Web Publishing Framework in Cocoon

Examensarbete 20p i datavetenskap av Staffan Kvisth

Institutionen för Datavetenskap, Umeå Universitet

Examinator, Umeå Universitet: Per Lindström
Intern handledare, Umeå Universitet: Jan-Erik Moström
Extern handledare, Bytbil.com: Hans Svedåker

Abstract

Programming web applications using common frameworks like JSP, PHP and ASP etc. requires you to adopt a page centric design for your application's flow of control logic. Your application will take the form of a state machine where the application logic will be spread out across different pages (where pages roughly correspond to the nodes in the state machine.) This can be a problem when the application grows, as the application flow of control logic will be more and more hidden among code to handle transitions between the different states in the application. Looking at the code, it will not be immediately obvious for an uninitiated person to see what the application does.

It has been shown by others (see ref [3] and [4]) that the use of *continuations* can help keeping the application flow of control logic together into one linear program. Continuations are the encapsulation of the concept of partially evaluated functions. When an operation in your application results in a web page being sent to the client web browser the application program state is stored in a continuation object. The continuation object can later be re-invoked when the user submits a response to the web page, at which time the application is resumed in exactly the same state it was in when the web page was sent to the client. The application can then examine the response from the user and take appropriate action. To the programmer of the application flow of control logic, it looks just like that the page submission/response had been a blocking call to an input function. This way, using continuations, the application flow of control code can be written in a traditional linear fashion, in one single source file. This as opposed to the flow of control code being distributed on several separate web pages as is common for applications written in JSP, PHP and ASP.

It is argued that future web application languages should support the concept of continuations in order to allow web applications to be written in a linear style and thereby make them easier to build and maintain.

This thesis takes a closer look at a web application framework named Apache Cocoon [6] that has support for using continuations. It is shown by example how linear style programming in Cocoon has a number of benefits and gives cleaner and more correct code than would have been the case had we written the examples using the page centric approach.

Part of this thesis has been implementing an example real-world web application called WPFC using the Cocoon framework. The example application shows that the necessary features for successfully implementing a real-life web application are present in Cocoon and makes it a viable alternative or supplement to regular web application frameworks.

With features like Java J2EE compatibility and adherence to the MVC (Model View Controller) design pattern, the thesis concludes that Cocoon is an excellent platform for overall web application development.

Sammanfattning

Att bygga en webbapplikation med hjälp av de idag vanligt förekommande ramverken JSP, PHP, ASP etc., innebär att man tvingas ta explicit hänsyn till applikationens sid-orienterade struktur i koden för applikationens flödeslogik. Applikationen tar formen av en tillståndsmaskin där koden för flödeslogiken ligger utspridd i de olika webbsidorna. Man kan tänka sig att sidorna utgör tillståndsmaskinens noder och länkarna mellan sidorna utgör dess övergångar (maskinens bågar).

I takt med att webbapplikationen växer (och därmed antalet noder i tillståndsmaskinen) kommer koden för flödeslogiken att bli allt svårare att följa eftersom varje sida måste innehålla kod för att hantera ett växande antal tillstånd. Hanteringen av applikationens olika tillstånd måste explicit kodas in i programmet. Detta leder till att det blir svårt att utifrån koden följa programflödet i applikationen eftersom flödeslogiken bäddas in i kod för tillståndshantering.

För att hålla ihop koden för flödeslogik och göra tillståndshanteringen implicit kan man använda sig av *continuations*. Detta har visats av bl.a. Queinnec, ref [3] och [4]. En continuation är ett objekt som representerar en partiellt evaluerad funktion. Genom att använda continuations kan flödeslogiken i webbapplikationer skrivas som ett traditionellt linjärt program, där mekanismen att sända ett formulär (en webbsida) och vänta på svar från användaren ser ut som ett blockerande anrop till en inmatningsfunktion.

Jag visar i detta arbete hur ett webbapplikationsramverk som kallas Cocoon [6] kan användas för att med stöd av continuations bygga webbapplikationer där flödeslogiken är skriven på linjärt sätt och där tillståndshanteringen är implicit. Rapporten visar hur linjär programmering i Cocoon ger renare och mer korrekt kod som är enklare att följa och förstå. Detta jämfört med hur koden för samma applikation skulle se ut om den skrivits i ett sidorienterat ramverk utan stöd för continuations (i detta fall JSP.)

En del av detta arbete har varit att i ramverket Cocoon implementera en prototyp av en verklig webbapplikation. Prototypapplikationen visar att Cocoon har de nödvändiga egenskaperna som krävs av ett ramverk för att det skall vara praktiskt möjligt att bygga större webbapplikationer med det.

Innehållsförteckning

1 Inledning

1.1 Bakgrund och problembeskrivning

Att bygga webbapplikationer med hjälp av vanligt förekommande ramverk såsom JSP, PHP och ASP kräver att man anpassar systemets struktur utifrån de webbsidor som ingår i applikationen. Webbsidorna blir systemets grundkomponenter och koden för att hantera flödet i applikationen blir utspridd över dessa sidor snarare än att vara samlad i en överskådlig modul. Ens applikation tar formen av en slags tillståndsmaskin där noderna representeras av webbsidor och bågarna utgör länkarna mellan webbsidorna. Applikationer som byggs på detta sätt sägs vara *sid-orienterade* (eng. *page centric*, se ref [4])

I stora applikationer kan denna tillståndsgraf bli omfattande, och eftersom koden för att hantera övergångarna mellan de olika tillstånden i systemet är utspridd över webbsidorna (noderna) kan det vara svårt att få en samlad överblick över systemet.

Situationen blir än mer komplex på grund av att systemets användare kan bläddra fram och tillbaks mellan sidorna på ett oförutsägbart sätt (till exempel med hjälp av webbläsarens framåt/bakåtknappar och history-lista). Utvecklaren måste ta hänsyn till att användare kan hoppa godtyckligt mellan de olika sidorna i systemet; det är inte säkert att användare alltid följer de övergångar (bågarna i tillståndsmaskinen) som utvecklaren tänkt sig. I värsta tänkbara fall måste programmeraren skriva kod för att hantera alla möjliga tillstånd i varje sida (nod) som ingår i systemet.

I applikationer som byggs sidorienterade uppstår situationen att användaren, snarare än utvecklaren, styr programflödet i applikationen. Detta fenomen brukar kallas ”inversion of control” (se ref [4]). Eftersom denna ”inversion of control” är svår att handskas med har en lösning som bygger på användandet av *continuations* föreslagits (se ref. [4] och [5]). En continuation är ett objekt som representerar en partiellt evaluerad funktion, eller ”resten av programkörningen”.

Genom att använda continuations kan koden som hanterar programflödet skrivas på *linjärt sätt*. Dvs. på ett sätt som om mekanismen att sända ett formulär till användarens webbläsare och sedan vänta på användarens svar, är ett anrop till en blockerande input-funktion. (En blockerande funktion är en funktion som stoppar allt övrigt programflöde i applikationen tills hela funktionen exekverat klart.) Notera dock att det handlar om ett slags simulerad blockering. Webbapplikationen som helhet blockeras inte, det är bara programkörningen för en enskild klient som ser ut att blockeras.

Continuations gör det möjligt att skriva koden som hanterar programflödet i applikationen på traditionellt *linjärt* sätt. Med linjär kod menas kod där alla instruktioner och funktionsanrop är blockerande. Dvs. programmeraren kan vara säker på att alla tidigare instruktioner och anrop i programmet är helt slutförda innan programmet fortsätter med nästkommande instruktion. På detta sätt kan koden för programflödet i applikationen hållas ihop i en enhet och behöver inte distribueras till varje enskild sida.

1.2 Målbeskrivning

Ett syfte med detta examensarbete är att visa hur ett ramverk med stöd för continuations som kallas Apache Cocoon [6] kan förenkla koden för programflödet i en webbapplikation. Detta jämfört med koden för en motsvarande applikation implementerad utan användning av

continuations. Apache Cocoon är ett ramverk för att bygga webbapplikationer i Java och jag har utnyttjat det i examensarbetet för att skriva exempel som åskådliggör de kodmässiga skillnaderna mellan program skrivna i Cocoon jämfört med program skrivna för det tillika Javabaserade ramverket JSP. Exemplen är skrivna på sådant sätt att continuations används i de versioner av exemplen som är byggda för Cocoon, medan JSP-versionen av respektive exempel inte använder continuations. Dvs. Cocoon-exemplen är skrivna på linjärt sätt, medan JSP-exemplen är sidorienterade.

Som en del i examensarbetet har jag implementerat en prototyp av en verklig webbapplikation som jag kallar WPFC (Web Publishing Framework in Cocoon). Denna applikation utvecklades i samarbete med ett företag som heter Bytbil.com. Bytbil.com har i dagsläget ett webbaserat system där bilhandlare runt om i landet kan lägga in annonser på bilar dom har till försäljning. Allmänheten kan sedan gå in på www.bytbil.com och söka bland bilarna. WPFC är en prototypversion skriven i Cocoon av Bytbils befintliga system. Syftet med denna prototyp är att praktiskt visa att Cocoon inte bara är teori, utan att det faktiskt är möjligt att bygga stora webbapplikationer med Cocoon som komplement eller ersättare till JSP.

Under arbetet med att implementera WPFC tydliggjordes vikten av att utforma applikationen enligt MVC-designmönstret (Model-View-Controller, se ref. [9]) för att åstadkomma en bra uppdelning mellan den grafiska representationen av applikationen och flödeslogiken respektive layoutlogiken. Ett av huvudkriterierna för Bytbil var att denna uppdelning skulle vara tydlig och väldefinierad i prototypapplikationen. Av den anledningen tas teorin bakom MVC-designmönstret upp i detta dokument samt en beskrivning av hur MVC implementerats i Cocoon och hur det används i WPFC.

Bytbil ansåg att en viktig uppgift för WPFC-prototypen var att den skulle ha en funktion som gjorde det möjligt att relativt enkelt kunna generera olika typer av dokument och sidor utifrån samma grunddata och utan att programändringar behövde göras i applikationen. Speciellt ville man att WPFC skulle kunna generera PDF-filer utifrån samma kod och data som används för att generera HTML-sidorna i systemet. Man ville också kunna generera sidor i olika språk utan att programändringar behövde göras. I WPFC har jag visat hur detta kan åstadkommas genom att använda Cocoons dokumentpubliceringsmekanism. Av den anledningen beskrivs Cocoons XML/XSLT-pipelinebaserade dokumentpubliceringsmekanism samt hur den hanterar flerspråksstöd.

1.3 Beskrivning av Cocoon

Innan vi tittar närmare på hur användandet av continuations och MVC påverkar implementationen av webbapplikationer är det på sin plats att ge en översiktlig bild av hur Cocoon är uppbyggt och fungerar. Cocoon är ett ganska stort system (över 300MB för en fullständig installation!), men det är inte uppbyggt som en monolit bestående av miljontals rader kod. Cocoon är istället ett hopplock av en mängd andra Apacheprojekt och komponenter. Dessa omnämns i Cocoon som ”blocks”. Det finns hundratals block i Cocoon, och att gå igenom alla skulle ta för stor plats för att göras här. Jag kommer bara att beskriva ett urval block som jag anser vara dom viktigaste, nämligen: *Flowscript*, *XSP* och *velocity* samt själva grundkomponenten i Cocoon som hanterar Cocoon *pipelines*.

1.3.1 Pipelines

En applikation i Cocoon består i grunden alltid av ett antal pipelines vars uppgift är att bearbeta XML-data på olika sätt. En pipeline startar alltid med en *generator*-komponent som genererar ett XML-dokument på något sätt (t.ex. genom att läsa in dokumentet från en fil). Detta XML

dokument skickas sedan genom pipelinens olika steg i tur och ordning där det bearbetas av olika *transformer*-komponenter. En transformer kan lägga till, ta bort, eller förändra XML-dokumentet på olika sätt. Till exempel kan man tänka sig en transformer som ersätter vissa XML-taggar med data som den hämtar från en databas. I det sista steget i pipelinen passerar XML-dokumentet en så kallad serializer-komponent. Serializern bygger ett nytt dokument i ett format som en klient till applikationen kan förstå utifrån det XML-dokument som skickats genom pipelinen. Oftast är klienten till applikationen en webbläsare som visar HTML-sidor, och serializerns uppgift är då att bygga ett HTML-dokument utifrån XML-dokumentet och sända tillbaks HTML-dokumentet via HTTP till klienten.

För att sy ihop dessa pipelines till en applikation definieras en mappning mellan inkommande URL-adresser och vilken pipeline som skall exekveras när respektive URL-adress anges av användaren. Denna mappning definieras i en textfil som kallas `sitemap.xmap`.

Cocoon är tänkt att fungera som en plugin i en befintlig webbserver och Cocoons grunduppgift är att analysera inkommande URL-anrop till webbservern, matcha anropet mot de URL-pipeline-mappningar som anges i `sitemap.xmap` och starta den pipeline som matchar. De generator-, transformer- och serializer-komponenter som definierar varje pipeline är vanliga Java-objekt (som implementerar vissa av Cocoon definierade Java-interface) som exekveras av Cocoon i tur och ordning.

1.3.2 Flowscript

Blocket ”Flowscript” används för att ge stöd för continuations i Cocoon. Eftersom Java som standard inte har stöd för continuations har man importerat komponenten Mozilla Rhino [11] i Cocoon och döpt denna till Flowscript. Mozilla Rhino är ett projekt för att implementera språket JavaScript i Java, och Mozilla Rhino har stöd för continuations.

Man kan definiera en pipeline så att den istället för att skapa ett resultatdokument genom att anropa generator-, transformer- och serializer-komponenter istället gör ett anrop till en Flowscript-funktion. Denna funktion kan (och bör) göra allt som går att göra med hjälp av de vanliga pipelinekomponenterna, men med det tillägget att man i Flowscript kan använda continuations. JavaScript brukar förknippas med små programsnuttar som körs i klientens webbläsare, så det skall påpekas att det i fallet med Flowscript gäller att dessa funktioner körs på servern.

Många hävdar att det är en nackdel att man är tvungen att skriva sina pipelines i Flowscript/JavaScript ifall man vill använda continuations. JavaScript anses inte vara ett lika ”fint” språk som Java, och jag håller till viss del med om detta. Men mina erfarenheter från implementationen av WPFC visar att Flowscript-koden kan hållas relativt kort och enkel även i större applikationer om man programmerar enligt MVC-paradigmen. Så man behöver inte drabbas av eventuella brister i JavaScript gentemot Java ifall man är försiktig.

För att generera webbsidor i Flowscript använder man funktionen *sendPageAndWait()*, som förutom att skicka den genererade sidan via HTTP till klienten (webbläsaren) skapar ett continuation-objekt. En referens till detta continuation-objekt finns tillgängligt i den genererade webbsidan så att t.ex. submit-knappar i HTML-koden kan referera till denna. När sedan användaren gör submit på webbsidan skickas denna continuation-referens via URL’en till webbservern och vidare till Cocoon. Cocoon slår då upp det refererade continuation-objektet och aktivera detta. När continuation-objektet aktiveras återupptas exekveringen av Flowscript-funktionen i den instruktion omedelbart efter *sendPageAndWait()* anropet. På detta sätt åstadkommer Flowscript att *sendPageAndWait()*-funktionen ser ut att vara en blockerande funktion som inte returnerar förrän användaren svarat på den sida som skickades.

