

Framework til webbaseret selvbetjening

"All good things come to he who waits"
Violet Fane (1843-1905)

Jesper Lindholt Ottosen
P. Langs Vej 6, Holme
8270 Højbjerg
jesper @ o2sn.dk

Forsideillustration af Majken Olesen Sander.
Specialet er skrevet og tegnet i Sun Microsystems StarOffice™ 6.0.
Java, J2EE, J2SE, Enterprise JavaBeans, JavaServer Pages, Java Servlet
og Java Server Faces er registrerede varemærker hos Sun Microsystems, Inc.

Abstract

The recent years have shown a large increase in the usage of web-based self-services like online banking, online government self-services and online telecommunication self-services. A web-based self-service is an application with a World Wide Web user interface provided by an organization, where the users may view and manipulate the properties of a long-term relationships with the organization.

Currently web-based self-services is developed using a number of different web technologies and general web application frameworks. Although current research discuss web application development in general, very little research is available on the construction of and common features of web-based self-service applications.

The aim of this thesis is to design an object-oriented framework for web-based self-service applications. Using software design patterns the framework provides hot spots for common properties of web-based self-service applications independently of the applied business domain and underlying web application framework technology.

The constructed self-service application framework has an application-oriented focus on personalization, framework layering and a MVC Model 2 architecture. It provides facilities for the constructing of a multi-level adaptable web-based interface and a state-based evaluation of the business rules within each self-service operation.

Indholdsfortegnelse

1 Indledning.....	9
1.1 Problemformulering.....	9
1.2 Proces og resultat.....	9
1.3 Opbygning.....	10
1.4 Tak til.....	10
2 Webbaseret selvbetjening.....	11
2.1 Værdien af selvbetjening.....	11
2.1.1 Digital offentlig forvaltning.....	11
2.1.2 Onlinebanking.....	12
2.1.3 Telekommunikation.....	13
2.2 Egenskaber ved selvbetjening.....	14
2.2.1 Passwordbeskyttet.....	14
2.2.2 Brugertilpasset.....	14
2.2.3 Samling af muligheder.....	15
2.3 Afgrænsning.....	15
2.3.1 Platformsspecifik konfiguration.....	15
2.3.2 Portaler.....	16
2.3.3 Webbaserede programmer	16
2.3.4 Webbutikker.....	16
2.3.5 CRM-systemer.....	17
2.4 Konklusion.....	17
3 Objekt-orienterede frameworks.....	19
3.1 Frameworks generelt.....	19
3.1.1 Objekt-orienterede frameworks.....	20
3.1.2 Framework-hot spots.....	20
3.1.3 Relation til Template Method.....	21
3.2 Webapplikation-frameworks.....	22
3.2.1 Java 2 Enterprise Edition.....	22
3.2.2 MVC Model 2-arkitekturen.....	24
3.2.3 Jakarta Apache Struts.....	25
3.3 Konklusion.....	27
4 Krav til selvbetjeningsframework.....	29
4.1 Motivation.....	29
4.1.1 Vurdering af eksisterende frameworks.....	29
4.1.2 Etablering af et nyt framework.....	31
4.2 Overordnet design.....	31

4.2.1	Applikationsorienteret.....	32
4.2.2	Flere abstraktionslag.....	32
4.2.3	Direkte og indirekte brugertilpasning.....	32
4.2.4	MVC Model 2-arkitektur.....	33
4.3	Komponenter.....	33
4.3.1	Brugertilpasset View.....	34
4.3.2	Tilstandsbaseret Controller.....	34
4.4	Konklusion.....	35
5	Et framework over frameworks.....	37
5.1	Analyse.....	37
5.1.1	Formalisering.....	37
5.1.2	Grundkonstruktion.....	39
5.1.3	Unification.....	40
5.1.4	Seperation.....	41
5.2	Design.....	42
5.2.1	Kombination.....	42
5.2.2	Parameterstyring.....	43
5.2.3	Kaldstyring.....	45
5.3	Implementation.....	46
5.3.1	WebLet-frameworket.....	46
5.3.2	Greetings-applikationen	48
5.4	Konklusion	49
5.4.1	Relateret forskning.....	50
5.4.2	Evaluering.....	50
5.4.3	Fremtidigt forskning.....	51
6	Brugertilpasset View-lag.....	53
6.1	Analyse.....	53
6.1.1	Formalisering.....	54
6.1.2	Dimensionerne.....	55
6.2	Grafiske komponenter.....	55
6.2.1	Beskrivelse.....	56
6.2.2	Indhold i det grafiske design.....	56
6.2.3	Grafisk design i indholdet.....	57
6.2.4	Afkobling af grafik og operationer	58
6.2.5	TagLib-mærkater	59
6.2.6	Implementation.....	60
6.2.7	Eksempel.....	62
6.2.8	Interaktion.....	63
6.2.9	Alternativer.....	64
6.3	Tekstkomponenter.....	65
6.3.1	Beskrivelse.....	66
6.3.2	TagLib-mærkater	66

