett ramverk fokuserat på utvecklingen av användargränssnitt. Resultatet var ett ramverk som använder en MVC samtidigt som det bygger på användningen av UI-

```
<%@ taglib uri="http://www.makumba.org/presentation" prefix="mak" %>
<html>
<head>
<title>Create new employee record</title>
</head>
<body>
<mak:newForm type="company.Employee" method="post" action="index.jsp">
  <table>
    <tr>
      <th>Name</th>
      <td><mak:input field="name" /></td>
    </tr>
    <tr>
      <th>Surname</th>
      <td><mak:input field="surname" /></td>
    </tr>
    <tr>
    <th>Birthdate</th>
    <td><mak:input field="birthdate" /></td>
    </tr>
    <tr>
    <th>Salary</th>
    <td><mak:input field="salary" /></td>
    </tr>
    <tr>
      <td>
            <input type="submit" value="Add">
            <input type="reset" value="Cancel" onClick="javascript:back();">
      </td>
    </tr>
  </table>
</mak:newForm>
</body>
</html>
```

Figur 2.5: Makumba kodexempel
url: http://www.makumba.org/page/QuickStart

komponenter. Dessa UI-komponenter kan hantera många av de vanligaste delarna av en webbapplikation och underlättar därmed utvecklingen av webbsidor.

JSF bidrar med färdig livscykel-, event-, och inputhantering, sköter rendering, och levereras med ett komponentbaserat ramverk. Främst är dock JSF en standard, vilket innebär att utvecklare inte behöver begränsa sig till en utvecklingsklient[16]. Innan JSF 2.0 bestod JSF i grova drag av två huvudsakliga delar. Förutom att skapa en MVC-inspirerad utvecklingsmiljö, innehar JSF två olika taggbibliotek för JSP ämnade för att uttrycka UI-komponenter inom en JSP-sida och för att knyta dessa komponenter till serversidans objekt. Knappar, länkar, etc definieras av JSF:s komponentarkitektur, baserade på designmönstret *composite*, som även tillåter utveckling av egna komponenter[9]. Att JSF är baserat på en MVC-arkitektur ökar möjligheterna att separera design ifrån logik, ett av huvudmålen med JSF[16].

Eftersom JSP har en egen livscykel, uppstod problem med att kombinera JSF och JSP. Av den anledningen utvecklades *Facelets* som från och med JSF 2.0 blev stan-

dard för vyhantering. Med den nya tekniken *Facelets* underlättas hanteringen av hela webbsidor för utvecklaren av webbapplikationer men även utvecklingen av skräddarsydda komponenter. JSF 2.0 innebar också förenklingar för navigationsregler och andra delar av JSF som tidigare krävde deklarering i en konfigurationsfil.

Kapitel 3

Undersökning

3.1 Analys av <mak:list> och <mak:value> i Makumba

<mak:list>

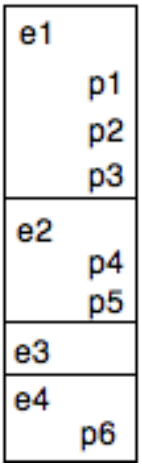
Med syfte att förenkla utveckling av webbapplikationer är Makumbas taggar framtagna för att efterlikna SQL-frågor, vilket även påverkar den bakomliggande logiken. Det är **<mak:list>** som ställer frågan till databasen. Givet en SQL-fråga, står denna tagg för FROM, WHERE, ORDER BY, etc. SELECT tas dock om hand om av **<mak:value>**.

SQL-exempel:

*SELECT * FROM employee AS e WHERE name=Sven*

För snabb hantering av begäran och svar måste Makumba vara optimerat så att alla frågor till databasen ställs på en gång. Ovanstående resulterar i ett något annorlunda krav på **<mak:list>**, att, i de fall då **<mak:list>** är nästade, gå igenom alla barn och, för alla barn som är av typen **<mak:list>**, spara deras frågor. Därefter måste alla frågor ställas till databasen samtidigt. Slutligen itererar **<mak:list>** över svaren från databasen. För att åstadkomma detta beteende hos **<mak:list>** har Makumba utvecklat *Page Analysis*. De aspekter av **<mak:list>** som ska simuleras i en JSF-miljö är insamlingen av frågor av den typ som exemplifierats ovan, ställa dem vid samma tillfälle, och sedan iterera över resultatet.

Beteendet hos **<mak:list>** visas väl genom ett exempel. Om två **<mak:list>** är nästade ska två frågor ställas till databasen och två svar returneras. Värt att notera är att den nästade **<mak:list>**:s fråga kombineras med den övre **<mak:list>**:s fråga. Det har implikationen att resultaten i svaret till fråga två binds till svaren i fråga ett. Antag att fråga ett resulterar i svaren e1, e2, e3 ... Detta innebär att svaren i fråga två blir (e1,p1) (e1,p2), (e1,p3), (e2, p4)... Därmed skulle en iteration över resultatet inneha följande utseende:



Figur 3.1: Grupperingsexempel

<mak:value>

De frågor som **<mak:list>** ställer till den databasen generar svar i form av tabeller som lagras temporärt. Med **<mak:value>** väljs de attribut av svaren som ska skrivas ut.

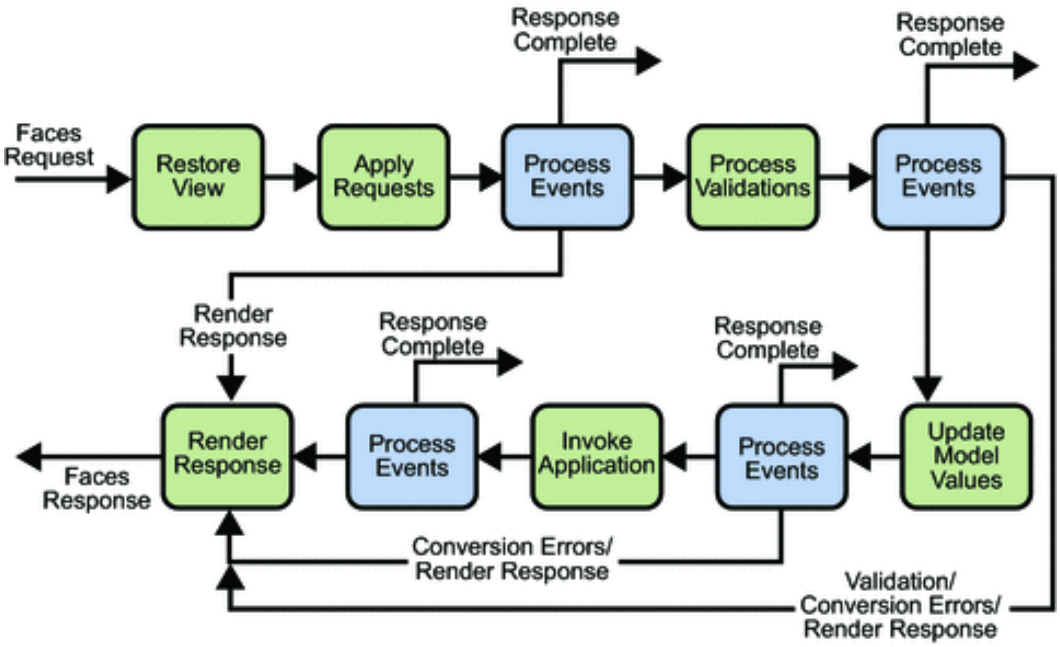
SQL-exempel:

SELECT e.name, e.position FROM employee AS e WHERE name=Sven

Ovan väljs de attribut som ska skrivas ut till name och position. Att skriva ut resultatet av denna fråga med möjlighet att välja attribut är den viktigaste aspekten av **<mak:value>**. I Makumba byggs frågor till databasen utifrån **<mak:value>**:s SELECT krav.