1.3.3 Generering av dynamiska sidor: XSP och Velocity

När jag skrivit att en webbsida genereras har jag nämnt att detta kan göras endera av en pipeline, eller via *sendPageAndWait()*-funktionen i Flowscript. Egentligen är det bara pipelines som genererar sidor i Cocoon eftersom det som händer när *sendPageAndWait()* anropas är att en angiven pipeline exekveras för att generera en sida.

Ofta vill man kunna visa dynamiska data i den sida som skall visas; dvs. man vill att sidan skall visa data som räknats fram av Flowscript-koden. Cocoon levereras som standard med en hel uppsjö pipelinekomponenter för att åstadkomma just detta. Jag kommer här att bara nämna två: XSP och Velocity.

Anledningen till att jag beskriver XSP är att denna komponent i dokumentationen för Cocoon omnämns som ”standard” att användas när man behöver generera dynamiska webbsidor. Som beskrivs i kapitel 1 anser jag dock att denna komponent har ett antal brister och därför inte bör användas. Utifrån arbetet med att skriva WPFC-applikationen förordar jag att man istället använder komponenten Velocity för att generera dynamiska webbsidor.

Både XSP och Velocity är implementerade i form av pipeline generator-komponenter. Båda genererar sidor utifrån en mall-fil skriven i språken XSP respektive Velocity, samt de Java/Flowscript-objekt som skickats via parametrar till *sendPageAndWait()*. En XSP-mall är en XML-fil där man inom speciella taggar kan skriva vanlig Javakod för att generera text som ersätter taggen i den färdiggenererade sidan. Denna Javakod kan använda sig av parametrarna till sendPageAndWait() för att generera dynamiskt innehåll i resultatsidan.

En Velocity-mall består av all statisk text (t.ex. HTML) som den genererade sidan skall bestå av, samt ett antal Velocity-specifika kommandon för att generera de dynamiska delarna av sidan. De Velocity-kommandon som står till förfogande är mycket enkla: det enda man kan göra är att loopa (kommandot ”#foreach”), sätta upp villkor (”#if”) samt hämta data från de Java/JavaScript-objekt som skickats som parametrar via *sendPageAndWait()*. Enligt min erfarenhet från arbetet med att implementera WPFC är detta enkla språk fullt tillräckligt för de behov man har att skriva kod för presentationslogik i den dynamiska webbsidan. Jag anser till och med att denna enkelhet är en fördel eftersom den uppmuntrar till att man endast placerar kod för presentationslogik i webbsidorna, och inte annan kod som inte hör hemma där enligt MVC-modellen.

2 Uppdelning mellan presentation, data och logik i Cocoon

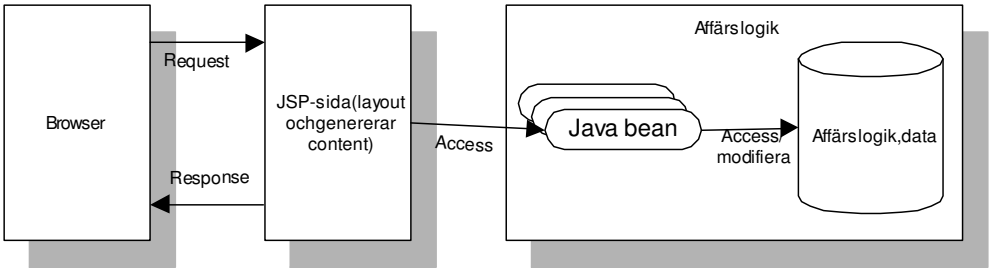
När vi nu i kapitel 1.3 gått igenom grunderna i hur Cocoon fungerar är vi redo att titta närmare på vilka mekanismer som Cocoon erbjuder för att hjälpa till att hålla isär logik, data och presentation/layoutinformation i de webbapplikationer man skriver. Min erfarenhet från WPFC är att MVC (Model-View-Controller)-modellen är bra att följa ifall denna uppdelning skall bli lyckad. Som vi skall se har Cocoon utmärkt stöd för MVC, varför det är på sin plats att förklara vad som menas med detta begrepp. Det gör vi i detta kapitel.

2.1 Model 1 och Model 2 arkitektur

Många webbpubliceringssystem som byggts med hjälp av någon av dom vanligaste plattformarna JSP, ASP eller PHP består av en mängd sidor som innehåller presentationsbeskrivning, presentationslogik och affärslogik ihopblantat. D.v.s. man har sidor som innehåller både HTML och programkod om vartannat. Detta kan i förlängningen leda till system som är svåra att underhålla och förstå för andra än programmakarna själva. Om förändringar i layout skall göras, måste den som gör dessa förändringar vara kunnig i både programmering och layout.

I byggandet av ”traditionella” system, d.v.s. system som inte är webbaserade, har man sedan länge talat om så kallade n-lager (n-tier) system som ett sätt att i någon mån minska risken för att man hamnar i situationen att presentationslogik, data och affärslogik blandas. Ett system som använder 3-lager kan till exempel bestå av en underliggande databas, ett plattformsoberoende program som hanterar affärslogiken samt en programdel skrivet för något specifikt GUI API (till exempel MFC eller Motif) som hanterar presentation och användarinmatning. En liknande uppdelning för webbapplikationer borde vara ett steg i rätt riktning.

För att hålla isär kod för affärslogik och presentationslogik föreslås i version 0.91 av JSP-specifikationen att man delegerar all affärslogik till att hanteras av Java beans, istället för att skriva sådan kod direkt i JSP-sidorna. Denna arkitektur kallas i specifikationen för en Model 1-arkitektur. Enligt resonemanget ovan skulle detta motsvara ett 2-tier system. Genom att använda Model 1 arkitekturen hoppas man kunna göra JSP-sidorna mer oberoende av förändringar i den underliggande affärslogiken. Förändringar i affärslogiken kan göras utan att JSP-sidorna behöver ändras så länge gränssnittet mellan affärslogik och JSP hålls konstant.



Figur 1 Schematisk bild av ett Model1-system

Problemet med Model 1 arkitekturen är att JSP-sidorna kommer att behöva både *presentera* content och *generera* content utifrån resultat som hämtas från affärslogiken. Att generera content

kräver oftast att programkod måste skrivas, så i Model 1 arkitekturen är det fortfarande svårt att göra en uppdelning där man inte skall behöva vara programmerare för att hantera presentationens layout.

För att kunna separera content-generering från content-presentation beskriver JSP-specifikationen en arkitektur där man adderat en action-hanterare i form av servlets som tar hand om alla requests och kommunicerar med affärslogiken för att generera content. JSP-sidorna får sedan tillgång till detta genererade contents för presentation. Denna arkitektur benämns Model 2 och gör det möjligt att hålla JSP-sidorna fria från content-genererande Java-kod.

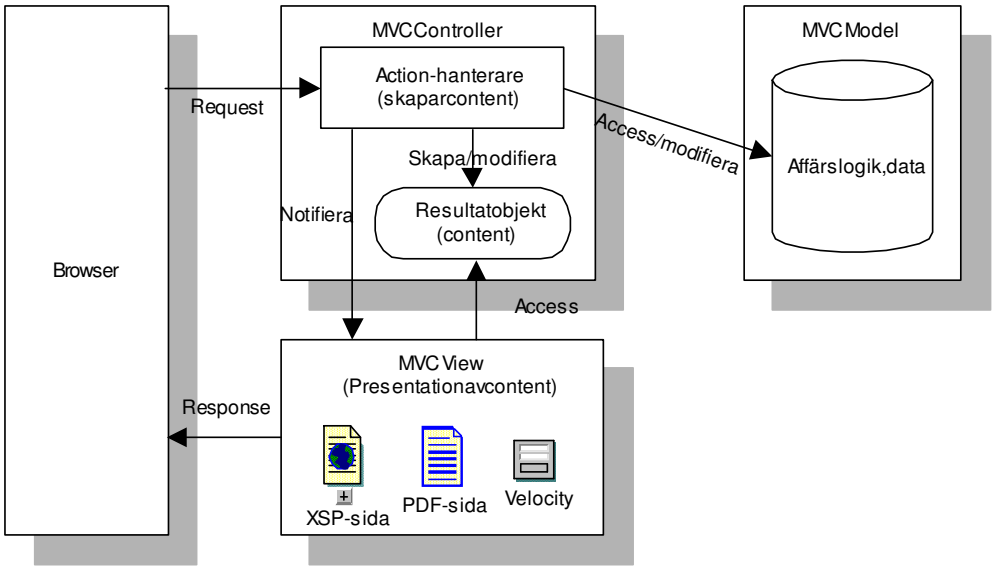
2.2 MVC arkitektur

Att dela upp system i olika lager för affärslogik, presentation och data är enkelt i teorin. I praktiken är det dock lätt hänt att till exempel presentationslogik och affärslogik börjar blandas ihop på grund av svårigheter att definiera vad som är vad. Ett design-pattern som kallas Model-View-Controller (MVC, se [9]) kan dock ge vägledning i situationer då gränserna mellan de olika ansvarsområdena är svåra att se.

I MVC representerar *modellen* det data och den logik som beskriver affärsprocessen man vill att systemet skall modellera.

Vyn i MVC hanterar *presentationen* av modellen. Vyn ser till att modellens data visas på rätt sätt. Vyn måste hålla sig uppdaterad om förändringar av datat i modellen, endera genom att begära uppdatering om nuvarande tillstånd i modellen eller genom att ta emot notifieringar om förändringar i modellen. Det bör vara möjligt att ha flera vyer mot ett och samma modellobjekt.

Controllern i MVC hanterar användarinteraktionen och konverterar denna till en serie *actions*. Actions skickas till modellen som utför motsvarande affärsuppgifter varpå vyn ändras för att visa resultatet.



Figur 2 Schematisk bild av ett MVC-system

Begreppet MVC härstammar från språket Smalltalk; men på grund av dess objektorienterade och generella karaktär kan MVC användas som design-pattern även i andra miljöer. Exempel på implementationer av MVC i JSP-miljö ges i ref [1] samt i Apacheprojektet *Struts*. Model 2 som beskrivits ovan, är en variant av en MVC arkitektur där modellen representeras av affärslogiken, controllern representeras av servlets och vyn representeras av JSP-sidor.

2.3 MVC och Cocoon

Cocoon's arkitektur uppmuntrar till att man strukturerar sina webbapplikationer enligt MVC-modellen. I arbetet med att utveckla WPFC kom jag underfund om att denna struktur fungerar utmärkt, men att man av praktiska skäl bör lägga all kod som implementerar MVC-model-komponenten (dvs. affärslogiken) i en egen server utanför Cocoon (dvs. i en separat process). I princip skulle man kunna implementera hela affärslogiken i Flowscript samt några eventuella hjälpklasser i Java. Men av orsaker som beskrivs i kapitel 4.3.2 är en bättre strategi att låta MVC-model-komponenten ligga utanför Cocoon-applikationen.

I de Cocoon-applikationer som jag skrivit har jag använt Cocoon-komponenter till att implementera två av MVC-modellens delar:

1. MVC-Vyn i form av Cocoon pipelines och Velocity-sidor.
2. MVC-Controllern i form av Cocoon Flowscript.

MVC-model-komponenten har placerats i en separat Java-RMI-server [14] som kör i en egen process oberoende av Cocoon.

För att skapa en action-hanterare (dvs. den komponent som dirigerar http-anrop till MVC-controllern) i sin Cocoon-applikation skall man i sin `sitemap.xmap`-fil definiera följande pipeline:

```
<map:match pattern="*.action">
      <map:call function="{1}"/>
</map:match>
```

Detta innebär att alla HTTP-requests på formen "http://.../funktionsnamn.action" kommer att skickas till Flowscript-funktionen "funktionsnamn" i MVC-controller-komponenten.

På grund av att Flowscript stöder continuations (så som beskrivs närmare i kapitel 1.3.2) kan man skriva koden i MVC-controller-komponenten på traditionellt linjärt sätt. Man slipper skriva kod för att explicit hålla reda på programmets tillstånd (så som skulle ha behövts i MVC-controllern för ett JSP-program). För att hantera continuations behövs dock en pipeline i `sitemap.xmap`. Den skall definieras på följande sätt:

```
<map:match pattern="*.continue">
      <map:call continuation="{1}"/>
</map:match>
```

Detta innebär att alla HTTP-requests på formen "http://.../continuation-id.continue" kommer att leda till att Cocoon aktiverar den continuation som har ID "continuation-id". När funktionen *sendPageAndWait()* anropas i MVC-controllerns Flowscript-kod genereras detta continuation-id (som i själva verket är ett långt 128-bitars tal som anses "omöjligt" att förfalska). Funktionen *sendPageAndWait()* skapar en ny sida att visas i klientens webbläsare med hjälp av en MVC-vy-komponent. MVC-vyn i form av en Velocity-sida ser till att lägga in detta continuation-id med ändelsen ".continue" i alla länkar för att göra submit på sidan. När användaren sedan trycker på någon submit-knapp kommer ramverket att se till att rätt continuation aktiveras och motsvarande

MVC-controller-funktion som gjorde det ursprungliga *sendPageAndWait()*-anropet kan fortsätta exekvera. Hanteringen av continuations beskrivs mer ingående i kapitel 3.1.

Cocoon underlättar implementationen av MVC-vy-objekten med hjälp av komponenten Velocity (det finns fler liknande komponenter, t.ex. XSP, men Velocity är det som jag använt mig av). Mekanismen för att skicka data i form av Java/Flowscript-objekt från MVC-controllern finns färdig (objekten skickas som parametrar i *sendPageAndWait()*-funktionen) och Velocity innehåller ett enkelt språk i vilket man kan skriva presentationslogiken.

3 MVC-Controllern och problemet med ”*inversion of control*”

I begynnelsen skrevs datorprogram på ett sådant sätt att interaktionen med användaren skedde utifrån programmets kontroll. Med hjälp av någon `input`-funktion ställde programmet frågor till användaren, och programmet fortsatte inte exekveringen förrän frågan besvarats. Input-funktionen sägs vara *blockerande*. I detta dokument betecknas program som fungerar på detta sätt, att vara skrivna på ett *linjärt sätt* (Eng. *linear style*). Program skrivna på linjärt sätt har den goda egenskapen att det inte uppstår oförutsedda tillstånd i dom (förutsatt att alla möjliga svar hanteras på korrekt sätt i varje fråga programmet ställer - men det underlättas genom att programmet kontrollerar vilka möjliga svar som kan ges.) Linjära program har bara en enda evalueringsstack att ta hänsyn till vilket gör dom relativt enkla att följa.

När client/server-system skulle byggas blev programmeringen dock mer komplicerad eftersom dessa system kräver att servern alltid skall vara beredd att betjäna förfrågningar. Det går inte att skriva en server på linjärt sätt om den skall kunna betjäna flera klienter samtidigt. Servern måste kunna hentera en evalueringsstack för varje klient den betjänar vilket omöjliggör linjär programmering. En lösning på detta problem är att låta servern vara multitrådad och ge en servertråd till varje uppkopplad klient. På detta sätt kan varje servertråd använda blockerande funktioner utan att för den skull blockera övriga servertrådar.

När grafiska användargränssnitt (GUI) började användas, synliggjordes oförmågan hos program skrivna på linjärt sätt att passa in i denna miljö. Eftersom den stora poängen med ett GUI är att ersätta blockerande och begränsande programstyrda input-funktioner med ett mer öppet system som *kontrolleras av användaren*.