6.3.3	Implementation.....	67
6.3.4	Eksempel.....	68
6.3.5	Interaktion.....	69
6.3.6	Alternativer.....	70
6.4	Menukomponenter.....	70
6.4.1	Samlet menulayout.....	70
6.4.2	TagLib-mærkater	71
6.4.3	Implementation.....	72
6.4.4	Eksempel.....	73
6.4.5	Interaktion.....	74
6.4.6	Alternativer.....	74
6.5	Teknisk evaluering.....	75
6.5.1	Performance.....	75
6.5.2	Vedligeholdelse.....	76
6.5.3	Anvendelse.....	76
6.6	Konklusion.....	77
6.6.1	Relaterede design patterns.....	77
6.6.2	Evaluering som framework.....	78
6.6.3	Fremtidigt forskning.....	79
7	Tilstandsbaseret Controller-lag.....	81
7.1	Analyse.....	81
7.1.1	Beskrivelse.....	82
7.1.2	Forespørgelserne.....	84
7.1.3	Formalisering.....	85
7.2	Design.....	86
7.2.1	Komponenter.....	87
7.2.2	Indkapsling.....	88
7.3	Implementation.....	89
7.3.1	Indpakning.....	89
7.3.2	Hot spots.....	90
7.3.3	Tilstandsmaskine.....	92
7.3.4	Tilstande.....	94
7.3.5	Parametrisering.....	97
7.4	Eksempel.....	99
7.4.1	J2EE-applikation.....	99
7.4.2	Struts-applikation.....	100
7.5	Konklusion.....	102
7.5.1	Relateret forskning.....	102
7.5.2	Evaluering som framework.....	102
7.5.3	Fremtidigt forskning.....	103
8	Konklusion.....	105
8.1	WS-frameworket.....	105

8.1.1	Komponenter.....	105
8.1.2	Vurdering af Struts og J2EE.....	106
8.2	Fremtidig forskning.....	106
9	Referencer.....	107
10	Indhold på vedlagt cd-rom.....	113
10.1	Oversigt.....	113
10.1.1	Dokumenter.....	113
10.1.2	Dokumentation.....	113
10.1.3	Kildekode.....	114
10.1.4	Systemer.....	114
10.2	Vejledninger.....	115
10.2.1	Greetings-applikationen.....	115
10.2.2	MDP-applikationen.....	115
10.2.3	SBC-applikationen.....	116

1 Indledning

Dette speciale indeholder analyse, design og implementation af et objekt-orienteret framework. Målet for frameworket er at det skal udgøre en løsning til nogle af de generelle problemstillinger, der opstår i forbindelse med udviklingen af webbaseret selvbetjening.

Offentlige organisationer og private virksomheder tilbyder med webbaserede selvbetjeninger både nye og forbedrede servicemuligheder til brugerne, samtidig med at der opnås væsentlige besparelser i form af effektiviserede og digitaliserede arbejdsgange for organisationen. Onlinebanking og digital offentlig forvaltning er eksempler på webbaserede selvbetjeninger, der er med til at styrke offentlige og kommercielle ydelser via World Wide Web.

1.1 Problemformulering

På trods af et stigende antal selvbetjeninger og mere hyppig brug af webbaseret selvbetjening er der, så vidt vides, ikke megen forskning, der relaterer sig direkte til design og implementation af webbaserede selvbetjeninger. Ligeledes findes der kun få forskningsmæssige resultater, der diskuterer brugen af frameworks til at løse de overordnede problemstillinger i forbindelse med webbaseret selvbetjening.

Delmålene i dette speciale er at:

1. Redegøre for begrebet webbaseret selvbetjening
2. Analysere egenskaber ved webbaseret selvbetjening
3. Designe et framework til webbaseret selvbetjening
4. Implementationen af et framework til webbaseret selvbetjening
5. Vurdering af alternativer til frameworkets design og implementation

1.2 Proces og resultat

Resultatet indeholdt i dette speciale er dels en redegørelse og analyse af begrebet webbaseret selvbetjening, dels design og implementation af et framework til applikationer indenfor begrebet. Frameworket illustreres ved tre webapplikationer, som beskriver:

- en arkitekturmæssig abstraktion over eksisterende frameworks
- en opdeling af brugerfladen i uafhængige dele
- en ensartet og trinvis afvikling af regler og forudsætninger

Der er benyttet en eksperimentel udviklingsproces, hvor hver delløsning først er blevet analyseret, designet og beskrevet. Derefter er hver delløsning blevet implementeret i en prototype, afprøvet og vurderet.

1.3 Opbygning

Specialet er opbygget af indledningen, seks indholdskapitler, en konklusion samt to tillæg. Af de seks indholdskapitler benyttes de første to til en redegørelse og analyse af karakteristika ved henholdsvis webbaseret selvbetjening (kapitel 2) og objekt-orienterede frameworks (kapitel 3). Kapitel 4 indeholder den overordnede designbeskrivelse af et framework til webbaseret selvbetjening. Kapitel 5 omhandler, hvorledes to eksisterende whitebox-hot spots kombineres til ét nyt framework. I de to efterfølgende kapitel analyseres, designes og implementeres to delkomponenter af frameworket. Kapitel 6 omhandler komponenter til brugerfladen og kapitel 7 komponenter til afvikling af forespørgelser. Kapitel 8 indeholder den afsluttende konklusion på specialet.