3.2 Analys av JavaServer Faces för Konceptbeviset

Livscykeln i JavaServer Faces



Figur 3.2: JSF livscykeln
url: http://java.sun.com/javaee/5/docs/tutorial/doc/figures/jsfIntro-lifecycle.gif

JavaServer Faces har en välutvecklad livscykel, som dock lämnar utrymme för manipulation, både för komponentutvecklare och för sidutvecklare. Det är nödvändigt för ett ramverk som vill implementera Model 2 (införande av Model-**View**-Controller (MVC) i webbapplikationsutveckling) att inneha en livscykel[3].

Den livscykel JSF använder utvecklades för hantering av diverse sysslor webbapplikationer har gemensamt, exempelvis att visa och hämta samt konvertera inkommande parametrar till lämpliga datatyper, validera dessa, samt att spara dessa parametrar eller hämta ny information. Förutom att hantera dessa kan JSF:s livscykel använda inkommande URL för att bestämma vem som utför databehandlingen, bestämma nästa vy för klienten, och utföra all programlogik som krävs för rendering av svaret[4].

Det finns tre olika typer av begäran/svar -par som resulterar i användning av JSFs livscykel. Dessa är icke-JSF begäran generar JSF svar, JSF begäran genererar icke-JSF svar samt JSF begäran genererar JSF svar. Fall ett och två är specialfall, som inte går igenom samtliga faser i livscykeln. Detta sker endast i det sista fal-

let. Livscykeln hanterar två typer av förfrågningar, *initial request* och *postback*. En *initial request* sker om sidan anropas för första gången, en *postback* om en form på sidan fyllts i[6]. En vy i JSF 2.0 utvecklas i *Facelets*, där de taggar som skapar UI-komponenter genererar en del av komponentträdet med komponenten **UIViewRoot** som bas[3]. När ett anrop görs som involverar en JSF-vy, anropas **FacesServlet**, som hanterar sidan med hjälp av **FacesContext**, som behandlas senare. JSF-implementeringen, som fyller rollen som kontroller, startar *Restore View*-fasen. Här kan två saker inträffa, beroende på vilken form av begäran som sker. Är det en *initial request* skapas en ny **View**, trädet byggs upp, därefter går processen direkt vidare till *Render Response*-fasen där trädet sparas i **FacesContext**. Om det istället är en *postback*-begäran söks det i **FacesContext** efter den **View** som stämmer överens med förfrågningen [6]. Om **View** finns hämtas den, annars får en ny **View** skapas och komponentträdet byggas upp på nytt. Bara om begäran är en *postback* går livscykeln vidare till *Apply Request Values*, annars går den till *Render Response*.

Postback sker när en form eller dylikt fyllts i på en för klienten redan renderad sida. En del av de värden bundna till UI-komponenterna, eller de objekt som håller UI-komponenternas värden. I *Apply Request Values*-fasen hämtas de nya värdena från begärans parametrar och sparas lokalt. Om en UI-komponent har dess **immediate**-attribut satt till true behandlas all information- och eventhantering associerad till denna komponent omgående.

Process Validations, den efterföljande fasen, behandlar all valideringen knuten till de UI-komponents som finns i vyn. Om ett fel uppstår läggs ett meddelande till **FacesContext**-instansen för att visas för användaren när renderingen genomförts. Efter de värden som hämtats från begärans parametrar genomgått validering uppdateras attributen på de objekt bundna till sidan. Detta sker i **Update Model Value**-fasen. I vissa fall måste en *application level*-event, såsom skicka en form eller följa en länk, behandlas, vilket görs i den näst sista fasen, *Invoke Application*[6]. I alla de faser listade ovan kan man genom manipulation av **FacesContext** få livscykeln att gå till direkt antingen till *Render Response* eller *Response Complete*, vilket avslutar livscykeln. Specifika komponenter kan även behandlas separat från livscykeln.

I den avslutande *Render Response* fasen renderas den slutgiltiga sidan som skickas till klienten och visas i dennas webbläsare. *Render Response* fasen resulterar i komponenterna renderade i valfri *markup*, som, bland annat, kan vara HTML, XML eller WML. I denna fas sparas vyn för att kunna hämtas vid ett senare tillfälle, till exempel vid en *postback* begäran. Resultatet av denna fas är det som skickas till klienten, därmed ingår det i *Render Response* fasen att kombinera statiskt och dynamiskt innehåll, samt sammanfogning av de olika dynamiska elementen till en enda respons[6].

Det som de facto sker i varje fas är att **UIViewRoot** anropar en ny metod. JSF

följer designmönstret *composite*, vilket innebär att varje komponent under **UIViewRoot** anropar sin motsvarande metod.

Utveckling av vyer i JavaServer Faces

JSF 2.0 har, för att skapa bättre funktionalitet, anammat vyteknologin *Facelets*.

En sida utvecklad med JSP kompileras till en *servlet* första gången den anropas. Därmed knöts JSP till Servletteknologin och dess livscykel, vilket i sin tur skapade problem för tidigare versioner av JSF än 2.0. *Facelets* däremot är utvecklat helt för JSF, och istället för att kompileras till en *servlet* kompileras *Facelets* till ett komponentträd[12]. En *Facelet*-sida kan vara både .xhtml eller .xml. Andra viktiga aspekter som gör *Facelets* till en bra och smidig vyutvecklingsmiljö är dess template system och dess *composite components*. *Templates* delar upp filer i två grupper, *template files*, sidor med data som återanvänds, och *template clients*, sidor med information. *Template files* definierar utseende och kontinuerligt återanvänd data (till exempel en logga) som information definierad på *template clients* anpassar sig efter dessa.

_template.xhtml

```
<title> <ui:insert name="title">Placeholder Title</ui:insert> </title>
```

_welcome.xhtml

```
<ui:composition template="_template.xhtml">

        <ui:define name="title">Welcome</ui:define>
```

Figur 3.3: Facelet exempelkod

Composite components är en annan metod för att uppmuntra kodåteranvändning. Inkluderat i JSF är även en mängd olika UI-komponenter definierade i taggbiblioteken som nämnts ovan. Dessa anropas via:

<h:..> och **<f:..>** kan nu användas för att presentera data, skapa fält och knappar, etc. Dessa taggar skapar UI-komponenter, som exempel skapar **<h:inputText>** en *HtmlInputText*-komponent. Denna komponent läggs till det komponentträd som är en representationen av vyn. Senare renderas trädet till HTML för att skickas till klientens webbläsare.