Det är i ett GUI-system ”förbjudet” att använda blockerande funktioner eftersom dom sätter GUI-funtionaliteten ur spel. Man har löst interaktionsproblemet genom att låta program anmäla sitt intresse av att bli notifierade när olika händelser inträffar; t.ex. när användaren skriver in text, trycker på knappar, klickar med musen etc. Alla program måste ha en *event-loop* som väntar på sådana händelser. Programmeraren måste se till att event-loopen innehåller kod för att *explicit* hantera programmets olika tillstånd och att den hanterar alla möjliga flöden i programmet.

Uppkomsten av webbsystem komplicerar bilden ytterligare genom att dom överlämnar kontrollen av hela GUI’t till en fristående webbläsare som arbetar off-line och bara kommunicerar med programmet (servern) när användaren skickar formulär och förfrågningar om sidor till servern. Webbläsarens ”tillbaka”-knapp och möjligheten att fritt hoppa fram och tillbaks bland tidigare besökta sidor gör att programmet i servern inte bara måste kunna hantera flera parallella evalueringsstackar; den måste även vara beredd på att en klient på ett oförutsägbart sätt hoppar fram och tillbaka i programflödet inom varje evalueringsstack.

Med GUI och webbsystemen har man flyttat kontrollen av programflödet från programmet självt över till användaren. Detta brukar i datalogiska sammanhang betecknas som ”inversion of control”. Inversion of control är dock inte gratis. Priset man betalar är ökad komplexitet i de program som skall exekvera i GUI och/eller webbmiljö.

Trots de tillkortakommanden som linjära program uppvisar utgör linjär programmering på grund av sin enkelhet ett attraktivt mål att återvända till. Å andra sidan vill man inte gå miste om alla de fördelar moderna client/server, GUI och webbsystem erbjuder tack vare denna ”inversion of control”. Hur skall man då gå tillväga för att både äta kakan och ha den kvar?

3.1 Continuations

Webbapplikationer kännetecknas av att användaren har stor frihet att själv styra programflödet genom att bläddra fram och tillbaka mellan applikationens sidor. Denna egenskap som kallas ”inversion of control” är som tidigare beskrivits svår att handskas med och leder till att koden för att hantera flödeslogiken i applikationerna delas upp och sprids ut på flera ställen. Applikationerna blir sidorienterade och det blir svårt att utifrån programkoden se vad systemen gör. För att komma till rätta med detta och kunna skriva koden för programflödet i webbapplikationer på traditionellt linjärt sätt (dvs. som om alla i programmet ingående instruktioner var blockerande) har föreslagits att man använder continuations. Se ref [4]och [5].

Continuations är ett begrepp som ursprungligen kommer från λ-kalkyl och funktionella språk, se t.ex. [12] och [13]. En continuation kan beskrivas som ett objekt som representerar ”resten av evalueringen av programmet”, eller en partiellt evaluerad funktion.

När en applikation skapar ett continuation-objekt sparas applikationens tillstånd - dvs. programräknare, register och variabler - i continuation-objektet. Programmet kan sedan fortsätta att göra andra saker, men genom att packa upp och aktivera continuation-objektet kan programmet när som helst hoppa tillbaks till det tillstånd som continuation-objektet representerar. Detta stämmer bra överens med hur en webbapplikation fungerar: När applikationen visar ett formulär (en webbsida som skall fyllas i av användaren) skapas ett continuation-objekt. Applikationen kan sedan gå vidare och serva andra requests. När användaren fyllt i formuläret och sedan gör submit kommer applikationen att använda continuation-objektet för att återgå till det tillstånd som applikationen befann sig i då formuläret skickades till användaren.

Teorin bakom continuations och deras betydelse för webbaserade system har beskrivits av bl.a. Queinnec [4] och [5]. Jag kommer i detta dokument att utvidga detta med ett praktiskt exempel som visar skillnaden mellan ett program som utnyttjar continuations (program skrivet i Cocoon Flowscript) och ett program skrivet i JSP som inte använder continuations.

Flowscript använder first-class continuations. Med ”first-class” menas att vanliga variabler i språket kan tilldelas värdet av ett continuation-objekt. Dvs. ett continuation-objekt skiljer sig inte från ”vanliga” objekt i språket (som t.ex. klassobjekt, arrayer etc.).

3.2 Exempel på MVC-controller i JSP vs. Flowscript

I detta kapitel ges ett exempel som syftar till att visa skillnaden mellan en MVC-controller skriven i JSP och en MVC-controller skriven i Flowscript. I detta fall visar JSP-versionen strukturen för ett sidorienterat system. Flowscript-versionen visar hur systemet ser ut om det skrivs i ett språk som har stöd för continuations och som därmed kan skrivas på linjärt sätt.

Båda versionerna av programmet är tänkta att fungera på samma sätt utifrån användarens perspektiv. Programmets uppgift är enkel: Läs in två heltal, summera dessa och presentera resultatet. Heltalen matas in av användaren via två HTML-formulär och resultatet visas på en separat HTML-sida.

För att förenkla exemplet har jag i båda implementationerna bortsett från problemet att användaren kan mata in ett ogiltigt ”tal” i de HTML formulär som används för att läsa in talen. Användaren kan till exempel mata in en textsträng istället för ett tal, eller skicka en tom sträng. Ett komplett program skulle behöva kod för att verifiera att de inmatade talen är giltiga innan beräkning kan ske. Jag har för enkelhets skull i exemplet antagit att de HTLM-forulär som användaren använder för att mata in talen alltid ger ett giltigt (ej tomt) heltal som resultat. I figuren nedan visas grafiskt användarens bild av hur systemet fungerar:

Figur 3 Användarens bild av additionsexemplet

3.2.1 MVC-controller i JSP

Detta exempel visar hur programmet för att addera två tal kan implementeras i JSP. Programlistningen nedan visar hur MVC-controllern som definierar programflödet kan se ut. Begreppet MVC och hur det stöds i Cocoon beskrevs i kapitel 1. För ett komplett fungerande system i JSP krävs förutom detta även kod för att implementera själva MVC-ramverket som en Java servlet. Hur man implementerar ett MVC-ramverk i form av en Java servlet ligger utanför ramarna för detta dokument, men exemplet bygger på det MVC-ramverk som beskrivs i [1] kapitel 6.

Förutom MVC-ramverket behövs webbsidorna `firstNumber.jsp`, `secondNumber.jsp` samt `showResult.jsp` för att hantera inmatningen av talen samt presentera resultatet. Dessa sidor listar jag inte här, men dom är bara enkla JSP-sidor som läser in två tal via HTML-formuär samt skriver ut resultatet från beräkningen som lagrats i request-parametern ”view_result”.

```
/**
 *  MVC Controller servlet action that calculates the sum of two numbers
 */

import javax.servlet.http.*;
import actions.*;

public class AddTwoNumbers implements Action {
    public static final String RESTART = "restart";
    public static final String FIRST_NUMBER = "firstNumber";
    public static final String SECOND_NUMBER = "secondNumber";
    public static final String RESULT = "result";

    public ActionRouter perform(HttpServlet servlet,
        HttpServletRequest request, HttpServletResponse response)
    throws java.io.IOException, javax.servlet.ServletException {
        HttpSession session = request.getSession();

        // Check if the user wants to restart the computation
        if(request.getParameter(RESTART) != null) {
            session.setAttribute(FIRST_NUMBER, null);
            session.setAttribute(SECOND_NUMBER, null);
            return new ActionRouter("/firstNumber.jsp");
        }

        // Fetch any submitted number from the request
        String firstNumberString = request.getParameter(FIRST_NUMBER);
        String secondNumberString = request.getParameter(SECOND_NUMBER);
        if(firstNumberString != null) {
            session.setAttribute(FIRST_NUMBER, firstNumberString);
```

```
        } else
        if(secondNumberString != null) {
            session.setAttribute(SECOND_NUMBER, secondNumberString);
        }

        // If any of the two numbers are missing, show the corresponding
        // number input form.
        if(session.getAttribute(FIRST_NUMBER) == null) {
            return new ActionRouter("/firstNumber.jsp");
        } else
        if(session.getAttribute(SECOND_NUMBER) == null) {
            return new ActionRouter("/secondNumber.jsp");
        }

        // Both numbers are available, calculate sum
        int result = Integer.parseInt(
        (String)session.getAttribute(FIRST_NUMBER)) +
            Integer.parseInt((String)session.getAttribute(SECOND_NUMBER));

        // Initialize view data
        request.setAttribute("view_"             +             FIRST_NUMBER,
            session.getAttribute(FIRST_NUMBER));
        request.setAttribute("view_"             +            SECOND_NUMBER,
            session.getAttribute(SECOND_NUMBER));
        request.setAttribute("view_" + RESULT, Integer.toString(result));

        return new ActionRouter("/showResult.jsp");
    }
}
```

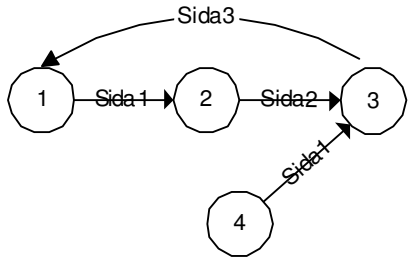
Figur 4 MVC controller i form av en Java servlet/JSP

Programmet fungerar så att *perform*-funktionen i ovanstående MVC action ”AddTwoNumbers” kommer att anropas när användaren anger URL-adressen `http://<host>/AddTwoNumbers.do` i sin webbläsare. MVC-ramverket har sett till att mappa alla adresser som slutar med `.do` att anropa perform-funktionen i motsvarande action-klass. Alla i systemet ingående JSP-sidor (d.v.s. firstNumber.jsp, secondNumber.jsp etc.) skall skicka den inmatade informationen via request-variabler genom ett anrop till `http://<host>/AddTwoNumbers.do`.

Man ser tydligt att merparten av koden består av tillståndshantering. Controllern måste explicit hålla reda på programmets tillstånd med hjälp av sessionsvariablerna ”firstNumber” och ”secondNumber”. Innan beräkningen kan ske måste controllern försäkra sig om att båda talen har matats in på sidorna first- och secondNumber.jsp. D.v.s. den måste försäkra sig om att de båda sidorna first- och secondNumber.jsp har visats för användaren.

När talen matats in kan beräkningen göras och lagras i request-variabeln ”view_result”. Därefter görs en HTTP-forward till sidan showResult.jsp som läser av ”view_result”-variabeln och skriver ut den (MVC-ramverket använder objektet ActionRouter för att göra en HTTP-forward.)

JSP-exempelprogrammet kan sägas implementera en state-maskin med följande tillståndsdiagram:



Tillstånd	Sessionsvariabel ”firstNumber”	Sessionsvariabel ”secondNumber”	Sida att visa	Nytt tillstånd
1	null	null	firstNumber.jsp	2
2	x	null	secondNumber.jsp	3
3	x	y	showResult.jsp	1
4	null	y	firstNumber.jsp	3

Figur 5 Tillståndsdiagram för JSP-versionen av exemplets MVC controller.

Kolumn 1 - 3 anger de olika ursprungstillstånden programmet kan befinna sig i. Kolumnen ”sida att visa” anger vilken webbsida programmet visar som resultat av tillståndet. När användaren gör submit på denna sida leder det till att programmet hamnar i ett nytt tillstånd som anges i sista kolumnen.

Den explicita hanteringen av programtillståndet som controllern måste göra påverkar kodens utseende så att den blir ganska svårläst. Det är enligt min mening inte uppenbart vid första anblicken att se var programmet gör.

3.2.2 MVC-controller i Flowscript

Detta exempel visar hur programmet för att addera två tal kan implementeras i Cocoon Flowscript. Programlistningen visar koden för MVC-controllern. Till skillnad från i JSP stöds MVC som standard i Cocoon, så MVC behöver inte implementeras separat. Det enda som behövs förutom nedanstående är webbsidorna `first-number.vm`, `second-number.vm` och `result.vm`. Dessa sidor listas inte här, men dom är enkla *Velocity*-sidor (se [2]) som läser in två tal med hjälp av HTML-formulär och presenterar resultatet av beräkningen genom att skriva ut värdet som lagrats i Flowscript-objektet ”result”.

```
/**
 *  MVC Controller in FlowScript that calculates the sum of two numbers
 */

importPackage(java.lang);

function addTwoNumbers()
{
    // Fetch the first number
    cocoon.sendPageAndWait("show-first-number.vm", {});
    var firstNumberString = cocoon.request.getParameter("firstNumber");

    // Fetch the second number
    cocoon.sendPageAndWait("show-second-number.vm", {});
    var secondNumberString = cocoon.request.getParameter("secondNumber");

    // Perform the computation
    var result = Integer.parseInt(firstNumberString) +
        Integer.parseInt(secondNumberString);

    // Show result
    cocoon.sendPage("show-result.vm", {
        "result": result,
        "firstNumber": firstNumberString,
        "secondNumber": secondNumberString
    });
}
```

Figur 6 MVC controller i form av Cocoon Flowscript

I denna version hanteras programmets tillstånd helt implicit. När ett anrop till funktionen *sendPageAndWait* har skett kan controllern vara säker på att respektive tal verkligen matats in och att talet finns tillgängligt i kommande steg. D.v.s. controllern kan vara säker på att den sida som controllern gjort sendPageAndWait på verkligen har visats för användaren; den kan också vara säker på att sidan har visats i den ordningsföljd som controllerprogrammet definierar. SendPageAndWait skapar ett continuation-objekt som skickas som implicit parameter till den specificerade Velocity-sidan (ett godtyckligt Java/Flowscript-objekt kan skickas via den andra parametern till sendPageAndWait, men här skickas ingenting förutom det implicita continuation-objektet, därav parametern {}). Velocity-sidan använder ID't för detta continuation-objekt som submit-adress i den HTML-form som Velocity-sidan visar.

När användaren angett ett nummer i HTML-formen och gör submit kommer ett nytt HTTP-request att sändas till Cocoon-servern. Cocoon slår upp och aktiverar det continuation-objekt som har det ID som angetts i URLen. MVC-controller-programmet i Flowscript kommer då att återuppta exekveringen precis som om sendPageAndWait returnerat. På detta sätt kommer funktionen sendPageAndWait att fungera som en blockerande input-funktion utifrån programmets synvinkel.

3.2.3 Skillnader i funktionalitet mellan JSP- och Flowscript-exemplen

De båda ovan beskrivna implementationerna av exemplet förutsattes fungera på samma sätt utifrån användarens synvinkel, men vid närmare granskning går det att hitta skillnader:

Trots (eller på grund av) våra ansträngningar att explicit hantera kodningen för programmets tillstånd i JSP-exemplet kan dess funktionalitet vara en aning förvirrande för användaren.

3.2.4 Problem för JSP-exemplet att hantera navigationsknapparna

Antag att användaren anger de båda talen på sidorna 1 och 2; programmet visar då resultatet. Om användaren sedan backar två steg visas sidan 1 i webbläsaren. Användaren skriver in ett nytt värde för första talet och gör submit. Nu visas resultatsidan med resultatet summan av det nya första talet + det tidigare andra talet. Detta är förmodligen inte det användaren förväntat sig. Enligt användarens sätt att se det borde programmet nu befinna sig i tillstånd 1, och enligt tillståndsdiagrammet i Figur 5 borde sida 2 visas efter det att sida 1 ifyllts.