Det første tillæg (kapitel 9) består af en liste over anvendt litteratur, referencer til artikler om selvbetjening og omtalte teknologier. Det sidste tillæg (kapitel 10) beskriver indholdet på den vedlagte cd-rom, som indholder kildekoden til et framework til webbaseret selvbetjening, dokumentation af kildekoden og binære filer til de systemer, der er blevet brugt i forbindelse med implementationen. Design patterns og lignende begreber skrives med **fed skrift** første gang. Citater skrives med *kursiv*.

1.4 Tak til

Emnet selvbetjening er valgt med inspiration fra mit arbejde hos TDC (Internet), hvor jeg de seneste år har fulgt udviklingen af webbaseret selvbetjening og observeret en række udviklingsmæssige udfordringer. Blandt andet på grund af arbejdet har specialet været nogle år undervejs, så det er glædeligt at se det færdigt. Det har krævet både afsavn og en ekstra indsats for mig og mine nærmeste.

Den største og dybeste tak går til min kone Anne. Til min familie, kolleger og venner en varm tak for, at de har holdt mig ud trods sliddet og fraværet. Tak til Majken for en unik forsideillustration og til Anders K. Olsen og Anne for kommentarer og korrekturlæsning. Til min vejleder Henrik en tak for godt mod- og medspil og for hjælpen med at få en læseplads til specialearbejdet.

2 Webbaseret selvbetjening

En webbaseret selvbetjening er en brugerflade på World Wide Web, hvor brugeren har løbende mulighed for at manipulere egenskaberne ved en langvarig relation til en organisation (et firma, offentlig myndighed m.v.).

Blandt de mest populære og omtalte danske selvbetjening er Told- og Skattestyrelsens "Tast Selv", Nordea Netbank [Nordea] og Danske Bank Netbank [DB] og e-boks.dk. Indenfor telekommunikation har de fleste operatører alle forskellige muligheder for selvbetjening.

I denne sammenhæng er selvbetjening ikke de kortvarige relationer, som f.eks. i forbindelse med onlinehandel eller tilsvarende enkeltstående handlinger. Gode webbaserede muligheder kan være et middel til at opnå øget kundeloyalitet for kortvarige relationer. Har en udbyder af mobiltelefoni en selvbetjening til optankning af eksempelvis taletid, kan dette fastholde brugerne udover den initiale salgssituation.

Da selvbetjening spænder over både kommercielle og ikke-kommercielle sfærer, benævnes brugerne af webbaseret selvbetjening som brugere og ikke som kunder, borgere eller hvad de måtte være i de konkrete sammenhænge. Tillægsordet webbaseret vil i de følgende kapitler som regel blive udeladt og webbaseret selvbetjening blot omtalt som selvbetjening.

I de følgende tre afsnit gives en beskrivelse af begrebet selvbetjening. Først beskrives ud fra forskellige kilder værdien af selvbetjening, dernæst beskrives generelle egenskaber ved begrebet og til sidst foretages en afgrænsning til beslægtede webapplikationer.

2.1 Værdien af selvbetjening

Selvbetjening er et væsentligt samfundsøkonomisk aktiv med fordele og besparelser både for brugere og organisationer. Specielt indenfor telekommunikation, offentlig forvaltning og onlinebanking findes der dokumentation for værdien og brugen af selvbetjening.

2.1.1 Digital offentlig forvaltning

Offentlig selvbetjening er blandt andet (fra [Eberhard+ '02, side 21]):

- Udfylde selvangivelse og opdatere forskudsskema
- Bestille og forny lån af bøger på offentlige biblioteker
- Udfylde skemaer vedrørende flytning, lægeskift, daginstitutioner
- Modtage ejendomsinformation
- Ansøgning og beregning af pension

Dekan Tom Lautrup-Pedersen fra det Samfundsvidenskabelige Fakultet, Aarhus Universitet skriver, at: *"Den offentlige forvaltning og service til borgerne står over for en udfordring, når andelen af ældre i samfundet stiger samtidig med, at andelen af erhvervsaktive er uændret. Det betyder, at der bliver stillet krav til det offentlige om at skulle yde den samme service til en lavere pris. Her spiller digital forvaltning en central rolle"* [Pedersen '02].

Selvbetjening indenfor digital offentlig forvaltning er ikke kun et dansk fænomen. Analyseinstituttet Taylor Nelson Sofres udgav i starten af 2003 en årlig analyse af "government online usage" [Mellor+ '02] på baggrund af interview med ca. 31.000 personer i 31 lande verden over. Et af resultaterne, der dokumenteres, er, at 30% af de adspurgte havde benyttet digital offentlig forvaltning indenfor de sidste 12 måneder. 53% af de adspurgte danskere svarede bekræftende hvilket placerer Danmark på en delt 3. plads.

Hos e-blanket.dk omtales amternes besparelspotentiale ved brugen af digitaliseret blankethåndtering: *"Gennemsnitsbesparelsen for blanketter inden for det offentlige estimeres f.eks. til 424 kr. pr. indsendt blanket. Dermed sikres en samlet besparelse i millionklassen ved blot få tusinde indsendelser om året"* [e-Blanket].