Komponenter och deras Roll i JavaServer Faces

En central del av JavaServer Faces är dess utnyttjande av designmönstret *composite*, som utnyttjas genom användning av JSF:s *User Interface Components*. Dessa är en av de två olika sorters komponenter som JSF erbjuder. UI-komponenter kan renderas till en valfri *markup*. Burns et al definierar *user interface components* som:

"... en specifik typ av komponenter som visar user interface content som kan modifieras av användaren..." [3]

I JSF-sammanhang är UI-komponenter ett viktigt begrepp, vilket refererar till en specifik **UIComponent** -klass som definierar grundläggande beteende av komponenten oavsett hur den renderas. Dock brukar termen UI-komponenter ofta användas för att beskriva ett helt set innehållandes både UI-komponenter och "hjälp"-komponenter såsom **Renderers**.[3]

JSF:s UI-komponentarkitektur utgår ifrån den abstrakta klassen **UIComponent** och dess subklass **UIComponentBase**. Därigenom ärver alla komponenter de metoder specificerade för dessa två, dock kan de även implementera olika interface, bland annat *NaimingContainer*, som anger ett *naming scope* vars syfte är att binda ihop dess barn. Förutom att JSF gör det lättare att utveckla nya komponenter genom denna arkitektur[14], erbjuder JSF ett set av färdiga komponenter för HTML-klienter, varav **<h:inputText>** är en. även dessa ärver ursprungligen från **UIComponent**. Viktigast här är dock möjligheten att producera egna, nya UI-komponenter[3].

Som förtäljs i avdelningen om livscykeln, börjar varje fas i livscykeln med att anropa en av **UIViewRoot**s metoder, vilket resulterar i en kaskad av liknande metoder, en för varje komponent i trädet. Detta innebär att utveckla komponenter sker genom att skriva över de metoder som är fördefinierade av **UIComponentBase**. En komponent utvecklad för att skriva ut en textrad, 'Hello World' till exempel, behöver skriva över metoden **encodeEnd(), encodeBegin()**, eller **encodeAll()** (de metoder som renderar det slutgiltiga *markup*). Viktigt att komma ihåg är att om komponenten som utvecklas ärver ifrån **UIComponentBase** måste metoden **getFamily()** skrivas över. För en komplett lista över de funktioner som kan skrivas över refereras till JavaServer Faces API. Det går även att utgå ifrån och ändra i källkoden för att kunna påverka färdiga komponenter såsom **<h:inputText>**, eller **<ui:repeat>** för att skapa skräddarsydda komponenter.

JSF har ett stort stöd för utvecklande av nya komponenter som inte bara sträcker sig till UI-komponenter, utan även de som kallas 'hjälp'-komponenter. Dessa består av tre olika klasser, **Renderer**, **Tag Handler**, och **Attached-Objects**. **Attached-Objects** är en samling av olika klasser som bidrar med att förbättra en UI-komponents funktionalitet, koverterare och validatorer tillhör denna kate-

```
@Override
encodeAll (){
      Response writer = new Responsewriter ()
      writer.startElement("p", null);
      writer.writeAttribute("style", "h1", null);
      writer.writeText("Hello World", null);
      writer.endElement("p");
```

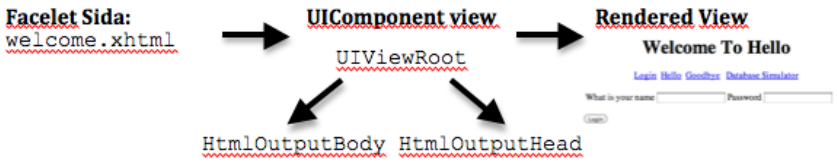
```
//Koden bredvid i HTML
//<p style="h1">
//Hello World
//</p>
```

Figur 3.4: UI-komponent kodexempel

gori. Renderare är ansvariga för att inkapsla data och en tagg-hanterare kopplar ihop UI-komponenter med renderare[3]. Det är även fullt möjligt att ändra delar av JSF:s inre programvara, **Non-UI Custom Components**, till exempel kan **Viewhandler** dekoreras för att ändra **UIViewRoot**[9].

Rendering av Sidor i JavaServer Faces

Den sida som presenteras till klienten kallas **View** inom JSF och består av ett komponentträd med UI-komponenter . När en sida anropas, byggs **View** upp med hjälp av komponentträdet vars bas är komponenten **UIViewRoot**. Komponentträdet är de facto serversidans representationen av vyn presenterad till klienten. Populeringen av trädet sker med hjälp av en FaceletFactory som bygger en *Facelet*. *Facelet*:en innehåller bygginstruktioner för vyn[13]. För att möjliggöra visning av sidan i klientens webbläsare renderas därefter trädet, tillsammans med alla statiska element, till *markup* i JSF-livscykelns sista fas, *Render Response*. Denna *markup* skickas sedan till klienten[3].



Figur 3.5: JSF sidrendering

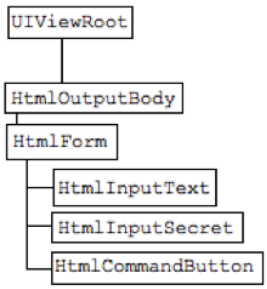
För rendering av sidan är de viktigaste faserna i JSFs livscykel *Restore View* och *Render Response* . Som nämnts tidigare finns två typer av förfrågningar som behandlas av livscykeln, *initial request* och *postback*-begärans. I *Restore View* uppdateras, om det är en *postback*-begäran, **View** utifrån den information som finns lagrad i **FacesContext**. Om det istället är en *initial request* som ska behandlas skapas en ny, tom **View** som därefter fylls. **FacesContext** är en lagringsinstans som innehåller all information om den förfrågan som bearbetas för tillfället. Varje förfrågan skapar därmed en instans av **FacesContext**, som följer med under livscykelns olika faser. Genom att manipulera denna kan information sparas, hämtas och användas

av olika komponenter och livscykelfaser. Ett exempel på detta är sparandet av en **Managed Bean** under *Render Response*, för att sedan hämta denna vid *postback*.

```
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core">
<h:body>
      <h:form>
            <table>
                  … (vanliga html taggar) …
                        <h:inputText id="userid" value="#{login.userid}"/>
                  … (vanliga html taggar) …
                        <h:inputSecret id="password" value="#{login.password}"/>
                  … (vanliga html taggar) …
      <h:commandButton value="Login" action="#{login.loginAction}"/>
```

Figur 3.6: Facelet-sida kodexempel

För att exemplifiera skapandet av en **View** och dess komponentträd ska kodexemplet ovan användas. Värt att notera är att bara de taggar med prefixen h ovan läggs till i komponentträdet, vilket är varför delar av koden har utelämnats. En mottagande webbläsare kan tolka och rendera HTML, det dynamiska innehållet (i detta fall UI-komponenterna som består av input-fälten och *commandButton*) måste därmed renderas till HTML-*markup* för att sedan visas upp av webbläsaren. Sidan skulle resultera i följande komponentträd.