Efter det att resultatsidan visats första gången och användaren sedan backat tillbaks två steg till sidan ett är programmets tillstånd fortfarande i tillstånd 3. Programmets tillstånd och presentationens tillstånd har hamnat i osynk. Detta på grund av att webbläsaren aldrig skickar någon information till programmet om att användaren trycker på ”tillbaka”-knappen. När användaren gör submit på sidan 1 märker programmet att både session-variabeln ”firstNumber” och ”secondNumber” finns ifyllda och visar då resultatsidan 3.

Det finns fler varianter av ovanstående problem: Antag att användaren gör submit på sida 1 och därefter backar tillbaks till sida 1. Om användaren sedan väljer att uppdatera sida 1 (dvs. använder sig av webbläsarens funktion att ladda om den nuvarande sidan) kommer sida 2 att visas. Användaren förväntar sig dock att sidan 1 skall visas i uppdaterad form.

Detta beror precis som tidigare på att programmets tillstånd och presentationens tillstånd har hamnat i osynk. Programmet befinner sig i tillstånd 2 medan det i presentationen ser ut som att det befinner sig i tillstånd 1.

Det går som sagt att hitta fler varianter på samma tema, men själva grundorsaken till dessa problem är att den konceptuella funktion hos webbläsarens navigationsknappar inte återspeglas i hur datat i de globala sessionsvariablerna som definierar programmets tillstånd hanteras. Utifrån användarens synsätt betyder ”tillbaka”-knappen ”gå tillbaks till föregående steg i programflödet”, men i själva verket händer ingenting i programmet när användaren trycker på ”tillbaka”. Det finns ingen koppling mellan webbläsarens ”tillbaka”-knapp och programmet.

3.2.5 Problem för JSP exemplet att hantera terminerade sessioner

Ett annat problem som kan uppkomma i JSP-versionen, men inte i Flowscript-versionen, hittas om vi studerar tillståndsdiagrammet i Figur 5. Tillstånden 1, 2 och 3 utgör det normala flödet i programmet. Tillstånd 4 är dock lite speciellt eftersom det vid första anblicken verkar som att detta tillstånd aldrig borde kunna uppstå – ingen båge leder till detta tillstånd i grafen.

Det finns dock ett ”naturligt” sätt för detta tillstånd att uppkomma: om användaren anger det första talet på sida 1 och därefter låter sidan 2 visas en lång stund; så lång att HTTP-sessionen hinner gå ut. När sedan användaren gör submit på sidan 2 kommer alla sessionsvariabler att vara nollställda (eftersom en ny session skapas och ersätter den gamla) och programmet hamnar i tillstånd 4.

En lösning på problemet är att låta programmet visa ett meddelande för användaren om att denne måste börja om eftersom sessionen terminerat. Att implementera detta kräver dock en åtgärd av programmeraren, och om möjligheten att denna situation kan uppstå förbises av programmeraren kommer detta aldrig att bli gjort. Det finns en risk att denna typ av problem inte hittas i ett JSP-system, i synnerhet ifall systemet är stor och komplicerat.

Flowscript-systemet bygger också på HTTP-sessioner, så användaren som använder vårt Flowscript-exempel kommer också att märka att sessionen gått ut ifall han väntar med att göra submit av en sida tillräckligt länge. Skillnaden här är att programmet inte kommer att hamna i något oförutsett tillstånd på grund av detta. När sessionen terminerar avbryts programmet och

när användaren sedan gör submit på sidan vars session gått ut kommer användaren att hänvisas till en i Cocoon konfigurerad sida som förklarar vad som hänt. Dvs. submit-knappen hänvisar till en continuation som inte längre existerar eftersom dess session har gått ut, och Cocoon dirigerar automatiskt om sådana requests till en speciell felmeddelandesida.

Jag påstår att detta beteende är säkrare eftersom det här inte finns någon risk att programmet hamnar i ett oförutsett tillstånd. Det finns inget sätt för programmet att fortsätta efter det att sessionen terminerat – *inte ens i princip*. Detta på grund av att alla till sessionen hörande continuations invalideras då sessionen går ut.

3.2.6 Problem för JSP-exemplet att hantera förfalskade request

JSP-programmet kan också hamna i tillstånd 4 om en användare förfalskar ett HTTP-request så att det för programmet ser ut som en submit från sidan 2 - utan att användaren först har gjort submit på sidan 1. I vårt fall hanteras denna situation på ett korrekt sätt genom att sida 1 kommer att visas, men i mer komplicerade system kan problem av denna typ vara svåra att hitta och åtgärda. Om inte alla möjligheter för en användare att komma åt skyddade sidor förutses och hindras av programmeraren kan det leda till säkerhetshål i systemet.

I Flowscript är det inte möjligt för en användare att förfalska requests på ett sådant sätt att programmet hoppar till tillstånd för vilka programmet ännu inte genererat evalueringsvägar till. Detta beror på att användaren måste ange ett continuation id vid varje request. Eftersom continuation id inte kan förfalskas är det omöjligt för användaren att ange en continuation till något tillstånd som ännu inte evaluerats fram av programmet. Specifikt för vårt exempel gäller att tillstånd 4 i Flowscript-versionen inte kan uppstå. *Inte ens i princip*, eftersom användaren måste ange continuation id från sida 1 innan programmet visar sida 2.

3.2.7 Problem med continuations

Jag har identifierat några problem som bör beaktas när man väljer att skriva sin MVC-controller i Cocoon Flowscript:

1. Flowscript continuation-objekt sparas i sessionen (dvs. i servern) efter det att ett anrop till *sendpageAndWait* skett. Eftersom hela programmets tillstånd sparas i varje continuation kan det leda till att minnesåtgången blir stor ifall man har definierat många variabler och/eller om dessa variabler innehåller mycket data. McGuire [3] beskriver ett system där continuation-objekten lagras i klientens webbläsare istället för i servern. Detta skulle vara en lösning på problemet, men det stöds inte i Flowscript.
2. Varje continuation är unik och går inte att ”flytta över” till en annan webbserver även om denna kör samma program. Detta är ett problem ifall man vill bygga en site med lastbalansering. Dvs. en site där man kör samma program på flera parallella webbservrar och vill kunna dirigera inkommande HTTP-requests till den server som för närvarande har lägst belastning. En webbserver som får en continuation id till en continuation som genererats av en annan server kan inte använda denna continuation.
3. Om en användare sparar adressen till en sida som har en continuation i sin URL kommer användaren inte att kunna återvända till sidan via den sparade adressen efter det att användarens session gått ut. När användarens session går ut invalideras automatiskt alla continuation-objekt som tillhör denna session. Nästa gång användaren försöker använda sin sparade URL kommer det continuation-objekt som anges i URLens continuation-id inte att vara giltigt. Användaren kommer istället att få se ett felmeddelande som talar om att sidan inte kan hittas.

3.3 Slutsatser från continuation-exemplet

Även om man bygger ett JSP-system helt enligt MVC/Model2 arkitekturen kommer de i systemet ingående webbsidorna att avspeglas i hur koden för controllern utformas. Man kommer i koden för MVC-controllern att tvingas ta explicit hänsyn till att systemet är sidorienterat. Detta på grund av att JSP inte har stöd för continuations, och MVC-controllern därför inte kan skrivas på linjärt sätt.

Trots att det beskrivna exemplet är väldigt litet och enkelt visar det sig att den sidorienterade JSP-versionen innehåller flera ”fallgropar” som gör att programmet inte fungerar som avsett i alla situationer. Detta beroende på att programmets tillstånd och presentationens tillstånd hamnar i osynk när användaren godtyckligt bläddrar mellan sidor med hjälp av webbläsarens navigationsknappar. Det finns risk att dessa problem blir större och allvarligare ju större sidorienterade JSP-system man bygger.

Systemet byggt i Cocoon Flowscript löser dessa problem eftersom mappningen mellan sidornas continuations och programmets tillstånd bibehålls när användaren bläddrar mellan sidor. Presentationens tillstånd hålls alltid synkroniserat med programmets tillstånd.

Jag hävdar med utgångspunkt från ovanstående exempel och från de problem som beskrivs i kapitel 3.2.3 att webbsystem skrivna på linjärt sätt är betydligt enklare att bygga och enklare att *bygga rätt*, än sidorienterade webbsystem. System skrivna utan continuations (dvs. skrivna på ett sidorienterat sätt) får en struktur där de olika ingående sidorna har den centrala rollen snarare än den algoritm programmet är tänkt att implementera.

4 Genomförandebeskrivning: Implementation av WPFC

För att ge ett praktiskt exempel på hur en verklig applikation kan implementeras med hjälp av Cocoon har jag i samarbete med företaget Bytbil.com skrivit en prototyp av deras befintliga webbpubliceringsapplikation som är ett system för att annonsera ut bilar på webben. Prototypen implementerar en delmängd av den funktionalitet som finns i Bytbils nuvarande system. Prototypsystemet är ett slags "proof of concept" som implementerar några av de viktigaste funktionerna från det nuvarande systemet i grova drag.

WPFC är det samlingsnamn som jag använder för att beteckna delarna i den prototyp av Bytbil.com's webbpubliceringsapplikation som jag implementerat. Detta kapitel beskriver hur jag genomförde arbetet med att implementera WPFC-systemet. De problem jag stötte på på vägen beskrivs, samt deras konsekvenser och lösningar.

4.1 Om Bytbil.com

Företaget Bytbil.com har en plattform för att bygga webbpubliceringssystem åt andra företag inom bilbranchen. Bytbil.com bygger webbtjänster för tidbokning mm. åt bilverkstäder och har också en annonssite där återförsäljare kan publicera information om bilar dom har till försäljning. Det nuvarande ramverket som man använder för att bygga dessa publiceringssystem kring är skrivet i Python/Fast CGI, med en MySQL-databas i botten.

Egentligen består Bytbils system av två webbapplikationer: en ”kundwebb” där Bytbils kunder (dvs. bilhandlarna) själva kan lägga in information, bilder etc. på de bilar som respektive bilhandlare har till försäljning. Den andra delen (här kallad ”publik webb”) utgörs av en webbapplikation där allmänheten kan gå in och söka bland bilarna som bilhandlarna har lagt in, för att förhoppningsvis hitta bilar som uppfyller deras önskemål.

Utvecklingen av webbapplikationerna görs av två avdelningar inom företaget. Dessa avdelningar har olika inriktningar och ansvarsområden: en avdelning består mestadels av tekniker som ser till att den bakomliggande funktionaliteten/logiken i applikationerna fungerar. En annan avdelning har hand om presentationsdelen av systemen, d.v.s. layout och användargränssnitt. Man har också ett antal branschkunniga personer (bilbranschen) som ansvarar för sälj och för den information/data som behandlas i systemen.

4.2 Problembeskrivning

För att de olika avdelningarna skall kunna arbeta så oberoende av varandra som möjligt vill Bytbil.com att det skall finnas väldefinierade gränssnitt mellan systemens olika delar. Man vill kunna göra ändringar i den information/data som systemen behandlar utan att systemens logik och presentationslager påverkas alltför mycket. Vidare vill man att webbapplikationernas utseende skall gå att ändra utan att behöva programmera om systemens logik, och/eller behöva ändra datat.

För närvarande lagras all affärsrelaterad information (dvs. kundinformation och data om bilarna) i en databas, vilket gör att definitionen av de olika tabellerna i databasen kan sägas utgöra gränssnittet som möjliggör separationen av data från systemens övriga delar (logik och presentation). Man har på Bytbil.com uppfattningen att denna uppdelning fungerar bra, men att separationen av presentationslager från logiklager skulle kunna förbättras.

För att göra uppdelningen mellan logik och presentation tydligare har man tittat på olika XML/XSLT-baserade webbapplikationsramverk för att se vilka lösningar för detta de kan erbjuda. Exempel på webbapplikationsramverk som varit av intresse är PHP, JSP (Struts) och Cocoon.

Jag har i form av detta examensarbete gjort en prototyp i Cocoon av Bytbils nuvarande system i syfte att visa att det i princip skulle vara möjligt att migrera detta till Cocoon. Ett annat syfte med denna prototyp var att genom praktiskt arbete få en uppfattning om fördelar/nackdelar med Cocoon i jämförelse med JSP.

Genom att bygga prototypsystemet så att hela funktionskedjan fungerar (t.ex. från det att kunden (återförsäljaren) lägger in en annons tills dess att en användare tittar på annonsen i sin webbläsare) kommer man att få med alla systemets delar i prototypen. Syftet är att man skall kunna återanvända prototypens struktur ifall man väljer att implementera ett skarpt system.

4.3 Implementation

En stor del av arbetet med att implementera WPFC gick åt till att välja ut vilka delar i Cocoon (”blocks”) som var mest lämpade för uppgiften. Eftersom Cocoon består av så många block och där vissa egentligen löser samma sak fast med olika teknik, tog det lång tid att lära sig använda och prova fram vilka block som fungerade bra. Till slut kom jag i alla fall fram till en uppsättning block som fungerade bra: förutom Cocoon core-funktionalitet använde jag mig av Flowscript (för MVC-control-komponenten), Velocity (för att generera dynamiska sidor) samt blocket FOP (för att generera PDF-dokument).

4.3.1 Att hålla isär affärslogik och presentationslogik

De böcker och det mesta av den on-line-information jag läst om Cocoon beskriver XSP som en central komponent i Cocoon att användas för att generera dynamiska webbsidor. Detta gjorde att jag till en början lade in all kod – affärslogik, flödeslogik och presentationslogik hopblandat – i de i WPFC ingående XSP-sidorna. Vartefter programmet växte blev det allt svårare att få grepp om programflödet i applikationen eftersom flödeslogiken var utspridd på olika XSP-sidor och därmed svår att följa. Jag hade råkat ut för samma problem som man ofta gör när man skriver applikationer i JSP eller motsvarande page-tagging-språk.

Problemet med att programkoden, både affärslogik och presentationslogik, var utspridd på de olika XSP-sidorna var inget som jag reflekterade så mycket över i början. Van vid page-tagging-språk betraktade jag det som att ”så är det ju alltid när man bygger webbsystem”.

På grund av problemet att min applikation bestod av ett antal svåröverskådliga XSP-sidor insåg jag så småningom att flödeslogik och affärslogik måste flyttas ut från sidorna och att systemet skulle byggas enligt MVC-modellen. Jag började vid denna tidpunkt att titta närmare på hur en MVC-controller kunde implementeras på ett snyggt sätt i Flowscript.

Jag upptäckte snart dom stora fördelar Flowscript, tack vare stöd för continuations, ger i samband med byggandet av en MVC-controller. Flowscript gjorde det möjligt att helt eliminera flödeslogik och affärslogik från MVC-vyerna (XSP-sidorna).”På köpet” blev flödeslogiken samlad på ett ställe och kunde skrivas på linjärt sätt.

Så småningom tog jag dessutom beslutet att helt överge XSP till förmån för Velocity. Detta på grund av de problem med XSP som beskrivs i kapitel 4.3.3 och 4.3.4.

4.3.2 Praktiska problem med kod för MVC-model-komponenten i Cocoon

Euforisk över det nya verktyg vid namn Flowscript som jag hittat för att skriva koden för flödeslogiken med, började jag flytta över mer och mer kod till Flowscript-delen av mitt system. Till en början försökte jag implementera även affärslogiken, dvs. det som motsvarar MVCs *model*, i Flowscript och/eller Java-beans som anropades direkt av Flowscript. Efter en tid upptäckte jag dock praktiska problem med detta:

1. Dålig typkontroll i Flowscript:

   Flowscript är på många sätt ett utmärkt programmeringsspråk, men Flowscript är dynamiskt typat. Dynamisk typning innebär att variabler inte har typ, men de värden man tilldelar variabler är ändå alltid av en viss typ. Detta leder lätt till att typfel uppstår i run-time då man t.ex. försöker jämföra två variabler som man tror är av samma typ, men som i run-time visar sig vara helt olika. I vanlig Java skulle man istället ha fått typfel vid kompileringen, vilket jag anser vara en mycket bättre lösning. Att Flowscript inte har typkontroll vid kompileringen gjorde mig tveksam till att skriva stora program i Flowscript.