PLS Rambøll Management dokumenterer i "Den Digitale Borger 2002" [Eberhard+ 02] brugen og kendskabet til offentlig selvbetjening. Der påpeges, at kendskabet til offentlig selvbetjening kun gradvist skrider frem, trods et stort udvalg af muligheder. Den største udfordring for de offentlige faciliteter er, at nyhedens interesse er faldet, så der i stedet skal fokuseres på, at selvbetjening vil give reelle tidsmæssige besparelser. Af de adspurgte svarer 69% bekræftende på, at det offentlige burde yde selvbetjening i øget omfang, og 64% bekræftende på, at de i højere grad vil benytte sig af det offentliges selvbetjening fremover.

2.1.2 Onlinebanking

Onlinebanking er et væsentligt forretningsgrundlag for alle banker, specielt for de banker, der ikke har fysiske filialer, men udelukkende baserer sig på selvbetjening. I onlinebanking har brugeren ofte følgende muligheder (set hos Nordea):

- Se banksaldo, posterings og betalinger
- Betale regninger og overføre penge
- Kommunikere med bankrådgiver
- Lægge budget, regne på lån og ansøge om kassekredit
- Bestille rejsevaluta og kreditkort

Finansrådet [Finansrådet] har på deres hjemmeside en statistik over antallet af aktive onlinebankingaftaler. Den viser en markant stigning fra 502.600 medio 1999 til 1.764.554 medio 2002, en stigning på ca. 350% over 3 år. Nordea, som er den største nordiske bank med over 9.7 millioner kundeforhold omtaler på deres hjemmeside, at deres onlinebanking benyttes af 3.3 millioner internetbrugere 10 millioner gange om måneden.

Medio marts 2003 var der et hardwarenedbrud hos Danske Bank, som gjorde blandt andet deres online bank utilgængelig i flere dage. Omkostningerne bliver ikke kun et omsætningstab, men også et stort prestige- og troværdighedstab. Selvbetjening forventes at være tilgængelig på alle tidspunkter af døgnet, og hvis det er midlertidigt utilgængeligt på grund af planlagt eller akut driftsvedligeholdelse, giver det tab af troværdighed og i yderste konsekvens tab af brugere.

2.1.3 Telekommunikation

Selvbetjening indenfor telekommunikation er eksempelvis (set hos [TDC]):

- Skift af password til internetopkobling
- Bestilling af IP-adresser og hastighedsændringer på ADSL-forbindelser
- Se oversigt over mobil-, internet- og fastnetopkald
- Foretage optankning af taletidskort
- Aktivere og benytte forbrugskontrol, jfr. [Telestyrelsen]

Indenfor telekommunikation forventes det, at indførelsen af mere selvbetjening til serviceinformationer, produktændringer og salg kan nedbringe antallet af opkald til kundeservice-callcentre. En Forrester-rapport [Zurek '01] beskriver begrebet selvbetjening, som *"der, hvor kunderne kan hjælpe sig selv"*. Samme rapport beskriver, at indførelsen af selvbetjening har et investeringsafkast (Return on Investment) på 200%.

Artikler hos både InformationWeek [Watson+ 01] og Yankee Group [Yankee '00] anfører, at indførelsen af selvbetjening ikke nødvendigvis reducerer omfanget af callcenter-opkald, men i stedet kan medføre en stigning i kompleksiteten af de enkelte opkald. Hos [Watson+ 01] pointeres det, at indførelsen af selvbetjening er den rette strategi, og at en stigning i antallet af henvendelser er et godt tegn på at brugerne er engagerede og vil i kontakt med organisationen. Rådet er at [Watson+ '01,]: *"give brugerne adgang til de samme informationer uanset kommunikationskanalen og en mulighed for at hjælpe sig selv hvor det er muligt. Det vil både give brugerne, hvad de forventer og hjælper med til skære ned på callcenter-omkostningerne"*.

2.2 Egenskaber ved selvbetjening

En selvbetjening er generelt kendetegnet ved at være personlig. Dels giver et personligt password adgang til selvbetjeningen og dels tilpasses selvbetjeningens opbygning og indhold til brugerens relation. Følgende egenskaber kendetegner en webbaseret selvbetjening:

- Passwordbeskyttet
- Brugertilpasset
- Samling af muligheder

På forsideillustrationen illustreres begrebet selvbetjening ved en (passwordbeskyttet) hængelås, som giver (personlige) adgange til forskellige verdner med en lang række forskellige delelementer (muligheder).

2.2.1 Passwordbeskyttet

Den enkelte bruger logger ind i en selvbetjening ved at angive et personligt brugernavn og password. Brugeren identificeres entydigt på baggrund af disse informationer, som kan være enten tildelt eller selvvalgt. Mens brugeren er logget ind i selvbetjeningen, fremgår det af brugerfladen, at brugeren er logget ind, og det fremgår, hvordan selvbetjeningen afsluttes.

På grund af de personlige eller økonomiske oplysninger, der ofte håndteres i en selvbetjening, indbygges der ofte forstærkede sikkerhedslag enten ved brug af HTTPS-kommunikation med Secure Socket Layer-kryptering (SSL), eller ved at lade kommunikationen foregå i en signeret Java Applet. Yderligere information om mulige sikkerhedsovervejelser i selvbetjening forefindes i forbindelse med den offentlige digitale signatur [VTU '03] eller hos Open Web Application Security Project [OWASP].