Figur 3.7: Komponentträd i JSF

Viktigt här är att se vad som händer vid de tillfällen komponenter ligger innanför andra komponenter, såsom **<h:inputText>** ligger innanför **<h:form>** som ligger innanför **<h:body>**, taggarna blir nämligen barn till de taggar de ligger innanför. Trädet blir ett släktträd med **UIViewRoot** som urfader[3]. Det är komponentträdet lagrat i **FacesContext** som gör det möjligt för de UI-komponenterna i trädet att hålla reda på både sina barn och föräldrar. Detta görs genom metoderna **getChildren()** och **getParent()**, som alla UI-komponenter ärver av **UIComponentBase**[3].

Objekt i JavaServer Faces

I JSF är all domänlogik inkapslad av *Plain Old Java Objects* (POJO), som används för att frånskilja vyteknologin från logiken. En POJO som görs tillgänglig av JSF under exekvering kallas en **Managed Bean** [3]. Alla Java-klasser som har en allmän, argumentfri konstruktor kan registreras som en **Managed Bean**.

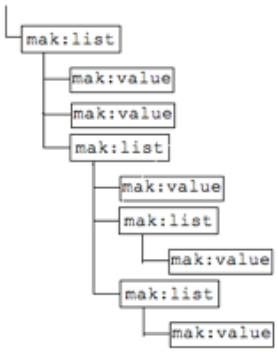
För att en **Managed Bean** ska kunna användas under exekvering måste den först deklareras till JSF. Inuti bönan ligger logik som kan nås och ändras via *ExpressionLanguage*. I en JSF-applikation är det **Managed Bean**s som håller data, till exempel via **Maps** eller **Lists**, som även kan anropas som **Managed Bean**s direkt[3]. Eftersom en **Managed Bean** är en Java-klass, innebär det att de är uppbyggda av vanlig Javakod. Det går även att binda UI-komponenters attribut till **Managed Bean**s, vilket gör det möjligt att utveckla specifika **Managed Beans** till specifika komponenter. Detta görs i xhtml sidorna.

Kapitel 4

Konceptbeviset

4.1 Implikationer av Undersökningen för Konceptbeviset

En *Facelet*-sida som utnyttjar **<mak:list>** och **<mak:value>** behövdes utvecklas. Den vy som slutligen skall presenteras för klienten är konstruerad av en *template client* som innehåller **<mak>**-taggarna samt en *template file*, som dock är frivillig. *Template client*-filen består av HTML-kod blandat med UI-komponenter, främst **<mak:list>** och **<mak:value>**. När sidan anropas tar JavaServer Faces livscykel hand om begäran, och skapar ett komponentträd med **UIViewRoot** som bas bestående av ovannämnda UI-komponenter. Denna lagras i **FacesContext**, vilket ger alla komponenter i trädet tillgång till hela trädet. Trädet renderas sedan till HTML och presenteras till klienten.



Figur 4.1: Komponentträd exempel

Det är utvecklingen av komponenterna **<mak:list>** och **<mak:value>** som är konceptbeviset, och hur de fick den önskade funktionaliteten gås igenom nedan. Dock har det visat sig finnas två sätt att gå tillväga, antingen infogas efterlängtad funktionalitet genom överskrivning av metoder, eller så kan man utgå ifrån källkoden och lägga till funktionalitet genom nya metoder och ny kod, dock utan att fullständigt skriva över existerande metoder. Eftersom önskat beteende är enbart fokuserad på rendering av information, varken inmatning från användaren eller något event sker, påverkas komponenterna bara av första och sista fasen av JSF:s livscykel. Dock måste de kunna hantera både *postback* och *initial request*. För att temporärt spara informationen inläst ifrån databasen behövs en **Managed Bean** användas.

Här kan det vara på plats att nämna den **Managed Bean** som utnyttjas för att simulera databasen. Informationen sparas med hjälp av datastrukturen **HashMap**. **HashMap**:s utnyttjar nycklar för att komma åt värden sparade i strukturen och är instanser av den abstrakta klassen **Maps**. Varje element i databasen är en instans av denna **HashMap**, och har strukturen:

(fName=Sven), (lName=Larsson), (position=floor)

Det första värdet, fName till exempel, är nyckeln. Om nyckeln stoppas in i strukturen returneras värdet associerad till nyckeln. Databasen i säg är en lista som håller flera **Maps**.

4.2 Analys av <ui:repeat>

<ui:repeat> och <mak:list>, ytlig funktionell överlapp

Det finns i JSF några olika funktioner som itererar över data, bland annat <c:forEach> och **<ui:repeat>**[13]. Den framtagna **<mak:list>** utgick ifrån **<ui:repeat>** för att ta tillvara på dess redan implementerade itereringsförmåga. **<ui:repeat>** besitter de olika attributen *value*, den data som ska itereras över, och *var*, den data som används under itereringen[13]. En jämförelse mellan Makumbas **<mak:list>** tagg och **<ui:repeat>** visar att båda har två liknande attribut, *value* och FROM, samt *var* och AS.

```
<mak:list from="employee e">
      <mak:value expr="e" />
</mak:list>
```

```
<ui:repeat value="#{database}" var="e">
      <h:outputText var="e" />
</ui:repeat>
```

Figur 4.2: <mak:list> & <ui:repeat>

Exemplet ovan visar några av de likheter som finns mellan de två. ytterligare lik-

heter inkluderar att båda kan nästas. **<ui:repeat>** innehar därmed en del av den funktionalitet som komponenten **<mak:list>** ska ha, men behöver utvecklas för att innehålla andra, bland annt möjligheten att samla in frågor från barn och ställa dessa, samt lägga till WHERE-villkor. Möjligheten att hämta ut bara en viss del av svaret, e.name till exempel, tas upp i konstruktionen av **<mak:value>**-komponenten.

<ui:repeat>:s funktionalitet

<ui:repeat> innehåller ett fåtal metoder som styr stora delar av dess funktionalitet. De metoder som är av störst vikt här är **process()** och **getDataModel()** som måste ändras för att uppnå önskad funktionalitet.

Process() anropas av andra metoder i varje livscykelfas. De metoder som anropar **process()** är specifika beroende på vilken livscykel exekveringen befinner sig i. Det som är viktigast för utvecklingen av **<ui:repeat>** till **<mak:list>** är vad som sker i denna metod. Det är här de metoder som är logiken bakom **<ui:repeat>** anropas. Det är därför av stor vikt att förstå vad som sker i metoden och när.

I **process()** kontrolleras det om **<ui:repeat>**-komponenten har några barn och om så är fallet sker en iterering över barnen. Iterering sker genom att för varje del data i en datamängd iterera över barnen en gång.