2. Jag var tvungen att göra omstart av hela Cocoonsystemet efter varje omkompilering av Javakod som anropades från Flowscript:

   Flowscript har direkt åtkomst till alla Java-objekt och funktioner. Detta gör att man enkelt kan flytta ut kod från Flowscript till vanlig Javakod, vilket jag till en början såg som en lösning till problemet i punkt 1. Problem uppstår dock med detta i det att Cocoon inte laddar om Java `.class`-filerna när man gör kodändringar och kompilerar om Javakoden. För att ändringarna skall ”ta” krävs en omstart av hela Cocoon-systemet. Detta upplevde jag som alltför omständligt och tidsödande eftersom det kan ta lång tid för Cocoon-systemet att starta (det tar nästan en minut för Cocoon att starta på min dator med 1GHz Pentium III.)

Om man väljer att starta Cocoon som en servlet i någon Java servlet container (t.ex. Tomcat, se [7]) kan man låta servlet-containern se till att Java `.class`-filerna laddas om ifall dom ändras (dvs. efter det att dom kompilerats om). Detta är en lösning till punkt 2 ovan, men det leder till andra problem, som t.ex. att ens program inte längre enkelt kan köras i en debugger.

För att lösa dessa problem bestämde jag mig för att bryta ut all affärslogik (dvs. allt som tillhör MVC:s model) och lägga den i en separat RMI-server. På detta sätt kunde jag göra en tydlig uppdelning mellan flödeslogiken (MVC controller-komponenten) och affärslogiken (MVC model-komponenten) med ett väldefinierat RMI-gränssnitt emellan. MVC-controllern kunde jag skriva helt i Flowscript samtidigt som jag höll dess komplexitet i schack genom att flytta ut all affärslogik till Javakod i RMI-servern.

Denna uppdelning visade sig fungera bra. Kodändringar i MVC-controllern kan göras ”on the fly” eftersom ändringar i Flowscript-koden ”tar” direkt. Cocoon uppdateras automatiskt när Flowscript-koden ändas. När kodändringar görs i affärslogiken (MVC-model-komponenten) behöver man bara kompilera och starta om RMI-servern (vilket inte tar lång tid). Bara när RMI-gränssnittet ändras måste man starta om både Cocoon och RMI-servern. Tanken är dock att detta inte skall behöva ske så ofta.

4.3.3 Svårlästa XSP-sidor

Som en del i Cocoon finns ett page-tagging-språk som kallas *Extensible Server Pages* (XSP) som på många sätt liknar JSP. En XSP-sida är ett XML-dokument med inbäddad Javakod. XSP är i Cocoon implementerat i form av en pipeline-generator-komponent vid namn ”serverpages”.

För att få Cocoon att visa en XSP-sida kan man i filen `sitemap.xmap` definiera en pipeline enligt följande exempel:

```
<map:pipeline>
  <map:match pattern="*.html">
    <map:generate type=”serverpages” src=”{1}.xsp”/>
    <map:transform type=”xslt” src=”document2html.xsl”/>
    <map:serialize type=”html”>
  </map:match>
</map:pipeline>
```

Figur 7 Exempel på Cocoon-pipeline för att hantera XSP-sidor.

Eftersom jag tidigare hade arbetat mycket med JSP och andra page-tagging-baserade språk försökte jag använda Cocoon på samma sätt; dvs. jag stoppade in Javakod för affärslogik och flödeslogik i XSP-sidorna. Från början såg jag Cocoon som en slags utökad version av JSP där all logik skulle finnas i systemets XSP-sidor. Jag betraktade XSP-sidorna som själva kärnan i applikationen. Jag försökte bygga systemet utifrån dom grundläggande sidorna som jag hade identifierat att systemet skulle bestå av.

När systemet blev större började jag råka ut för samma problem som brukar uppstå i system baserade på JSP: mina XSP-sidor blev allt mer svåröverskådliga och eftersom programmet var uppdelat på olika sidor var det svårt att följa flödet genom programmet. Jag identifierade problemet med att mina XSP-sidor var svåröverskådliga till att ha två orsaker:

1. Det är inte förenligt med Model2/MVC arkitekturen att lägga in kod för affärslogik i webbsidorna. Den enda logik (kod) som skall finnas i MVC-vyn (dvs. webbsidorna) är presentationslogik. Jag insåg att mitt sätt att bygga systemet inte var förenligt med MVC och att det som saknades i mitt system var en riktig MVC-controller.
2. Det sätt på vilket page-taggning fungerar gör XSP-sidorna svårlästa: Poängen med page-taggande språk är att man kan stoppa in programkod med t.ex. Java syntax i ett dokument som har XML-syntax. Men detta sätt att mixa olika syntaxer bidrar till att göra koden mer svårläslig.

Lösningen på orsaken till punkt 1 ovan var att sluta betrakta XSP-sidorna som den centrala delen i applikationen. XSP-sidorna är bara vyer som skall visa data för användaren; men datat som skall visas skall inte hämtas, ändras eller på annat sätt behandlas av XSP-sidan. MVC-controllern skall ”servera” XSP-sidan all data som behövs för vyn, och XSP-sidan skall bara innehålla enkel presentationslogik för att läsa av datat och skriva ut den. Hur detta gjordes beskrivs i kapitel 5.2.1.

För att åskådliggöra problemet i punkt 2 ovan följer här ett exempel: Nedan visas ett exempel på en XSP-sida som genererar ett antal HTML-länkar utifrån ett Java `ArrayList`-objekt som sidan tar som parameter från ett tänkt Flowscript-program som visar sidan med hjälp av *showPageAndWait()*-funktionen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsp:page language="Java" xmlns:jpath=http://apache.org/xsp/jpath/1.0>
<xsp:logic>
for(int i=0; i&lt;((ArrayList)<jpath:value-of select="linkList"/>).size(); i++) {
    <xsp:content>
        <a>
            <xsp:attribute name="href">
                <xsp:expr>
                    ((ArrayList)<jpath:value-of select="linkList"/>).get(i)
                </xsp:expr>
            </xsp:attribute>
            Link<xsp:expr>i</xsp:expr>
        </a>
    </xsp:content>
}
</xsp:logic>
</xsp:page>
```

Figur 8 Exempel på XSP-dokument

Det är i mitt tycke inte så lätt att se vad denna XSP-sida gör. Koden blir svårläst på grund av alla extra XML-taggar som måste till för att bädda in Javakoden i HTML-kod. Det finns i detta enkla exempel inte mindre än tre olika språk inblandade: XML, HTML och Java! Det gäller att hålla tungan rätt i mun för att de tre språkens olika syntax skall bli korrekt och kunna samsas. Man kan se hur de olika ”världarna” påverkar varandra:

XML-syntaxen gör att man tvingas skriva vissa Java-tecken i escaped form: mindre-än tecknet ’<’ måste skrivas ’<’ för att inte störa XML-parsern.

Den dynamiska Javakoden påverkar hur HTML-attributet ”href” skrivs. Man kan inte lägga in Javakod innanför citationstecken; därför måste ”href”-attributet genereras med hjälp av en separat XSP-tagg så att Javakoden som genererar länksträngen kan läggas utanför citationstecknen.

Det blir inte lättare av att XSP-dokument kompileras till Javakod innan dom visas, och om man gjort något syntaktiskt fel i koden kommer felmeddelandet om detta att referera till den maskingenererade Java-filen snarare än till XSP-filen. Den maskingenererade Javakoden skapas i en tempkatalog långt nere i Cocoons filträd, och ingen normal programmerare vill befatta sig med den.

4.3.4 Lättlästa Velocity-sidor

Eftersom jag inte gillade XSP började jag leta efter en ersättare. Cocoon levereras som standard med en komponent som kallas *Velocity Template Engine* (bara Velocity i fortsättningen). Liksom XSP är Velocity implementerat i form av en Cocoon-pipeline-generator-komponent. För att få Cocoon att visa en Velocity-sida kan man definiera en pipeline enligt följande exempel:

```
<map:pipeline>
  <map:match pattern="*.html">
    <map:generate type=”velocity” src=”{1}.vm”/>
    <map:transform type=”xslt” src=”document2html.xsl”/>
    <map:serialize type=”html”>
  </map:match>
</map:pipeline>
```

Figur 9 Exempel på Cocoon pipeline för att hantera Velocity-sidor.

Velocity är väldigt enkelt till sin uppbyggnad. Det enda man kan göra med Velocity är att hämta data från Java-objekt, göra loopar och villkorssatser. Det visar sig dock att detta räcker gott och väl för den presentationslogik webbsidorna behöver.

Nedan visas ett exempel på ett Velocity-dokument som genererar exakt samma HTML-kod som XSP-exemplet i Figur 8.

```
#set($i=0)
#foreach($link in $linkList)
    <a href=”$link”>Link$i</a>
    #set($i = $i + 1)
#end
```

Figur 10 Exempel på Velocity-dokument

I detta exempel tycker jag att det är ganska lätt att se vad den resulterande sidan kommer att innehålla: En lista med URL-länkar. Länkarna har skickats från Flowscript *sendPageAndWait()* i parametern $linkList.

Slutsatsen blev att jag började använda Velocity istället för XSP. Eftersom Velocity är ett så begränsat språk jämfört med Java kan det inte användas för att implementera affärslogik. Men detta är precis vad MVC-modellen säger: I vyn skall endast presentationslogik finnas. Affärslogik och flödeslogik skall inte finnas där. Det är ”förbjudet” för en vy att på egen hand hämta in data från någon extern källa. Den enda data som vyn har att arbeta med skall komma som parametrar från MVC-controllern.

5 Resultat: WPFC och samlade erfarenheter

Resultatet av examensarbetet är applikationen WPFC och de erfarenheter jag fick från implementationen av detta system. Detta kapitel börjar med att ge en beskrivning av hur WPFC fungerar tekniskt och sedan följer ett antal punkter om de erfarenheter jag skaffat mig under arbetet med att implementera WPFC.

5.1 Systembeskrivning av WPFC

Från användarens synvinkel består WPFC-systemet av två delar: WPFC publik webb och WPFC kundwebb. Kundwebben används av Bytbil.com's kunder, dvs. bilhandlare runt om i Sverige som prenumererar på tjänsten hos Bytbil.com att kunna annonsera ut bilar till försäljning på Bytbil.com's publika webb. Den publika webben utgörs av den del som allmänheten kan se och surfa in på för att söka efter bilannonser som bilhandlarna har lagt in.

Tekniskt kan WPFC-systemet delas in i tre huvuddelar: dom två webbsystemen publik- och kundwebb samt WPFC server. Den publika webben och kundwebben är implementerade i Cocoon medan WPFC-servern är en Java RMI-server med underliggande JDBC-databas. Anledningen till att jag implementerat WPFC-servern som en fristående RMI-server beror på de problem med att skriva affärslogiken i Cocoon som jag beskriver i kapitel 5.2.3.

5.1.1 WPFC-server

WPFC-servern är ett vanligt fristående Java-program som innehåller en mängd funktioner för att hämta och spara information om annonser, kunder och medlemmar i JDBC-databasen. WPFC-servern publicerar ett RMI-gränssnitt med vars hjälp klienter kan ansluta till servern för att få tillgång till dess funktionalitet.

WPFC-servern publicerar egentligen två olika RMI-gränssnitt: ett gränssnitt som används endast då en klient ansluter till servern (”loggar in”) samt det RMI-gränssnitt som implementerar själva WPFC-server-funktionaliteten; dvs. det gränssnitt som den anslutna klienten använder för att utföra alla ”nyttiga” funktioner.

Det RMI-gränssnitt som en klient använder för att ansluta till WPFC-servern har jag kallat `LoginServer`. LoginServer består bara av en enda metod:

```
public interface LoginServer extends Remote
{
       public String login(String userId, String password)
           throws  RemoteException;
}
```

WPFC-servern skapar en enda instans av LoginServer; dvs. LoginServer är ett singleton-objekt i programmet. För närvarande används inte parametrarna ”userId” och ”password”: alla klienter tillåts ansluta.

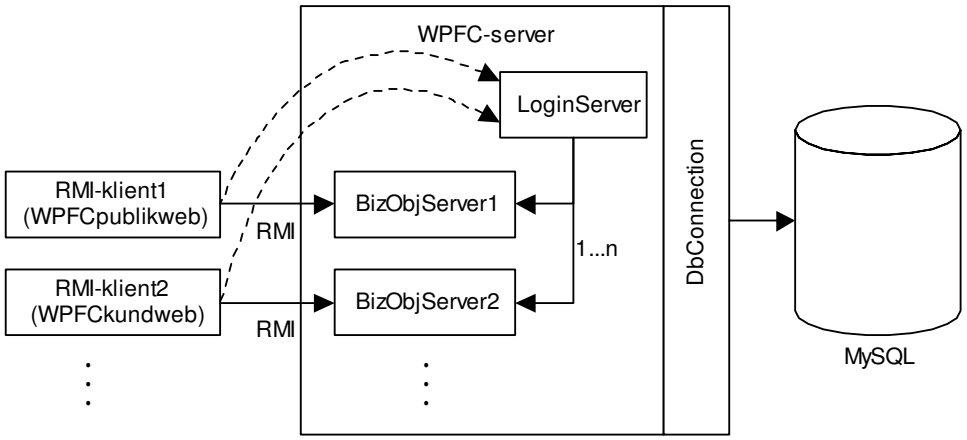
När WPFC-server startar är LoginServer-objektet det enda en klient kan känna till om servern. Klienten anropar login-metoden för att få ett RMI ID till en ”riktig” WPFC-server som jag har kallat `BizObjServer` i koden:

```
public interface BizObjServer extends Remote
{
       public Car getCar(int carId)
           throws RemoteException, CarNotFoundException;
       public int addCar(Car newCar)
           throws RemoteException, CarAlreadyExistsException;
       public void modifyCar(int carId, Car newCar)
           throws RemoteException, CarNotFoundException;