2.2.2 Brugertilpasset

Brugertilpasningen (personaliseringen) kan foregå på to måder, indirekte og direkte. Den direkte brugertilpasning består af de ting, som brugeren selv kan vælge. Dette kunne f.eks. være, at brugeren kan vælge, hvilke af de mulige menupunkter, der skal vises i selvbetjeningen, at brugeren kan navngive delelementer (kontonavne og kategoriseringer), eller at brugeren selv har mulighed for at vælge det sprog, brugerfladen skal vises i. Bortset fra valg af sprog afhænger det i høj grad af det konkrete forretningsområde, hvilke elementer der med fordel kan tilbydes en direkte brugertilpasning af.

Den indirekte brugertilpasning består af de delelementer, som organisationen benytter for at differentiere mellem forskellige typer af brugere. Til sammenligning er der en række fællestræk ved de former for indirekte brugertilpasning, der optræder i forskellige selvbetjeninge.

Findes der i organisationen forskellige typer af brugerne eller relationer, skal dette være afspejlet i selvbetjeningen. Hos Nordea er der forskel på menupunkterne i selvbetjeningen alt efter, om man har en Fordel- eller Basisrelation. Tilpasningen kan også have karakter af forskellige tekster og indhold alt efter den pågældende relationen, f.eks. at teksterne i selvbetjeningen omtaler enten en Fordels-konto eller en Basiskonto, og ikke blot en generel konto.

Brugerfladens grafiske design er også en del af den indirekte brugertilpasning, da den samme selvbetjening ofte skal præsenteres med forskellige udseender eller forskellige selvbetjeningsmuligheder i samme udseende. Dette kan enten være på grund af forskellige relationers grafiske stil (Basisrelationer har en bronzefarvet selvbetjening, andre en guldfarvet) eller på baggrund af det grafiske design på den hjemmeside, som brugeren tager udgangspunkt i (samme selvbetjening med forskellige brands).

2.2.3 Samling af muligheder

En selvbetjening består af en samlet adgang til en række muligheder, som brugeren kan vælge imellem, og som er afhængig af den pågældende brugers relationen til organisationen. Mulighederne i selvbetjeningen kan med fordel samles i en overskuelig menu, som viser netop de muligheder, som brugeren kan udføre.

Der kan være flere muligheder tilknyttet en enkelt relation, og de enkelte muligheder kan bestå af flere forskellige grundtrin. I forbindelse med en bankkonto er der ofte både mulighed for at se en posteringsoversigt og overføre penge. Posteringsoversigten består typisk af et enkelt grundtrin, mens en bankoverførelse f.eks. består af følgende grundtrin: vælg konto og beløb, vælg modtager og overførelsesdato. Hvert grundtrin tager udgangspunkt i de aktuelle indtastningsfelter, valideringsrutiner, efterfølgende og forrige grundtrin.

2.3 Afgrænsning

En selvbetjening defineres i dette speciale med udgangspunkt i ovenstående begreber. Der findes en række forskellige webbaserede brugerflader, der har nogle af de samme karakteristika som selvbetjeningen. Disse forskellige webbaserede systemer diskuteres og analyseres kort, for at afgrænse hvad der ikke umiddelbart betragtes som en del af begrebet webbaseret selvbetjening.

2.3.1 Platformsspecifik konfiguration

En platformsspecifik konfiguration er de webapplikationer, hvor brugeren benytter faciliteter direkte på den pågældende database eller platform. De forskellige faciliteter er placeret på den enkelte platform og der er ingen

samling på tværs af platformene indenfor den samme organisation. Ville man i 1996 bestille en hjemmeside hos TDC, foregik det direkte på en hjemmesideserver, mens bestilling af emailadresser foregik på en mailserv. Til sammenligning foregår disse ting i 2003 fra en samlet selvbetjening.

Platformsspecifikke konfigurationer er ofte første skridt til en samlet selvbetjening, hvor man kun logger ind ét sted for at få adgang til alle mulighederne. Faciliteter der stilles direkte til rådighed på platforme er ofte måden at introducere de første selvbetjeningsmuligheder.

2.3.2 Portaler

De første portaler og communities opstod omkring 1998, hvor specielt Netscape førte an med at tilbyde brugerne adgang til vejrudsigter og nyhedstjenester. Der findes fortsat mange portaler og communities på World Wide Web, hvor mennesker samles omkring chat, debat og profilinformation.

En portal ofte er drevet af en indholdsleverandør eller sponsor (Microsoft MSN, Netscape AOL m.v.) indeholder ofte flere forskellige indholdsområder eller communities. Sammenlignet med en selvbetjening er der i forbindelse med portaler og communities ikke fokus på de "forretningsmæssige" relationer mellem organisationen og brugeren, men primært fokus på brugeren selv og dennes relation til andre brugere af portalen.

2.3.3 Webbaserede programmer

Et webbaseret program er et eksisterende program, som har fået en webbaseret brugerflade. Webbaserede programmer er et supplement til eksisterende programmer på en arbejdsstation, eksempelvis webmail, webbaserede news og kalendere, webbaseret tekstbehandling m.v.