> Exempel: Om två **<ui:repeat>** är nästade, och den **<ui:repeat>** som är förälder har en datamodel av storlek tio, itereras det över barnen tio gånger. Om barnet har en datamodel av storlek fem, leder det till att för varje resultat från fadern, itereras det fram fem resultat av barnet. Detta ger oss en lista med 55 stycken rader.

När alla frågor ställts och resultaten lagrats ska den tillgängliga itereringsfunktionen i **<ui:repeat>** utnyttjas. Det behövs ingen större ändring av itereringsfunktionen för att komponenten skall fungera tillfredsställande. Den stora ändringen är ett tillägg för att för att kontrollera WHERE-villkoret. Tillägget kontrollerar alla nästade komponenters frågor och svar med varandra för att bestämma vad som skall skrivas ut.

Viktigt att poängtera är att frågeprocessen endast utförs för föräldrar**<mak:list>**. Genom att identifiera den första **<mak:list>**, går programmet igenom alla dess barn och itererar över dem. Det behövs då ingen frågeprocess för barnen då dessa blir behandlade av föräldern. Eftersom JSF erbjuder enkla metodanrop för att hämta en komponents förälder och barn, är det enkelt att kontrollera om en komponent har barn, föräldrar och av vilken typ de är.

En mindre komplicerad metod som måste justeras för att **process()** skall fungera är **getDataModel()**. Metodens uppgift är att hämta den datamängd som skall itereras över. Om man ser till hur **<mak:list>** ser ut i Makumba motsvaras denna datamängd utav det som anges i kommandot From=.... i taggen.

Databasen som efterfrågas hämtas i förälder **<mak:list>** genom inmatade attributet för FROM. För de nästade **<mak:list>** hämtas istället data från den temporärt sparade datan, och som stämmer överens med WHERE-villkoret. Det är inte en svår implementering, dock en nyckelfaktor för att itereringen i **<mak:list>** skall fungera.

4.3 Implementering av <mak:list>

Att kunna implementera ovanstående föreslagna ändringar i **<ui:repeat>**:s beteende innebär ett flertal förändringar i hur komponenten arbetar. Dock kommer konceptbevisets **<mak:list>** ha något annorlunda beteende än en färdigimplementerad **<mak:list>**.

Detta beror främst på att det inte ställs frågor till en databas, utan istället hämtas information ifrån en **Managed Bean** med namnet DBSimulator. Att ställa en fråga sker på följande vis, **<mak:list>** taggen kopplar ihop sig med DBSimulator och hämtar den data som stämmer överens med WHERE-villkoret, som testas av metoden **checkWhere()**. Därefter sparas den korrekta datan i **tmpDatabase**, en instans av **MakBackingBean**, en skräddarsydd **Managed Bean**. För **<mak:list>** att samla ihop alla barnens frågor och ställa dessa samtidigt är dock mer komplicerat. För att lösa detta problem används metoden **getChildren()**, som alla UI-komponenter innehar. Den överste mak:list taggen anropar **getChildren()** via metoden **checkChildren()**, samlar in barnens frågor och sparar dem i **tmpDatabase**. Den översta **<mak:list>** hämtar sedan dessa frågor, kontrollerar WHERE-villkoret, och sparar sedan den överensstämmande datan i **tmpDatabase**.

MakBackingBean

För att förstå hur frågor samlas in och ställs behövs förståelse för hur frågorna och svaren lagras. Detta sker i en **Managed Bean** som heter **MakBackingBean**. I **MakBackingBean** finns två datastrukturer av typen **Map**, **Map<String, List<Map» database**, och **Map<String, List<String» varFromWhere**. I **varFromWhere** sparas frågorna, som är enkelt strukturerade:

Fråga ett:

<mak:list from=#{dBSimulator} where=”name=Sven” var=info>
Fråga två:
<mak:list from=info where=”position=floor” var=info2>

Detta ger oss samma resultat som SQL-frågorna:

Fråga ett:
CREATE VIEW info AS
*(SELECT * From dBSimulator WHERE name=Sven)*
Fråga två:
*CREATE VIEW info2 AS (SELECT * From info WHERE position=floor)*

Anledningen till CREATE VIEW används är för att information sparas i **MakBackingBean** efter frågan ställts. Notera att detta skiljer sig något från hur **<mak:list>** implementerad på Makumba arbetar. Att spara en fråga görs med **var** som nyckel och en lista med FROM och WHERE som värde i ett **key:value pair**. De innehåller all information som krävs för att ställa en fråga. Viktigt att påpeka är att den sträng som står efter FROM, dBSimulator eller info till exempel, sparas under variabeln value inuti **<mak:list>**. Alla frågor kan sedan hämtas och gås igenom via metoden **getVarFromWhereKeys()** som returnerar alla nycklar i **Map**-instansen **varFromWhere**, med vilka all data i **varFromWhere** kan hämtas. Resultatet av frågorna ser ut som följande:

```
List<Map> = [ [ {”fName”=”Sven”} {”lName”=”Gustavsson”} {”position”=”floor”} ]
              [ {”fName”=”Sven”} {”lName”=”Larsson”} {”position”=”floor”} ] ]
```

Figur 4.3: Hur data sparas

Denna information sparas i **MakBackingBean.database**, även här med **var** som nyckel. Detta förenklar senare iterering över resultatet, och gör det enkelt för barnen att hämta resultatet av just sin fråga.

CheckChildren - metoden som samlar in barnens frågor

För att kunna spara alla barnens frågor måste **<mak:list>** kunna gå igenom alla dess barn och spara alla deras frågor i **MakBackingBean.varFromWhere**. Den metod som utför detta är **checkChildren()**.

Möjligheten för barnet att anropa sin **checkChildren()**-metod är helt beroende av att **<mak:list>** har kännedom om sina barn. Detta är kärnan av **checkChildren()**, och möjliggörs av metoden **getChildren()**. Resultat av detta metodanrop

```
private void checkChildren() {
    if (this.value == null) {                                   //Gör en
        ValueExpression ve =                                    valueExpression till en
              this.getValueExpression("from");                  sträng
        if (ve != null) {                                       annars kan den inte
                                                                sparas
            this.value=ve.getExpressionString();
        }
    }
    this.addFromWhere(this.var, this.value, this.where);

    List children = this.getChildren();                         //lägg till egna frågan
    if (this.getChildCount() > 0){                              //Hämta alla barn (inte
        for (int i = 0; i < children.size(); i++) {             barnbarn) till mak:list
          UIComponent kid = (UIComponent) children.get(i);
          if (kid instanceof ListTag) {                         //bara mak:list barnen
            ListTag rkid = (ListTag) kid;                       ska gås igenom
                                                                //Nödvändig casting
            rkid.notTop = true;

            if (rkid.value == null) {                           //Casta barnen till
              rkid.value =                                      ListTag, annars antas det
              (String) rkid.getAttributes().get("from");        de inte besitter
            }                                                   variablerna var, value
            this.addFromWhere                                   och where
              (rkid.var, (String) rkid.value, rkid.where);      //Koppla ihop barnen
                                                                med MakBackingBean
            rkid.tmpDatabase = this.tmpDatabase;                //Anropa barnenes
            rkid.checkChildren();                               checkChildren()

        }
    }
  }
}
```

Figur 4.4: checkChildren() kodexempel

är en lista med alla barn, dock inte barnbarn. Därmed behöver **checkChildren()** se till att de barn som är instanser av **<mak:list>** anropar sina **checkChildren()**-metoder. Därefter läggs de för varje barn specifika variablerna **var, from, where** som används för att ställa frågorna, till **tmpDatabase.varFromWhere**. Samtidigt sätts barnens **nonTop**-variabel, vars syfte är att hålla koll på om **<mak:list>** är nästad, till true.