       ... <Förkortat av utrymmesskäl> ...
}
```

Det är BizObjServer-objektet som implementerar själva WPFC-server-funktionaliteten. Dvs. BizObjServer innehåller all affärslogik i systemet. Ett BizObjServer-objekt med unikt RMI ID skapas för varje anslutande klient. På detta sätt kan WPFC-servern hantera flera samtidiga klienter.

Figur 11 visar en schematisk bild över de aktiva objekten som skapas av WPFC-servern när WPFC-klienter är anslutna. De båda anslutna WPFC-klienterna kan tänkas utgöras av WPFC publik webb respektive WPFC kundwebb.



Figur 11 Objektmodell för WPFC-server med anslutna RMI-klienter

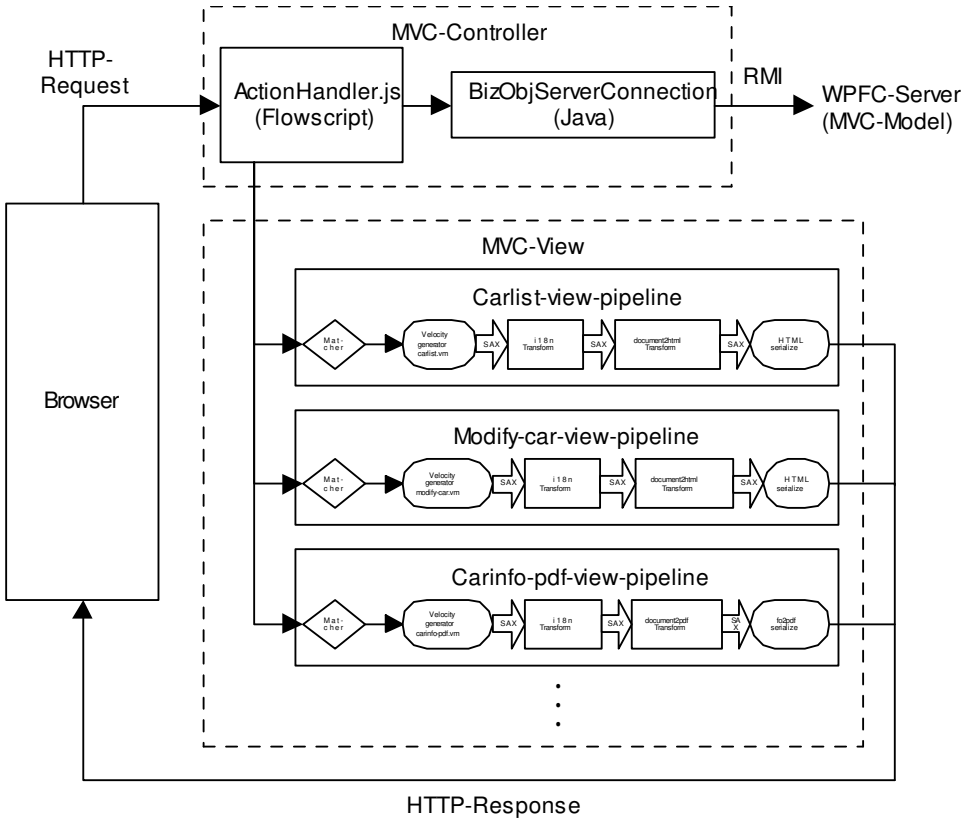
De ovan beskrivna Java-interfacen `LoginServer` och `BizObjServer` utgör WPFC-server RMI-gränssnittet. Om något i dessa interface ändras måste alla klienter uppdateras med den ändringen, dvs. kompileras om. Tanken är att detta inte skall ske ofta. Finessen med detta är att det mesta av WPFC-koden finns i de klasser som implementerar dessa Java-interface: `LoginServerImpl` och `BizObjServerImpl`. Det är *inte* nödvändigt att kompilera om klienterna ifall ändringar sker i implementationsklasserna.

Den uppmärksamme läsaren har noterat att objektet `Car` omnämns i BizObjServer-interfacet. `Car` är ett Java-interface och tillhör också BizObjServer RMI-gränssnittet. Liknande Java-interface finns definierade för alla objekt som WPFC-servern kan hantera: `Car`, `Customer`, `Member`, `Image` etc. Eftersom dom tillhör RMI-gränssnittet gäller att alla klienter måste kompileras om ifall något av interfacen ändras. Men precis som för BizObjServer och LoginServer finns underliggande implementationsklasser för interfacen som man kan ändra utan att klienterna behöver få reda på det. Det är dessa implementationsklasser som hanterar all

kommunikation med den underliggande MySQL databasen (se [8]) via en återanslutande databas-connector-klass som jag kallat `DbConnection`. Med återanslutande menas att klassen försöker återupprätta förbindelsen med databasen ifall den av någon anledning bryts. På så sätt får man ett mer lätthanterligt system eftersom databasen kan stängas av tillfälligt utan att man för den skull behöver starta om WPFC-servern.

5.1.2 WPFC-klienterna Kundwebb och Publik webb

De båda webbapplikationerna WPFC Publik webb och WPFC Kundwebb har likartad arkitektur. Båda är implementerade i Cocoon, båda fungerar som klienter till WPFC-servern och båda är strukturerade enligt MVC-modellen. Nedan visas en schematisk bild av hur klienterna är uppbyggda:



Figur 12 Objektmodell för WPFC Kundwebb (WPFC Publik webb har liknande struktur)

MVC-controllern utgörs av ActionHandler-modulen samt en RMI-server-connector vid namn BizObjServerConnector. BizObjServerConnectorn är ett singelton Java-objekt som hanterar uppkopplingen mot WPFC-servern. All kommunikation mot WPFC-servern går genom BizObjServerConnectorn. När en WPFC-serverfunktion anropas tar BizObjServerConnectorn reda på ifall en förbindelse redan finns uppkopplad, och om så är fallet används denna förbindelse. Ifall ingen förbindelse finns försöker BizObjServerConnectorn öppna en ny förbindelse mot WPFC-servern. Det gör den genom att anropa metoden login() i WPFC-serverns LoginServer-RMI-objekt. LoginServern skapar då en ny BizObjServer så som beskrivits i 5.1.1 och returnerar ett RMI ID till denna. BizObjServerConnectorn tar emot detta RMI ID och upprättar en förbindelse med BizObjServern.

ActionHandlern är helt skriven i Flowscript och består av ett antal funktioner som hanterar flödet i respektive applikation. När användaren klickar på någon länk i webbläsaren dirigeras requestet via `sitemap.xmap` till motsvarande Flowscript-funktion beroende på länkens URL. Det finns två typer av länkar i applikationerna, länkar vars URL har ändelsen ".action" och ".continue":

`.action` Request via länkar som har ändelsen ".action" skickas till motsvarande Flowscript-funktion via en pipeline i `sitemap.xmap` liknande:

```
<map:match pattern="*.action">
      <map:call function="{1}"/>
</map:match>
```

En .action-länk inleder ett nytt flöde för användaren genom programmet, så denna länktyp finns bara på "första sidan" i applikationen.

`.continue` Alla länkar på sidor som visas mitt i ett programflöde har ändelsen ".continue". Dessa request aktiverar motsvarande continuation i Flowscriptkoden. Användaren upplever detta som att programflödet fortsätter från den punk sidan visades. Dvs. från den punkt anropet till *sendPageAndWait()* gjordes för att visa sidan ifråga, och continuation-objektet därmed skapades. Continuation-objektet aktiveras genom en pipeline i `sitemap.xmap` liknande:

```
<map:match pattern="*.continue">
      <map:call continuation="{1}"/>
</map:match>
```

Generellt består varje action-funktion i ActionHandlern av tre delar, eller faser: I den första fasen hämtas relevant data in från WPFC-servern (dvs. från MVC Model-komponenten). I fas två visas den avsedda sidan med det insamlade datat. Detta sker genom ett anrop till *sendPageAndWait()*. Efter det att sidan presenterats för användaren och användaren gör ett nytt request genom att välja någon (.continuation-) länk på sidan, kommer continuationobjektet att aktiveras och exekveringen av funktionen att återupptas och inleda fas tre. Fas tre består i att ta reda på vilken åtgärd användaren tog på sidan, dvs. vilket nytt kommando som togs. Funktionen för att utföra denna åtgärd/detta kommando anropas sedan, och processen börjar om på nytt.

Fas två och tre ligger alltid inbäddade i en loop som visar sidan om och om igen tills användaren väljer åtgärden "exit", eller "gå tillbaks". När användaren gör "exit" från den ytterst omslutande loopen, dvs. i den funktion som anropades via en .action-länk, avslutas funktionen med att visa en "slutsida" för användaren. För att visa slutsidan används Flowscript-funktionen *sendPage()*. Funktionen sendPage fungerar precis som sendPageAndWait, men med skillnaden att ingen continuation skapas, och att funktionen ej blockerar. Funktionen sendPage skickar tillbaks den specificerade sidan i HTTP-responsen till användaren och det flöde som inleddes med <map:call function="...">-anropet avslutas.

När ActionHandlern i fas två anropar *sendPageAndWait()* för att visa den avsedda sidan skickar den med de Java-objekt som den har hämtat från WPFC-servern (MVC model-komponenten) samt continuation-objektet som skapas (continuation-objektet skickas implicit). Motsvarande

Cocoon pipeline anropas då för att visa sidan, och den pipeline-generator som skall visa sidan kan komma åt Java-objekten från MVC-model-komponenten som parametrar.

Den pipeline som visar sidan ingår i, och representerar, MVC-vyn. Jag har i WPFC, av skäl som beskrivs i kapitel 4.3.3 och 4.3.4 uteslutande använt mig av Velocity-generatorn för att generera det underliggande dokumentet till vyerna. Velocity lägger in de dynamiska innehållet i dokumentet utifrån de MVC-model Java-objekt som skickats från MVC controllern. Velocity lägger också in id för det continuation-objekt den fått från controllern i alla länkar som dokumentet innehåller. När det underliggande dokumentet har genererats av Velocity skickas det vidare i pipelinen till i18n-transformatorn som hanterar språköversättning. Om sidan (vyn) skall visas i HTML-format appliceras document2html.xsl på dokumentet som genererar sidans HTML-layout, sedan serialiseras sidan till HTTP-responsen som skickas tillbaks till användarens webbläsare.

5.2 Erfarenheter från implementationen av WPFC

Arbetet med att implementera WPFC gav många insikter om hur man bör gå tillväga för att bygga applikationer i Cocoon, men också erfarenheter om fallgropar och designmässiga snedsteg som man bör undvika. Detta kapitel beskriver de läxor jag lärde mig under arbetet med WPFC.

5.2.1 Separera affärslogik, flödeslogik och presentationslogik

För att undvika att drabbas av problem med att koden för de i webbapplikationen ingående sidorna blir komplicerad och svår att följa visar mitt arbete med WPFC att det är viktigt att man håller isär affärslogik, flödeslogik och presentationslogik. Cocoon hjälper till med detta genom sitt stöd för MVC(*Model-View-Controller*)-modellen.

All flödeslogik skall samlas i en MVC-controller-komponent. Även kallad ”action-hanterare”.

All affärslogik skall placeras i MVC-model-komponenten. Denna består av separata klasser skrivna i ”vanlig” Java, eller som i WPFC är placerad i en separat fristående server.

Dynamiska webbsidor skall endast innehålla enkel presentationslogik. Dom skall absolut inte innehålla kod för att på egen hand hämta in och modifiera data. Det enda data vyerna skall ha tillgång till är dom objekt som sänts till dom via parametrar från MVC-controller-komponenten.

Betrakta inte applikationens enskilda sidor som den centrala delen i applikationen. Webbsidorna är bara vyer som skall visa data för användaren; men datat som skall visas skall inte hämtas, ändras eller på annat sätt behandlas av inbäddad kod i sidorna. MVC-controllern skall ”servera” webbsidan all data som behövs för vyn, och sidan skall bara innehålla enkel presentationslogik för att läsa av datat och skriva ut den.

5.2.2 Skriv flödeslogik på linjärt sätt genom att använda continuations

På grund av webbapplikationers inneboende ”inversion of control”-egenskaper bör man använda continuations i koden för flödeslogiken. Flödeslogiken kan då skrivas på linjärt sätt och man kan i och med detta hålla ihop flödeslogiken på ett ställe. Flödeslogiken behöver inte spridas ut över de i systemet ingående sidorna för att i alla lägen hantera alla möjliga tillstånd som systemet kan befinna sig i.

Tack vare stöd för continuations i Cocoon Flowscript var det möjligt att helt eliminera flödeslogik och affärslogik från MVC-vyerna i WPFC. Flödeslogiken kunde samlas på ett ställe och skrivas på linjärt sätt.

5.2.3 Bryt ut MVC-model-komponenten till separat RMI-server

För att hålla Flowscript-koden som implementerar MVC-controller-komponenten (flödeslogiken) så enkel och tydlig som möjligt, Bryt ut all affärslogik i MVC-model-komponenten till en fristående server. Detta också på grund av praktiska skäl som beskrivs i kapitel 4.3.2.

5.2.4 Använd Velocity snarare än XSP för att generera dynamiska webbsidor

Velocity är ett fantastiskt smart språk att använda till presentationslogiken i webbsidorna. Använd Velocity i stället för det komplicerade och tungarbetade XSP. Din applikation blir tusenfalt lättare att förstå! WPFC använder uteslutande Velocity för alla sidor.

6 Slutsatser

Syftet med detta examensarbete var att visa hur arbetet med att utveckla webbapplikationer underlättas genom att använda ramverket Cocoons speciella egenskaper vad gäller continuations, MVC och XML-pipelinebaserade publiceringssystem.

Queinnec, ref. [4] och [5], har visat hur continuations hjälper till att komma bort från det sidorienterade sättet att skriva webbapplikationer på, och hur man istället kan skriva flödeslogiken i webbapplikationer på traditionellt linjärt sätt. Detta innebär att koden som hanterar programflödet kan skrivas på linjärt sätt. Dvs. koden kan skrivas som om mekanismen att sända ett HTML-formulär till användarens webbläsare och sedan vänta på användarens svar, är ett anrop till en blockerande input-funktion som inte returnerar förrän användaren fyllt i formuläret och gjort submit på detta. Denna rapport åskådliggör skillnaden mellan sidorienterade och linjära applikationer ytterligare genom att ge en praktisk jämförelse mellan en sidorienterad JSP-applikation och motsvarande linjära Cocoon-applikation.

Användandet av Cocoon's continuations ger upphov till ett problem som uppmärksammats i denna rapport: Eftersom continuation-objekt i Cocoon lagras i webbservern under tiden dom är aktiva uppstår problem ifall man vill kunna köra sin webbapplikation på flera servrar parallellt (i syfte att till exempel åstadkomma lastbalansering). Eftersom en webbsida som visas i klientens webbläsare bara har en referens till ett continuation-objekt kan inte ”svaret” i ett webbformulär skickas till en annan server än den server som initierade continuation-objektet. Referensen saknar mening i de andra servrarna eftersom respektive continuation-objekt inte finns definierade där.

För Bytbil.com, som jag implementerade prototypapplikationen WPFC tillsammans med, var det viktigt att få en demonstration av hur uppdelningen mellan data, layout och logik kan åstadkommas i en webbapplikation utvecklad i Cocoon. Rapporten visar hur användandet av MVC (Model-View-Controller)-designmönstret som stöds i Cocoon hjälper till att åstadkomma denna uppdelning.

Likaså var ett viktigt kriterium för Bytbil att det skulle vara enkelt att kunna generera webbsidor med olika utseende och i olika format utifrån samma grunddata, utan att man därför skulle vara tvungen att göra ändringar i koden för flödeslogik och affärslogik. Man ville att systemet t.ex. skulle kunna presentera webbsidor med olika layout och språk beroende på vilken kund och/eller klient som sidan ägdes/visades av. Man ville också se exempel på hur PDF-sidor kunde genereras utifrån samma mekanism som HTML-sidorna genererades ifrån. Denna rapport samt WPFC-prototypen visar hur implementationen av denna funktionalitet underlättas med hjälp av Cocoons XML-pipelinebaserade publiceringssystem.

Jag har med implementationen av WPFC och med exemplen i detta dokument praktiskt visat att Cocoon hjälper till att lösa flera av de svårigheter som man brukar råka ut för i samband med utveckling av webbapplikationer i traditionella webbutvecklingsramverk.

- Användandet av XML/XSLT i Cocoons pieline-publiceringssystem hjälper till att separera data från presentation/layout och affärslogik i WPFC. Det gör det också möjligt att skriva applikationer som kan generera sidor i olika format (t.ex. HTML och PDF) utifrån samma grunddata.

- Genom att bygga WPFC applikationen enligt MVC-modellen kan man separera presentationslogiken, flödeslogiken och affärslogiken från varandra. Men MVC på egen hand hjälper inte till att lösa de problem som webbapplikationers ”inversion of control”-paradigmen medför.

- För att underlätta hanteringen av ”inversion of control” används continuations i MVC-controllerns Flowscript-kod i WPFC. Detta underlättar väsentligt hanteringen av webbläsarens framåt/bakåt/history-funktioner.

7 Referenser

[1] Geary David M.: *Advanced Java Server Pages,* Sun Microsystems Press, 2001

[2] Velocity Template Engine website: http://jakarta.apache.org/velocity

[3] McGuire Tommy M.: *An Infernal Device, Browser-Stored Do-It-Yourself Continuations for Web Programming*, The University of Texas at Austin, 2003

[4] Queinnec Christian: *Inverting back the inversion of control or, Continuations versus page-centric programming*, Université Paris 6— Pierre et Marie Curie, France

[5] Queinnec Christian: *The Influence of Browsers on Evaluators or, Continuations to Program Web Servers*, Université Paris 6— Pierre et Marie Curie, France, 2000

[6] Apache Cocoon website: http://cocoon.apache.org

[7] Jakarta Tomcat website: http://jakarta.apache.org/tomcat

[8] MySQL website: http://www.mysql.com

[9] Krasner Glenn E. och Pope Stephen T.: *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*, ParcPlace Systems Inc. 1988

[10] Moscar Lajos & Aston Jeremy: *Cocoon Developers Handbook*, Sams Publishing, 2003

[11] Mozilla Rhino website: http://www.mozilla.org/rhino

[12] Danvy Oliver: *On Evaluation Contexts, Continuations, and the Rest of the Computation*, Department of Computer Science, University of Aarhus, 2004

[13] C.Strachey, C.P.Wadsworth: *Continuations: a mathematical semantics for handling full jumps*, PRG-11 Oxford University Programming Research Group 1974

[14] Java RMI website: http://java.sun.com/products/jdk/rmi/

Appendix A: Detaljerad beskrivning av Cocoon

Cocoon är ett ramverk för att bygga webbapplikationer i Java. Cocoon drivs ideellt som ett open source-projekt av Apache Software Foundation.

Cocoon är ett open source-projekt som drivs av Apache Software Foundation. Cocoon kan beskrivas som ett ramverk för hur XSLT-transformationer kan användas för att publicera XML-data på önskat sätt (t.ex. som läsbara HTML-dokument). Cocoon är byggt i Java och är ett open source-projekt som drivs av Apache.

1.1 Sitemap

Kärnan i en Cocoon-applikation utgörs av filen `sitemap.xml`. Detta är en XML-fil som beskriver konfigurationen för hela ens applikation. Varje webbapplikation har en egen sitemap, och man kan koppla ihop sitemaps i en trädstruktur där sitemaps ärver sin respektive förälders egenskaper. Egenskaperna kan dock överridas om så önskas.

1.2 Cocoon pipelines

Precis som en vanlig webbserver är Cocoons uppgift att hantera inkommande HTTP requests, behandla dessa requests och sända en HTTP-respons tillbaka till klienten för varje request. Cocoon behandlar ett inkommande request genom att söka upp en matchande *pipeline* och utföra de åtgärder som anges där för att generera ett resultat som sedan skickas som respons. Alla pipelines som ingår i ens webbapplikation definieras i `sitemap.xmap`-filen.

1.2.1 Gundkomponenter i pipelines

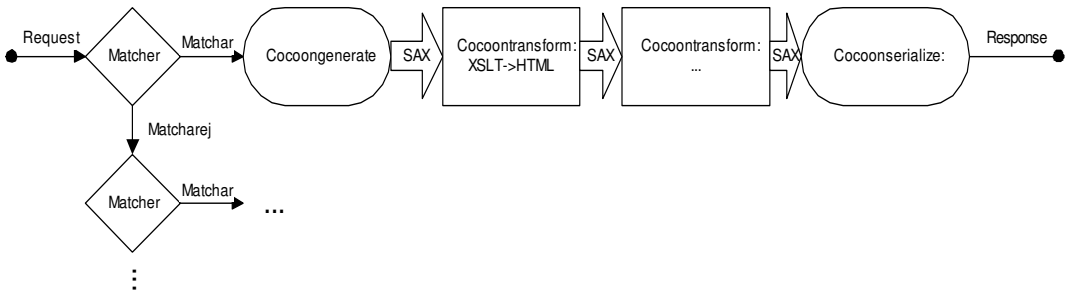
En Cocoon pipeline byggs upp utifrån fyra grundkomponenter som exekveras i nämnd ordning:

Request-matcher. Denna komponent används för att matcha det inkomna requestet mot en viss pipeline. Den första pipeline som matchar requestet exekveras.

Generator. En generator skapar en ström av XML SAX-events som skickas genom pipelinen. Exempel på en enkel generator är Exempel på generatorer Generatorn SAX-eventen utifrån en angiven datakälla. Datakällan kan t.ex. vara en fil eller en databas.

Transformer. Den genererade strömmen av SAX-events skickas igenom en eller flera angivna transformeringar för att skapa en ny ström av SAX-events. Ett exempel på en transformation är att applicera ett XSLT-stylesheet på SAX-strömmen för att skapa t.ex. HTML-kod.

Serializer. Den genererade och transformerade strömmen av SAX-events serialiseras till HTTP-response-objektet.



Figur 13 Schematisk bild av en Cocoon pipeline

Alla pipelines består minst av en generator och en serializer. Första operationen i en pipeline måste vara en generator, och den sista operationen skall vara en serializer. Där emellan kan finnas ett godtyckligt antal transformationer.

1.2.2 Specialkomponenter i pipeline

Förutom de ovan nämnda grundkomponenttyperna *matcher*, *generator*, *transformer* och *serializer*, finns några andra komponenttyper som kan stoppas in i en pipeline. Exempel på sådana komponenter är:

Actions. Denna komponent kan liknas vid en `if...then...`-sats.

Aggregering. Komponent för att slå ihop XML SAX-strömmarna från flera andra pipelines till en ström.

Custom: Man kan skriva egna komponenter i Java som man kompilerar och länkar in i Cocoon.

1.2.3 Exempel-pipeline XML->HTML

Nedan följer ett enkelt exempel för att åskådliggöra hur Cocoons pipelines fungerar och för att visa hur data och presentationsinformation kan hållas åtskiljda med hjälp av XML/XSL:

Antag att vi har ett system som hanterar felrapporter. En felrapport som lagts in i systemet lagras i XML-format enligt följande:

```
<?xml version="1.0"?>
<troubleticket>
  <id>123</id>
  <location>facility 7</location>
  <map>facility7</map>
  <description>
    Unit 14 at facility 7 is malfunctioning
  </description>
</troubleticket>
```

Vi antar för enkelhets skull att felrapporten lagras i en fil med namn `ticket123.xml`.

För att visa informationen om denna felrapport i HTML-format finns i filen `ticket2html.xsl` ett XSL stylesheet som ser ut så här:

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="troubleticket">
    <h1>Trouble ticket id: <xsl:value-of select="id"/></h1>
    <p>Description: <xsl:value-of select="description"/></p>
    <p>
      <img>
        <xsl:attribute name="src">
          <xsl:value-of select="map"/>.jpg
        </xsl:attribute>
      </img>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

Detta kommer i en HTML-webbläsare att visa felrapporten med felrapportens id som överskrift, följt av beskrivningen av felet, samt en .jpg-bild som antas visa en karta över hur man tar sig till den plats som felrapporten gäller.

För att få vår webbsite som kör Cocoon att visa denna HTML-sida när man anger `http://hostname/troubleticket123.html` i webbläsaren skall vi definiera en Cocoon pipeline enligt följande i Cocoons `sitemap.xmap`-fil:

```
<map:pipeline>
  <map:match pattern="troubleticket*.html">
    <map:generate type=”file” src=”ticket{1}.xml”/>
    <map:transform type=”xslt” src=”ticket2html.xsl”/>
    <map:serialize type=”html”>
  </map:match>
</map:pipeline>
```

När Cocoon får in ett HTTP-request med en URI som matchar ”troubleticket*.html” kommer Cocoon att exekvera denna pipeline. Vi använder här wildcard-matchern som är default i Cocoon. Tecknet ”*” i matcherns ”pattern”-parameter matchar noll eller flera godtyckliga tecken.

I satsen <map:generate> kommer Cocoon att anropa generatorn av typen ”file” som läser in den i parametern ”src” specificerade XML-filen och generera XML SAX-events för alla XML-taggar i denna fil som den skickar vidare till nästa steg i pipelinen. Matcharen ersätter beteckningen ”{1}” med värdet av den första wildcard-strängen som angetts i matcharens pattern, dvs. i detta fall strängen ”123”.

Vår felrapport skickas som XML-data till en transformer av typen ”xslt” som kör XML-datat genom det i parametern ”src” specificerade XSL-stylesheetet, dvs. vår ticket2html.xsl-fil. Den ursprungliga strömmen av XML SAX events innehållande data för felrapporten i filen ticket123.xml omvandlas nu till en ström av XML SAX-events som beskriver felrapporten i HTML-format.

Strömmen av XML SAX-events med beskrivningen av felrapporten i HTML-format skickas vidare till en serializer av typen ”html”. Denna skriver in HTML-koden i HTTP-responsen som därefter skickas tillbaks till webbläsaren och visas för användaren.

Händelseförloppet är dock inte riktigt slut ännu: i HTML-koden för vår felrapport finns en referens till en .jpg-bild i satsen: <img src=”facility7.jpg”>. När webbläsaren skall visa denna bild kommer den att göra ett nytt HTTP-anrop mot vår Cocoon webbserver för att ladda ner själva bilddatat. Vår webbserver kommer att få ta emot adressen ”http://hostname/facility7.jpg”. För att få detta att fungera måste vi tala om för Cocoon var bilden facility7.jpg finns och hur den skall skickas. Detta görs med hjälp av ytterligare en pipeline i Cocoons sitemap.xmap-fil:

```
<map:pipeline>
  <map:match pattern="*.jpg">
    <map:read type="resource"
       src="images/{1}.jpg"
       mime-type="image/jpeg"/>
  </map:match>
</map:pipeline>
```

Denna pipeline kommer att matcha vår URI ”facility7.jpg” och ladda ner den önskade bilden till webbläsaren. Vi har här till synes frångått regeln som säger att alla Cocoon pipelines måste börja med en generator och sluta med en serializer. Detta på grund av att komponenten <map:read> är en kombinerad generator och serializer i ett och samma objekt. En reader används när en resurs (t.ex. en fil) bara snabbt skall skickas som den är utan förändring till klienten. Vi har här för enkelhets skull antagit att bildfilen facility7.jpg finns i katalogen ”images”.

1.2.4 Generering av dynamiska sidor

Föregående exempel använder endast statiska sidor. Det finns dock flera olika sätt på vilket dynamiska sidor kan skapas. För att generera dynamiska sidor byter man bara ut ”file”-generatorn mot en generator som kan innehålla presentationslogik. Exempel på generatorer som kan generera dynamiska dokument är:

`serverpages`	Denna generator genererar ett dokument utifrån en XSP-sida. XSP är en förkortning för ”eXtensible Server Pages”. XSP är ett *page-tagging* språk liknande JSP. En XSP-sida är ett XML-dokument med inbäddad Javakod. XSP beskrivs närmare i kapitel 1.
`jsp`	Generatorn ”jsp” genererar dynamiska dokument utifrån en JSP-sida.
`php`	Generatorn ”php” genererar dynamiska dokument utifrån en PHP-sida.
`velocity`	Denna generator implementerar template-språket Velocity. Velocity beskrivs närmare i kapitel 1.3.3.

Det finns många fler generatorer än de ovan nämnda för att generera dynamiska dokument. Fler än vad som ryms att beskriva här. Som tidigare nämnts kan man också skriva egna generatorer i Java ifall dom som finns inte passar.

1.3 Cocoon Flowscript

En speciell pipelinekomponent är <map:call>. En pipeline som använder <map:call> kan se ut så här:

```
<map:pipeline>
  <map:match pattern="*.action">
    <map:call function="{1}"/>
  </map:match>
</map:pipeline>
```

Precis som för <map:read> gäller att ingen generator eller serializer skall anges i en sådan pipeline eftersom <map:call> förväntas göra allt som en generator och serializer normalt gör. Komponenten <map:call> används till att göra anrop till programkod skriven i *Flowscript*. I exemplet ovan kommer Flowscript-funktionen *AddTroubleTicket()* att anropas ifall man i webbläsaren anger adressen `http://hostname/AddTroubleTicket.action`. Cocoon kräver att denna funktion skall returnera en HTTP-respons till klienten (dvs. funktionen måste göra det som en serializer normalt sköter).

Som vi ser i kapitel 2.3 används Flowscript till att skriva *controllern* i en webbapplikation. Vad en controller är beskrivs detaljerat i kapitel 2.2, men förenklat kan controllern beskrivas som det program som definierar flödet i en webbapplikations användargränssnitt. Controllern tar emot ”actions”, dvs. alla HTTP-requests, som uppstår när användaren följer länkar eller gör submit på formulär bland applikationens webbsidor. Controllern ser till att varje action behandlas på avsett sätt och skickar sedan tillbaks en HTTP-respons till klientens webbläsare.

Flowscript är en version av JavaScript som implementerats i Java. Javaimplementationen av JavaScript kallas Rhino [11] och Flowscript bygger på en version av Rhino som utökats med att kunna använda *continuations*. Eftersom JavaScript oftast förknippas med program som körs i klientens webbläsare bör det förtydligas att Flowscript körs på servern.

För inbitna Javaprogrammerare kan det verka som att gå ett steg tillbaka att börja skriva sina webbapplikationer i JavaScript istället för Java. Anledningen till att JavaScript valts som grund för Flowscript är dock kriteriet att språket skall ha stöd för continuations. Java VM stöder inte continuations men däremot går det utmärkt att i Java implementera ett nytt språk (Flowscript) som har inbyggt continuation-stöd!

Den som trots allt är skeptisk mot JavaScript kan trösta sig med att Flowscript, tack vare att det är implementerat i Java, har direkt åtkomst till alla Java-objekt och funktioner – inklusive alla vanliga Javabibliotek etc. Om man inte gillar JavaScript kan man flytta merparten av all logik till vanliga Javaklasser och på så sätt hålla Flowscript-programmen väldigt ”tunna”. Jag har i min implementation av WPFC visat hur detta kan göras. Som beskrivs i kapitel 5.2.3 har jag tagit det hela ännu ett steg längre genom att flytta all affärslogik till en separat Java RMI-server och därmed kunnat låta mina Flowscript-program bara innehålla kod som hanterar flödet i applikationen.

Det mest intressanta med Flowscript är att det till skillnad från andra vanligt förekommande språk för webbutveckling (t.ex. JSP, ASP, PHP etc.) har stöd för continuations. Continuations beskrivs närmare i kapitel 3.1. McGuire [3] beskriver hur man på egen hand kan skapa stöd för continuation-liknande funktionalitet även i språk som inte har direkt stöd för continuations (t.ex. JSP och PHP) men i detta dokument bortses från detta.

1.4 Dokumentpublicering i Cocoon

Detta avsnitt ger några exempel på hur man kan använda Cocoons XML-pipeline-publiceringssystem för att generera sidor med olika utseende och/eller sidor av olika typ utifrån samma underliggande data och affärslogik.

1.4.1 Exempel-pipeline XML->WML, XML->PDF

Om man vill att det skall vara möjligt att via WAP koppla upp sig mot felrapporteringssystemet som beskrevs i exemplet i 0 och t.ex. kunna läsa felrapporter med sin mobiltelefon, kan detta ganska enkelt göras genom att man skapar ett nytt XSL-stylesheet som översätter vår felrapport till WML:

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="/">
    <wml>
      <card>
        <xsl:apply-templates/>
      </card>
    </wml>
  </xsl:template>
  <xsl:template match="troubleticket">
    <p><big>Trouble ticket id: <xsl:value-of select="id"/></big></p>
    <p>Description: <xsl:value-of select="description"/></p>
    <p>
      <img>
        <xsl:attribute name="src">
          <xsl:value-of select="map"/>.wbmp
        </xsl:attribute>
      </img>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

Vi antar att denna XSL finns i filen `ticket2wml.xsl`.

När en WML-klient begär en felrapport kommer requestet att se ut som `http://hostname/troubleticket123.wml`. Vi talar om för Cocoon hur en URI på formen troubleticket*.wml skall hanteras genom att lägga till en ny pipeline i `sitemap.xmap`:

```
<map:pipeline>
  <map:match pattern="troubleticket*.wml">
    <map:generate type="file" src="ticket{1}.xml"/>
    <map:transform type="xslt" src="wml2html.xsl"/>
    <map:serialize type="wml">
  </map:match>
</map:pipeline>
```

I WML brukar bildformatet `.wbmp` användas i stället för `.jpg` eller `.gif`. I detta fall kommer WAP-browsern att efterfråga bilden facility7.wbmp i stället för facility7.jpg. Vi måste därför ange en pipeline även för detta filformat:

```
<map:pipeline>
  <map:match pattern="*.wbmp">
    <map:read type="resource"
       src="images/{1}.wbmp"
       mime-type="image/wbmp"/>
  </map:match>
</map:pipeline>
```

samt naturligtvis se till att bildfilen `facility7.wbmp` faktiskt finns i `images`-katalogen.

Om man förutom HTML- och WML-formaten vill kunna få en sida för utskrift av felrapporten kan man låta Cocoon generera en PDF eller PostScript-fil utifrån XML-datat. Detta gör man på samma sätt som beskrivits i detta exempel för WML, med skillnaden att XSL-filen skall generera en sida i XSL:FO-format. XSL:FO är ett format som ingår i specifikationen av XSL och definierar ett antal XML-taggar som används för att beskriva utseendet/presentationen av en sida. FO står för *Formatting Objects*. I Cocoon ingår en serializer som kan skapa ett PDF- eller PostScript-dokument av en XSL:FO-sida. Förutom den XSL:FO-genererande XSL-filen behövs en Cocoon pipeline som matchar URI ”troubleticket*.pdf” och använder en serializer av typen ”fo2pdf”. Se vidare [10] för mer info.

Detta exempel visar att det med hjälp av Cocoons XML/XSL pipelinehantering relativt enkelt går att anpassa en applikation till att generera utdata på flera olika format utan att underliggande data behöver ändras.

1.5 Dynamisk anpassning av layout

Målet i detta avsnitt är att ge svar på frågan ”Hur klarar Cocoon kundanpassning av layouten på hela eller delar av systemet?” som ställdes i kapitel 1.2.

I WPFC-systemet har jag inte implementerat dynamisk anpassning av layout beroende på vilken kund/användare som loggat in. Systemet är dock förberett för detta och det enda som krävs för att få det att fungera är egentligen att skapa nya document2html.xsl XSL-stylesheets för varje kund/användare.

I Cocoons sitemap kan man komma åt bl.a. sessionsvariabler genom att använda notationen `{session:<variabelnamn>}`. Genom att sätta t.ex. sessionsvariabeln ”customer” till namnet på den kund som för tillfället är inloggad kan man få den pipeline som genererar alla HTML-sidor i systemet att använda olika `document2html.xsl` beroende på vilken kund som är inloggad. I WPFC är denna pipeline definierad enligt:

```
<map:pipeline>
  <map:match pattern="show-*.vm">
    <map:generate type="velocity" src="views/{1}.vm"/>
    <map:transform type="xslt"
src="{session:customer}/styles/document2html.xsl"/>
    <map:serialize type="html"/>
  </map:match>
</map:pipeline>
```

Detta gör att olika `styles/*`-kataloger kommer att användas för olika kunder (och därmed att olika `document2html.xsl` XSL-stylesheet kommer att användas). Katalogen `styles` innehåller all layoutinformation som WPFC-systemet använder; dvs. alla XSL-stylesheets och alla bilder etc.

1.6 Flerspråksstöd i Cocoon

Detta avsnitt beskriver hur Cocoons XML-pipeline-publiceringssystem kan anpassas för att kunna generera sidor i olika språk utifrån samma underliggande data och affärslogik, men beroende på användarens locale.

Cocoon har stöd för att bygga applikationer som hanterar fler än ett språk via en transformer pipelinekomponent som kallas ”i18n”. Termen i18n är härledd ur ordet *internationalization* som har 18 bokstäver mellan det första *i*:et och det sista *n*:et.

Om man vill ha flerspråksstöd i sin applikation skall man inkludera denna transformer i sin pipeline enligt följande exempel:

```
<map:pipeline>
  <map:match pattern="*">
    <map:generate type=”file” src=”{1}.xml”/>
    <map:transform type=”i18n”>
      <map:parameter name="catalogue-name" value="catalog"/>
      <map:parameter name="catalogue-location" value="translations"/>
    </map:transform>
    <map:transform type=”xslt” src=”document2html.xsl”/>
    <map:serialize type=”html”>
  </map:match>
</map:pipeline>
```

Parametern “catalogue-name” anger filnamnet på de *message catalogs* (”ordlistor”) som i18n-transformern använder för att översätta ord och fraser till det språk som önskas. I ovanstående exempel skulle den message catalog som används som default heta `catalog.xml`. En message catalog för det svenska språket skulle heta `catalog_se.xml`. För tyska skulle den heta `catalog_de.xml` osv.

Parametern ”catalogue-location” talar om för Cocoon i vilken filsystemskatalog applikationens message catalogs finns lagrade.

En användare specificerar vilket språk som önskas genom att ange parametern ”locale” i sina HTTP-requests mot servern. Om användaren vill ha sina sidor på tyska skall en adress liknande `http://hostname/somefile?locale=de` anges. Om parametern ”locale” inte anges kommer den message catalog som används som default att användas; dvs. `catalog.xml`.

En message catalog är en XML-fil som innehåller en key/value-tabell enligt följande format:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<catalogue xml:lang=”se”>
  <message key=”hello”>Hej</message>
  <message key=”goodbye”>Adjö</message>
  ...
</catalogue>
```

Man måste skapa en message catalog-fil för varje språk som skall stödjas. Ovanstående fil skulle med andra ord ha filnamnet `catalog_se.xml` medan en message catalog för tyska (där attributet `xml:lang="de"`) skulle ha filnamnet `catalog_de.xml`.

1.6.1 Översättning av enstaka ord

För att få en sida att använda flerspråksstöd skall man använda XML-taggen <i18n:text> samt inkludera XML *namespace* `http://apache.org/cocoon/i18n/2.0`. För att till exempel översätta ordet ”Hello” till svenska skall man skriva sin sida på följande sätt:

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns:i18n="http://apache.org/cocoon/i18n/2.0">
  <para><i18n:text>hello</i18n:text></para>
</page>
```

Om klienten frågar efter denna sida med ”locale” parametern satt till ”se” kommer i18n-transformern att slå upp nyckeln ”hello” i message katalogen `catalog_se.xml` och ersätta strängen ”hello” med ”Hej”.

1.6.2 Översättning av hela stycken

I många fall vill man kunna översätta hela meningar eller fraser; det är då obekvämt att använda hela frasen som nyckel i message catalog-tabellen. Istället kan man använda ett kort nyckelnamn för att identifiera frasen i message catalog-tabellen, t.ex. ”message1”:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue xml:lang="se">
  <message key="message1">
       Det var en gång...
  </message>
  ...
</catalogue>
```

För att referera till denna fras på sin sida kan man använda attributet `i18n:key` till taggen `<i18n:text>` enligt följande exempel:

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns:i18n="http://apache.org/cocoon/i18n/2.0">
  <para><i18n:text i18n:key="message1">
    Text to be displayed if the key message1 is missing in the message
catalog
  </i18n:text></para>
</page>
```

1.6.3 Parameteriserad översättning

Det kan hända att långa meningar som skall översättas innehåller parametrar. Antag att vi vill visa felrapportens id samt hur många felrapporter som finns registrerade i felrapporteringssystemet som beskrevs i 0. Vi vill att sidan skall visa meddelandet ”Visar felrapport id 123, totalt 27 felrapporter i systemet” i den svenska översättningen.

För att åstadkomma detta använder man taggarna <i18n:translate> och <i18n:param> i sitt dokument enligt följande:

```
<i18n:translate>
  <i18n:text i18n:key="noOfTroubleTicketsText">
    Translation is missing
  </i18n:text>
  <i18n:param>123</i18n:param>
  <i18n:param>27</i18n:param>
</i18n:translate>
```

i18n:text-taggen gör att i18n-transformern slår upp meddelandet med nyckeln "noOfTroubleTicketsText" i `catalog_se.xml`. Detta meddelande är parameteriserat enligt följande:

```
<message key="noOfTroubleTicketsText">
  Visar felrapport id {0}, totalt {1} felrapporter i systemet
</message>
```

i18n-transformern ersätter parametrarna {0} och {1} med de värden som angetts i respektive `<i18n:param>`-tagg i dokumentet.

1.6.4 Översättning av attribut

Ibland är det nödvändigt att översätta XML-attribut. Detta åstadkommer man genom att använda attributet `i18n:attr` i det XML-element som innehåller det attribut man vill översätta. Antag att vi på vår sida har en submit-knapp vars label vi vill ha översatt. Vi kan då definiera denna knapp med satsen:

```
<input type="submit" value="button1text" i18n:attr="value">
```

i18n-transformern kommer då att använda värdet på det attribut som specificeras i `i18n:attr` (dvs. "value" attributets värde "button1text") som nyckel till message katalogen för att slå upp den text som skall visas i dess ställe.

1.6.5 Översättning av datum och tid

i18n-transformern har stöd för att ange datum och tid på rätt sätt för olika språk. För detta används taggarna `<i18n:date>`, `<i18n:time>` och `<i18n:date-time>`. Dessa taggar använder ett antal attribut för att beskriva hur datum och tid skall översättas:

`value`	Anger det datum/den tid som skall visas.
`src-pattern`	Anger mönstret på datumet/tiden i value attributet. Exempelvis "dd-mm-yyyy" eller "FULL", "LONG", "MEDIUM" och "SHORT". Se Java API definitionen för klassen `DateFormat` för information om vad värdena betyder.
`pattern`	Anger mönstret för hur datumet/tiden skall visas.
`src-locale`	Anger mönstret på datumet/tiden i value attributet utifrån en fördefinierad

	locale. Exempelvis ”en_US”, ”en_UK”.
`locale`	Anger mönstret för hur datumet/tiden skall visas utifrån en fördefinierad locale.

Man kan se det som att attributen `src-pattern` och `pattern` anger längden på hur datumet skrivs: dvs. Om det skall skrivas ”2004-03-07” eller ”Söndagen den 7 mars 2004”. Attributen `src-locale` och `locale` anger vilket språk som skall användas för det valda mönstret i `pattern`. Om man i sitt dokument har elementet:

```
<i18n:date locale="se" pattern="FULL"
  src-locale="en_US" src-pattern="SHORT"
  value="03/07/2004"/>
```

kommer strängen ”Söndagen den 7 mars 2004” att skrivas ut. För fler exempel och mer detaljerad förklaring, se [10].

1.6.6 Översättning av nummer och valutor

Även nummer och valutor kan översättas på liknande sätt som för datum och tid. För detta används taggen `<i18n:number>`. Precis som för `<i18n:date>` kan man ange `src-pattern`, `pattern`, `src-locale` och `locale`-attribut som specificerar hur översättningen skall ske. Dessutom finns attributet `type` som specificerar vilken typ av nummer som anges som värde i `<i18n:number>` taggens `value` attribut. De nummertyper som hanteras är:

`number`	i18n-transformern hanterar numret som ett vanligt tal. T.ex. 2 143,5 (locale=”se”) eller 2,143.5 (locale=”en_US”)
`currency`	Numret hanteras som en valuta. T.ex. $27.24. Automatisk valutaomräkning hanteras inte.
`Currency-no-unit`	Numret hanteras som en valuta. T.ex. 27.24
`percent`	Numret hanteras som en procentsats. T.ex. 120%

För fler exempel och mer detaljerad förklaring, se [10].

1.6.7 Grafiska förändringar utifrån användarens Locale

I18n-transformern hanterar översättning av text, datum och nummer i Cocoon. Men den klarar inte av att göra andra locale-beroende förändringar i användargränssnittet, som t.ex. att visa en bild på en engelsk flagga när sidan visas på engelska och en svensk flagga ifall sidan visas på svenska.

För detta ändamål finns en Cocoon pipeline action-komponent som kallas ”locale”. Denna komponent gör HTTP-requestets locale-information tillgänglig för Cocoons pipeline. Denna information kan sedan användas för att styra flödet genom pipelinen beroende på vilken locale klienten använder.

Man kan t.ex. lägga in en locale action i sin pipeline enligt följande:

```
<map:pipeline>
  <map:match pattern="*">
    <map:act type="locale">
      <map:generate type=”file” src="{1}.xml?locale={locale}"/>
      <map:transform type=”i18n”>
         <map:parameter name="catalogue-name" value="catalog"/>
         <map:parameter name="catalogue-location" value="translations"/>
      </map:transform>
      <map:transform type=”xslt” src=”document2html.xsl”/>
      <map:serialize type=”html”>
    </map:act>
  </map:match>
</map:pipeline>
```

I denna pipeline gör satsen <map:act type=”locale”> att HTTP-requestets locale information fångas in och görs tillgänglig för satserna inuti map:act-elementet. Generatorn kan automatiskt hämta information om vilken locale användarens webbläsare är inställd på genom Xpath-anropet {locale} och tilldela URI-parametern ”locale” detta värde så att användaren inte explicit behöver ange `...?locale=”se”` vid varje request.

Användaren kan också bli presenterad sidor med helt olika gränssnitt genom att man t.ex. använder olika XSL-stylesheet beroende på vilket språk som används:

```
      <map:transform type=”xslt” src=”document2html_{locale}.xsl”/>
```

Med denna metod kommer det att vara möjligt att byta ut t.ex. bilder i sidorna beroende på vilken locale klienten använder.

1.6.8 Flerspråksstöd: Slutsats

Exemplen i detta kapitel visar att det relativt enkelt går att lägga till stöd för flera språk i en Cocoon-applikation förutsatt att man använt i18n-transformern från början. I18n-transformern hjälper till att minimera antalet förändringar i affärslogik som behöver göras för att stödja ett nytt språk.