Det mest kendte webbaserede program er nok MSN hotmail, der startede som en selvstændig facilitet i 1996, og siden blev opkøbt af Microsoft og indbygget i MSN-portal. Det ses oftere og oftere, at portaler indbygger flere og flere webbaserede programmer i deres sortiment af faciliteter.

2.3.4 Webbutikker

I en webbutik kan brugeren købe produkter og ydelser som f.eks. software, bøger, billetter og medlemskaber. Set i forhold til almindelig "fysisk" butikshandel, foregår handlen i en webbutik udelukkende ved, at brugeren betjener sig selv. Systemmæssigt er egenskaberne ved en webbutik blevet veletablerede, og flere udviklingshuse har lavet faciliteter til hosting og manipulation af ensartede og standardiserede webbutikker.

I sammenligning med begrebet selvbetjening omtalt ovenfor er det oftest kortvarige relationer, der etableres i en webbutik. Egenskaberne for relationen mellem brugeren og organisationen er begrænset til betalings- og adresseinformationer, som eventuelt kan gemmes til næste købsituation. I en webbaseret selvbetjening er købet ikke i fokus, men nærmere vedligeholdelsen af dynamiske egenskaber ved en længerevarende relation.

2.3.5 CRM-systemer

Et Customer Relationship Management-system (CRM-system) er et større forretningssystem, som håndterer kunderelationer og kundekommunikation. Siebel, Kana og BroadVision er blandt de største (amerikanske) leverandører af CRM-systemer.

Webbaseret selvbetjening er en enkelt af kommunikationskanalerne indeholdt i et CRM-system, andre kanaler er f.eks. email- og telefonkommunikation. I et CRM-system er der fokus på kommunikationen mellem brugeren og organisationen og ikke nødvendigvis på, at brugeren kan vedligeholde egenskaberne ved relationen med organisationen.

2.4 Konklusion

Der er ovenfor gjort rede for egenskaber og værdi ved selvbetjening indenfor digital offentlig forvaltning, onlinebanking og telekommunikation. Den samlede vurdering er, at selvbetjening er et samfundsøkonomisk aktiv for både brugere, kommercielle og offentlige organisationer.

Begrebet webbaseret selvbetjening defineres som værende en World Wide Web-brugerflade, hvor brugeren har en passwordbeskyttet brugertilpasset samling af muligheder for at manipulere egenskaberne ved en langvarig relation til en organisation.

3 Objekt-orienterede frameworks

For at udvikle og designe en selvbetjening benyttes umiddelbart de samme udviklingsteknikker og udviklingsteknologier som ved udviklingen af generelle World Wide Web-applikationer. Ofte udvikles webapplikationer med baggrund i et webapplikation-framework. Et webapplikation-framework er et objekt-orienteret framework, der stiller en udvidet funktionalitet og en abstraktion af [CGI]-kommunikationen til rådighed i en rammeløsning for webapplikationer. Eksempler på sådanne webapplikation-frameworks er blandt andet Java 2 Enterprise Edition [J2EE] og Apache Jakarta Struts [Struts].

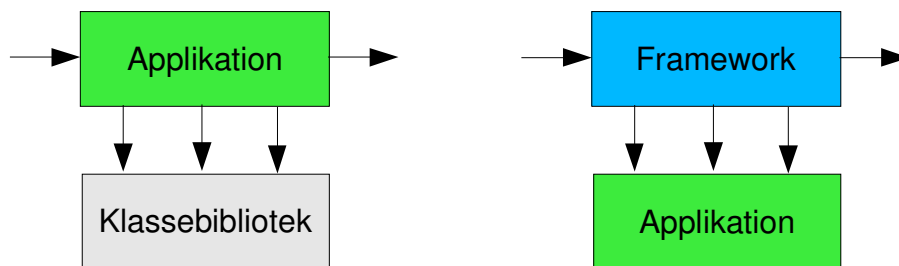
Et objekt-orienteret framework er en fællesbetegnelse for rammeløsninger med et objekt-orienteret design og en objekt-orienteret implementation til udvikling af selvstændige applikationer indenfor et afgrænset område. Et framework kendetegnes ved at pålægge applikationen en bestemt struktur, da det i forbindelse med brugen af frameworks er fra frameworket, at rammerne bliver sat for, hvordan applikationen afvikles.

I de nedenstående kapitler beskrives først egenskaber ved frameworks generelt og dernæst karakteristika ved objekt-orienterede frameworks. Derefter beskrives dels to udvalgte webapplikation-frameworks og dels den ofte benyttede MVC Model 2-arkitektur.

3.1 Frameworks generelt

Begrebet "framework" kan defineres som værende en generel softwareløsning, der kan bruges til at lave konkrete softwareapplikationer indenfor et bestemt domæne. Et framework er en ramme til generelle løsninger, og ikke en selvstændigt softwareapplikation i sig selv.

Et traditionelt softwareklassebibliotek kan ligeledes beskrives som bestående af generelle softwareløsninger, men er konceptuelt forskelligt fra et framework. I et klassebibliotek stilles forskellige faciliteter til rådighed, som kan benyttes af konkrete applikationer uafhængigt af, hvordan applikationerne i øvrigt er konstrueret. I modsætning til dette fastlægger den abstrakte struktur i et framework altid, hvorledes den konkrete applikation, der bruger frameworket, skal udføres.