AskQuestions - metoden som ställer frågorna och lagrar datan

Metoden **checkChildren()** samlar in och lagrar all den nödvändiga informationen som krävs för att ställa frågorna. Därmed är allt klart för den översta **<mak:list>** att ställa frågorna.

AskQuestions() kan delas upp i tre delar. Först ställs den översta **<mak:list>**s fråga, därefter hämtas barnens frågor från **tmpDatabase** och slutligen ställs barnens frågor. Som beskrivet ovan, sparas frågorna i en datastruktur **Map** med följande utseende:

Map<String, List<String,String» //**varFromWhere**<*var, fromWhere<from,where»*

För att få tillgång till själva frågan, from och where, används nyckeln var. Alla nyck-

```
private void askQuestions() {

      List val = (List) getAttributes().get("from");          //Hämtar databasen
      List newVal = checkWhereStatement(val, this.where);     //Kollar where vilkoret
      this.tmpDatabase.addToDatabase(var, newVal);            //Lägger till resultatet till en
                                                              MakBackingBean
      Set<String> questions =                                 //Hämta alla nycklar och gör
          this.tmpDatabase.getVarfromwhereKeys();             om dem till array
      Object[] qArray = questions.toArray();

      for (int j = qArray.length - 2; j >= 0; j--) {
                                                              //Hämta var för en fråga
          String tmpVar = (String) qArray[j];                 //Hämta From och Where
          List fromWhere =
                tmpDatabase.varFromWhere.get(tmpVar);         //Hämta från databasen det
                                                              tidigare sparade resultat
          val = this.tmpDatabase.database.get                 //Kolla where vilkoret
                ((String) fromWhere.get(0));                  //lägg till i databasen
          newVal = checkWhereStatement
                (val, (String) fromWhere.get(1);
          this.tmpDatabase.addToDatabase(tmpVar, newVal);
      }
  }
```

Figur 4.5: askQuestions() kodexempel

lar associerade med en **Map** kan hämtas via metoden getKeySet(). Denna anropas via **tmpDatabase.getVarfromwhereKeys()**, en metod definierad i **MakBackingBean**.

```
public Set<String> getVarfromwhereKeys() {   //En etod som returnerar alla nycklar
    return varFromWhere.keySet();            //associerade med varFromWhere (en Map)
}
```

Figur 4.6: getVarFromWhere() kodexempel

Självaste frågeställningsprocessen är relativt simpel, först hämtas en samling data (antingen från DBSimulator eller från resultatet av en tidigare fråga), därefter väljs den data som stämmer överens med WHERE-villkoret (görs i **checkWhere()** metoden), och slutligen läggs den till i **tmpDatabase.database**.

CheckWhere - metoden som kollar WHERE-villkoret

WHERE-villkoret kontrolleras med hjälp av datatypen **Map**:s inbyggda beteende. Ett WHERE-villkor har ett ”nyckel-värde”-utseende, till exempel fName=Sven. Detta blir, efter parsing, [fName,Sven]. Varje element i databasen är en **Map**, därmed kan fName stoppas in som nyckel, och sedan kan det värde som returneras jämföras med Sven. Är de samma ska elementet sparas. På detta vis sorteras alla element som inte uppfyller WHERE-villkoret bort.

```
private List checkWhereStatement(List val,
  String where) {                                  //För att förhindra en
    if (where != null) {                           nullPointExceptionError
        String[] whereStr = parseWhere(where);

        List tempList = new ArrayList<Map>();
        for (int i = 0; i < val.size(); i++) {
                Map tempVal = (Map) val.get(i);    //
                if (whereStr[1].equals
                   (tempVal.get(whereStr[0]))) {   // Kolla om name="Sven",
                    tempList.add(tempVal);         position="floor", etc
                }                                  // i sådana fall lägg till

        }
        val = tempList;
    }
    return val;
}
```

Figur 4.7: checkWhere() kodexempel

4.4 Implementering av <mak:value>

Till skillnad från **<mak:list>** skrevs **<mak:value>** egenhändigt från början, då denna är en mycket simplare komponent vars enda uppgift är att hämta data från sin förälder, en **<mak:list>**, och presentera denna för klienten.

För att få tillgång till all funktionalitet som behövs för en komponent ärver **<mak:value>** från **UIComponentBase**. Det ger oss möjligheter att utnyttja flera inbyggda metoder för att komponenten skall arbeta på ett önskvärt sätt.

I **<mak:value>** visas styrkan i sättet som JSF arbetar på. Genom den tidigare beskrivna metoden **getParent()** kan komponenter enkelt och med få rader kod hämta information från dess förälder. När **<mak:value>** blir kallad hämtar den först in attributvärdet från xhtml-sidan, tar ut den relevanta informationen genom en parsmetod och lagrar denna. Från **<mak:list>**-föräldern, som har hämtats genom **getParent()**-metoden, hämtas svaret på den fråga som **<mak:value>** skall visa sitt resultat ifrån. **<mak:value>** hämtar sedan ut rätt information från svaret som **<mak:list>** presenterar och skriver ut detta för rendering till klienten. Den kod som genomför detta visas nedan.

Den variant av **<mak:value>** som skrevs för detta konceptbevis är väldigt enkel och arbetar endast mot **<mak:list>** och den modellerade databas som används. Det är dock enkelt att utöka dess funktionalitet och anpassa den till andra taggar och databaser.

```
@Override                                                    //Metoden som skrivs över
public void encodeAll(FacesContext context) throws
     IOException {                                           //Hämtar attributet med namnet
     String expr = (String) getAttributes().get("expr");     "expr"
     this.expression = parseExpression(expr)[1];             //egen konstruerad parsmetod
     this.setHoldingBean();                                  //metod som anropar getParent()
     mp = holdingBean.iteratingOver;                         och hämtar det Map elementet
                                                             som ska skrivas ut
     ResponseWriter writer = context.getResponseWriter(); //Renderar komponenten i HTML
     if (mp != null) {
       if (this.expression.equals("*")) {                    //Väljer ut de värden som ska
          Set keys = holdingBean.getKeys();                  renderas, i detta fall alla
          Object[] qArray = keys.toArray();
          for (int j = qArray.length - 1; j >= 0; j--){
            writer.writeText((String) mp.get
                   (qArray[j]) + "  ", null);
          }
       }
       else if (mp.get(this.expression) != null) {           //Skriv ut de värden som
         writer.writeText(mp.get(this.expression), null); definieras av nycklarna
       }
     }
}
```

Figur 4.8: <mak:value> kodexempel

Kapitel 5

Konklusion

5.1 Diskussion av Konceptbeviset och Resultat

Det konceptbevis som beskrivits ovan har haft som syfte att visa hur och om en återimplementering av Makumba på JavaServer Faces-teknologin är möjlig. Med avsikt valdes två Makumba-taggar ut för implementering vars funktionalitet ställer krav på de delar i teknologin som tidigare har varit problematiska att hantera och implementera. På den äldre teknologin krävdes en avancerad mekanism, *Page Analysis*, för att taggar bland annat skulle ha möjlighet att nästas, men även för att uppnå önskad prestanda. En potentiell lösning på det problemet redovisas ovan i konceptbeviset där JavaServer Faces livscykel och sätt att hantera webbsidor utnyttjas.

De komponenter som har skapats baserar sig antingen på en redan existerande komponent eller ärver beteende från en standardkomponent. Att skapa en ny komponent baserad på en redan tidigare implementerad standardkomponent med liknande beteende gav en bra utgångspunkt för utvecklingen. Genom att analysera den existerande standardkomponentens funktion samt dess källkod förkortades designen av komponenten, eftersom mycket av det önskade beteendet redan var färdigskrivet och implementerat.

5.2 Implikationer av Konceptbeviset: Page Analysis kontra FacesContext

En av de viktigaste delarna av Makumba är dess mekanism för att analysera en JSP-sida. Mekanismen delger en Makumba-tagg information om webbsidan som den

annars inte skulle inneha. Genom att en tagg får information om hela sidans innehåll och uppbyggnad kan taggens exekvering optimeras och då leda till en effektivare körning av applikationen.

I JSPs stöd för taggbibliotek tillåts det att köras skräddarsydd kod vid specifika tillfällen enligt specifikationen. En viktig aspekt som JSP stödjer är att en tagg vet om vem dess förälder är, och kan på så sätt hämta viktig och nödvändig information. Något som JSP inte stödjer är däremot den viktiga funktionaliteten att ha kännedom om taggens barn eller var på webbsidan som taggen finns. Det motiverar väl användningen av en *Page Analysis*-mekanism som stödjer detta, eftersom flera funktioner hos många Makumba-taggar behöver denna information inte bara för att exekveras effektivt utan även för att överhuvudtaget fungera.

Den nya plattformen JavaServer Faces ger dock stöd för komponenter att ha kännedom om både sina föräldrar och barn samt resterande komponenter på webbsidan. Detta görs möjligt genom **FacesContext** där baskomponenten **UIViewRoot** lagras från och med den första fasen i JavaServer Faces livscykel.

Tillgången till **FacesContext** vid det tidiga stadiet i livscykeln ger utvecklaren stora möjligheter att använda sig av. I det konceptbevis som har presenterats ovan har detta utnyttjats på enklast möjliga sätt efter de krav som ställs på applikationen. Vid en full integrering av Makumba på JavaServer Faces -plattformen kan en mekanism liknande Makumbas *Pages Analysis* vara nödvändig. En sådan mekanism kan då dra nytta utav att JavaServer Faces tillhandahåller **FacesContext**, då **FacesContext** erbjuder för *Page Analysis* flera färdiga metoder att använda.

5.3 Framtida Arbete

Avhandlingen har visat hur en **<ui:repeat>** kan påverkas för att simulera **<mak:list>**, men för utvecklingen av Makumba för JSF finns mycket kvar att göra. Det finns många fler aspekter av **<mak:list>** kvar att implementera. Även de möjligheter som existerar för att manipulera JSFs inre aspekter behöver utforskas.

Andra lösningar

Som nämndes i avsnittet om UI-komponenter finns det många möjligheter att påverka JSF förutom att bygga UI-komponenter, bland annat kan man dekorera flera av de komponenter som skapar trädet, lägga till ett nytt steg i livscykeln eller förändra hur taggar hanteras. I den lösningen av problemet som presenterats här har den översta **<mak:list>** tagit hand om skapandet och insamling av frågor. Denna lösning fungerar mycket väl i de lägen då det bara finns en topp **<mak:list>** på si-

dan, men skulle det finnas flera syskon **<mak:list>** utan gemensam **<mak:list>** förälder skulle det kräva flera olika uppkopplingar till databasen. Det är därför viktigt att titta på andra sätt att lösa problemet.

Att utnyttja möjligheten att påverka **UIViewRoot** och dess **FacesContext** container skulle vara ytterligare ett sätt att lösa detta problem. **UIViewRoot** som komponent har därmed kännedom om sina barn vilket visas i konceptbeviset. **UIViewRoot** skulle alltså kunna skräddarsys så att den hanterar insamling och sparande av frågor. I dagsläget finns det väldigt lite information om hur detta skulle göras, men att dekorera **View-handler**, den komponenten som skapar **UIViewRoot**, är en potentiell lösning. Data skulle i detta fall kunna sparas via utnyttjande av **FacesContext getSessionMap()**. I den *Page Analysis* Makumba använder sig av för tillfället läggs det till en ny fas i livscykeln. Att göra samma eller liknande i JSF, vilket är fullt möjligt, är ytterligare en potentiell lösning på problemet.

Full integrering av Makumba

När det gäller den praktiska fullskaliga implementeringen av Makumba på JSF kvarstår mycket jobb. En integral del är hur JSF konverserar och kopplar upp sig mot databaser, något som denna avhandling inte har behandlat. En stor del av implementeringen av Makumba på JSF är utveckling av Makumba-komponenter till JSF. Med JSF kommer en stor mängd färdiga komponenter, vilken kan skapa överlapp med Makumba. Avhandlingen visade hur arbetet att utveckla komponenter blir lättare om man utgår ifrån redan färdiga komponenter. Därmed behövs en grundlig analys av Makumba-taggarna och JSF-komponenterna. Resultatet av denna analys bör peka ut vilka komponenter som ska ligga till grund för framtida Makumba-komponenterna. Hur utveckling av komponenterna ska ske behöver också en närgången analys. I avhandlingen adderades kod till **<ui:repeat>**:s källkod för att uppnå önskat beteendet. Det finns andra sätt att utföra detta, möjligen kan man låta **<mak:list>** ärva **<ui:repeat>**, lägga till de nödvändiga metoderna samt skriva över **getDataModel()** och **process()**.