Figur 1: Klassebiblioteker vs. frameworks [Fontura+ '02, side 6]

Den konkrete applikation, der tager udgangspunkt i et framework, indeholder nogle bestemte dele, som frameworket gør brug af. Dette er i modsætning til brugen af klassebiblioteker, hvor udførelsen er styret i det konkrete applikation, der kalder klassebiblioteket.

3.1.1 Objekt-orienterede frameworks

Et frameworks relation til en konkret applikation kan sammenlignes med det objekt-orienterede klasse-begreb. Klassebegrebet er en abstrakt beskrivelse, der altid skal konkretiseres (eller instansieres) i et konkret objekt for at være i brug. Tilsvarende skal et framework oftest "instansieres" i sammenhæng med en konkret applikation for at komme i anvendelse. Ved hjælp af et objekt-orienteret framework opnås genbrug af både frameworkets kode og frameworkets design.

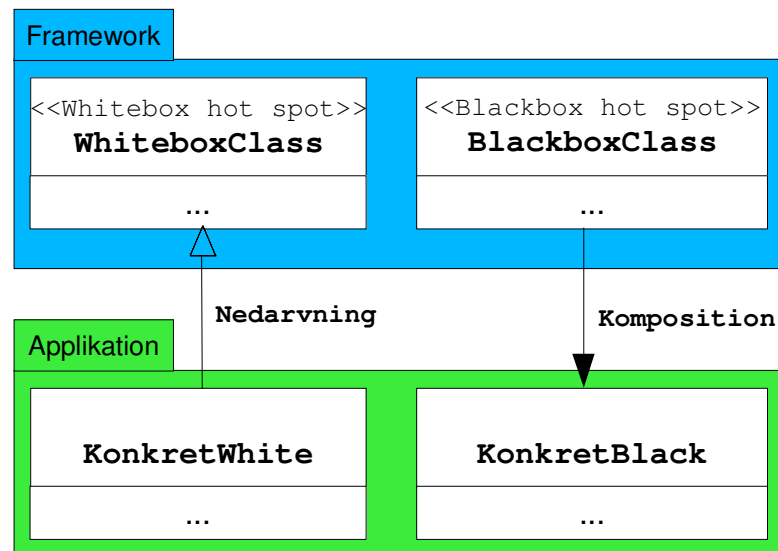
Begrebet framework er anvendt i både objekt-orienteret forskning og programmering. Inden for objekt-orienteret programmering er der stor anvendelse af frameworks på alle applikationsniveauer, specielt består mange frivillige open source-projekter af udviklingen af et framework inden for et bestemt domæne eller indenfor en bestemt type applikationer.

Af den objekt-orienterede forskning kan specielt nævnes værker af: Wolfgang Pree, Jan Bosch, Michael Mattsson og Ralph E. Johnson. Se referencelisten i kapitel 9 for konkrete referencer til deres relevante værker.

3.1.2 Framework-hot spots

Et objekt-orienteret framework definerer sine anvendelsesmuligheder i form af "hot spots" i modsætning til den fastlagte funktionalitet i frameworkets "frozen spots" [Markiewicz+ '01]. Hot spots i et objekt-orienteret framework karakteriseres ved enten at være whitebox (evt. glassbox) eller blackbox. Et blackbox-hot spot benyttes af den konkrete applikation via objekt-orienteret komposition eller delegering, mens et whitebox-hot spot benyttes via objekt-orienteret nedarvning.

Framework-hot spots betegnes enkelte steder som "hooks" eller "variationspunkter", da det er med udgangspunkt i de definerede hot spots, at frameworkets abstrakte løsning tilknyttes og varieres i forbindelse med de konkrete applikationer.



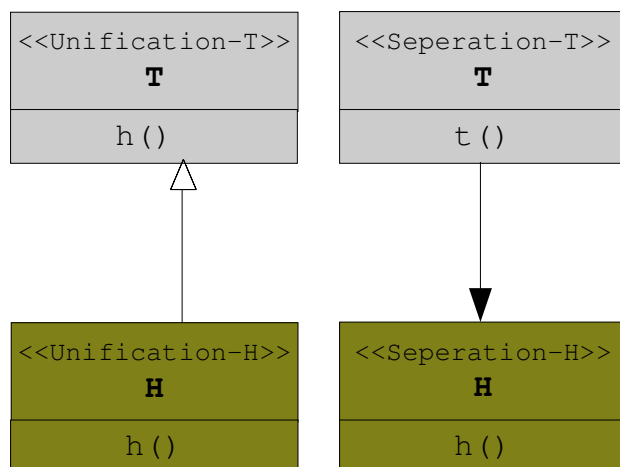
Figur 2: whitebox-hot spots vs. blackbox-hot spots

Et framework består ofte af både whitebox- og blackbox-hot spots. Er der overvejende flest whitebox-hot spots i et framework betegnes hele frameworket som et whitebox-framework og tilsvarende et blackbox-framework, hvis der i frameworket er overvejende flest blackbox-hot spots.