Kvarstående att utveckla för <mak:list> och <mak:value>

Det finns flera aspekter av **<mak:list>** som inte implementerats här, bland annat GROUP BY samt ORDER BY. Ett potentiellt sätt att lösa ORDER BY är att skriva en metod i **MakBackingBean** som hämtar alla värden, sorterar dem med hjälp av en **ArrayList**, och stoppar tillbaka dem i **MakBackingBean.database**. En aspekt av den utvecklade **<mak:list>** som definitivt behöver närmare undersökning är *ExpressionLanguage*. För att närmare efterlikna originaltaggen behöver **<mak:list from = {dBSimulator.database}>** kunna skrivas **<mak:list**

from=dBSimulator.database> även i JSF och på **<mak:value>** behöver *var* kunna tillämpas.

Dessutom har den utvecklade komponenten **<mak:list>** inte exakt samma beteende som taggen implementerade på JSP. Att utelämna beteende är motiverat eftersom det är *Page Analysis* förmåga att ge information om komponenters barn som varit huvudfunktionaliteten att implementera. Detta innebär i sin tur att än mer funktionalitet finns kvar att infoga i **<mak:list>**. Den funktionalitet som omnämns ovan anammades ej för att skapa förenklade omständigheter för implementeringen av de svårare aspekterna.

Det som främst behöver göras är genom att använda samma metod som i konceptbeviset, utnyttja metoden **getChildren()**, för att gå in i **<mak:value>**-barnen och hämta deras SELECT-villkor (SELECT e.name ...) samtidigt som **<mak:list>**-barnens frågor samlas in. Ett potentiellt sett skulle vara att ändra frågeställningsprocessen genom att utgå ifrån **<mak:value>**:s SELECT-krav istället för att utgå ifrån **<mak:list>**:s FROM för att bygga frågorna, vilket görs idag. En av anledningarna till att ej implementera detta beteende var att det skulle vara svårt att visa och testa detta utan faktisk uppkoppling till en databas. Dock är detta något som behöver lösas innan fler Makumba-taggar, så som**<mak:edit>**, kan implementeras på JSF eftersom **<mak:edit>** inte använder SELECT utan UPDATE. Under utveckling av konceptbeviset har ingen skillnad mellan JSP och JSF upptäckts som skulle förhindra användningen av Makumbas nuvarande lösning på detta problem.

5.4 Anledningar till Implementering

Det finns flera anledningar för Makumba att implementeras ovanpå JavaServer Faces. Förutom att Makumba anpassar sig till en standard, vilket gör det lättare för utvecklare att kombinera Makumba med andra ramverk, bidrar JSF med kraftfull funktionalitet. De delar av JSF som togs upp i avhandlings tredje del, JSF 2.0, kommer i följande stycke användas för att visa den potentiella vinsten av implementeringen.

Mängden komponenter, färdiga taggar, *templating* och enkla metoder för återanvändning av kod gör *Facelets* till ett mycket kraftfullare system än att bara förlita sig på HTML, så som användare av Makumba uppmuntras att göra. En av huvudanledningarna att implementera Makumba på JSF torde vara att komma åt just denna funktionalitet. Taggarna ser ut som traditionella JSTL-taggar, vilket förkortar inlärningskurvan för någon som använt sig av någon av dessa teknologier tidigare. Det går dock fortfarande att skriva i HTML, om så önskas.

Livscykeln är ytterligare en anledning att implementera Makumba på JSF. Om Ma-

kumba implementeras utan ändring av livscykeln finns en färdig, fungerande livscykel som inte behöver dekoreras för utveckling av webbapplikationer för Makumba. Dock finns det gott om möjligheter att utföra detta.

JavaServer Faces är en utvecklingsteknologi som har en hög inlärningströskel, vilket varit en av lärdomarna från detta projekt, som inte förminskas av bristen av information angående JSF 2.0. Viss funktionalitet associerad med JSF 2.0 har minskat denna tröskel. Makumba ovanpå JSF skulle kunna minska denna tröskel ytterligare, men öka funktionaliteten för erfarna användare av Makumba med Javakunskap, utan att samtidigt försvåra användningen av Makumba. Detta måste vara en av grundmålen för implementeringen av Makumba på JavaServer Faces, att genomföra en implementering som ej begränsar bakomliggande teknologin men möjliggör användning av samma teknologi utan kunskap om dess funktion.

Litteraturförteckning

[1] adobe.com. The evolution of jsp. http://www.adobe.com/devnet/server$_{a}rchive/articles/evolution_{o}f_{j}sp.html$, 32010.

[2] Mark Andrews. Story of a servlet: An instant tutorial. http://java.sun.com/products/servlet/articles/tutorial/index.html, 3 2010.

[3] E. Burns and C. Schalk. *JavaServer Faces 2.0: The Complete Reference. 1st ed.* The McGraw-Hill Companies, The McGraw-Hill Companies., 2010.

[4] Damodar Chetty. Jsf lifecycle. http://softwareengineeringsolutions.com/thoughts/frameworks/JSF-lifecycle.htm, 2 2010.

[5] Rudolf Mayer Cristian Bogdan. Makumba: the role of the technology for the sustainability of amateur programming practice and community. Communities and Technologies:Proceedings of the fourth international conference on Communities and technologies, 2009.

[6] Armstrong et al. The life cycle of a javaserver faces page. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSFIntro10.html, 2 2010.

[7] Gamma et al. Design patterns: Elements of reusable object-oriented software. Technical report, Reading, MA, USA: Addison-Wesley., 1994.

[8] Ventura et al. Jclec: a java framework for evolutionary computation. Technical report, Department of Computer Science and Numerical Analysis, University of Córdoba, 2007.

[9] Wojtowski. Fritmarkiewicz, Sakowicz. Easily extendible framework for computational purposes based on javaserver faces. Technical report, Department of Microelectronics Computer Science, Technical University of Lodz, 2006.

[10] M. Gagné, C. Parizeau. Genericity in evolutionary computation software tools: Principles and case-study. Technical report, Département de Génie Èlectrique et de Génie Informatique, Université Laval, 2005.

[11] Jeremy Grelle. Introduction to spring faces part 1. http://www.jsfcentral.com/articles/intro$_spring_faces_1.html, 92008.

[12] Richard Hightower. Facelets fits jsf like a glove. http://www.ibm.com/developerworks/java/library/j-facelets/, 4 2010.

[13] Jacob Hookom. Facelets - javaserver faces view definition framework. https://facelets.dev.java.net/nonav/docs/dev/docbook.htmltemplate-repeat, 3 2010.

[14] Anna Maria Jankowska and Andrzej Dabkowski. Content adaptation tag library – an approach for user interface adaptation for different devices. Technical report, Chair of Business Informatics, Frankfurt, n.a.

[15] Mark Kolb. A jstl primer. http://www.ibm.com/developerworks/java/library/jjstl0211.html, 2 2003.

[16] Qusay H. Mahmoud. Developing web applications with javaserver faces. http://java.sun.com/developer/technicalArticles/GUI/JavaServerFaces/, 8 2004.

[17] n.a. Javabeans. http://java.sun.com/docs/books/tutorial/javabeans/index.html, last visited: 2010-04-02.

[18] Andy Schwartz. What's new in jsf 2? http://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/, 7 2009.