3.1.3 Relation til Template Method

I [Pree '99] dokumenteres de to hot spot-typer som variationer over design patternet **Template Method** [Gamma+ '95]. Template Method-design patternet beskriver, hvordan en algoritme kan beskrives i dels det overordnede skelet for algoritmen og dels de grundlæggende metoder i algoritmen. De grundlæggende metoder betegnes **hook**-metoder, mens metoden indeholdende algoritmens overordnede skelet betegnes **template**-metoden.

Template Method-design patternet beskrives i [Pree '94] ved enten at være implementeret ved **Seperation** eller ved **Unification**. Ved Unification er template-metoden, $t()$, placeret i en superklasse, T , indeholdende kald til den konkrete hook-metode i subclassesen H . Ved Seperation benyttes objekt-orienteret komposition til at tilknytte klasse H , indeholdende hook-metoden $h()$, til klassen T hvor template-metoden $t()$ er placeret.



Figur 3: Unification vs. Separation [Fontura+ '02, side 77 og 83]

Konklusionen i [Pree '99] er, at whitebox-hot spots svarer til Unification-variationen af Template Method-design patternet, og at blackbox-hot spots svarer til Separation-variationen af Template Method-design patternet.

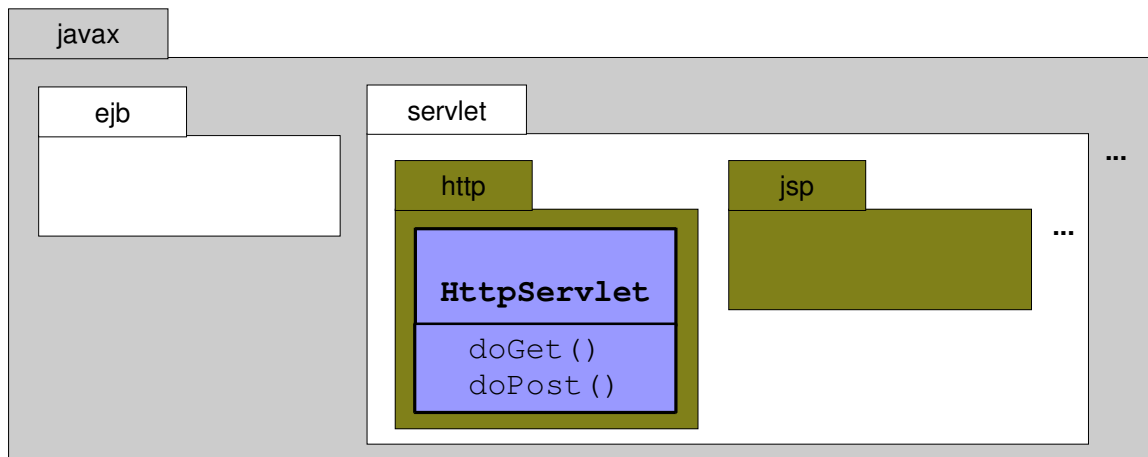
3.2 Webapplikation-frameworks

Antallet af objekt-orienterede frameworks til konstruktionen af webapplikationer er steget markant de seneste få år, så der nu findes en lang række forskellige kommercielle og open source-frameworks indenfor dette område. Hvert eneste af disse frameworks har hvert sit primære fokus, hver sine virkemidler og tilbyder hver især sine hot spots til webapplikationerne. Hos [Barracuda] findes en gennemgang af forskellige generelle webapplikation-frameworks, kommercielle såvel som open source. I dette speciale bliver der kigget nærmere på Struts og J2EE.

3.2.1 Java 2 Enterprise Edition

Et af de mest populære frameworks til webbaserede applikationer er webelementerne indeholdt i "Java 2 Platform, Enterprise Edition", som er en samling Java-specifikationer og -klasser målrettet mod implementation af forretningssystemer, deriblandt webapplikationer.

I forbindelse med udviklingen af webapplikationer er det specielt Enterprise JavaBeans, Java Servlet [Servlet] og JavaServer Pages [JSP], der tages udgangspunkt i. Selvom J2EE indeholder langt flere end blot disse teknologier, vil betegnelsen J2EE i denne sammenhæng blive brugt om de indeholdte "web tier technologies" [Singh+ '02, kapitel 4.2]

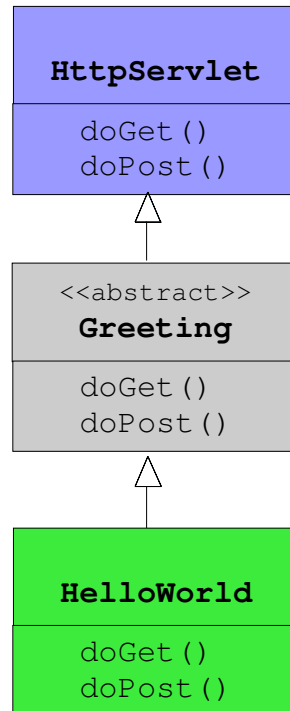


Figur 4: Java 2 Enterprise Edition

Webteknologierne i J2EE afvikles af en "servlet-container", der er en applikationsserver, der kan afvikle applikationer ifølge J2EE-specifikationerne. Servlet-container [Tomcat] fra Apache Software Foundation er reference implementationen for Servlet-og JSP- elementerne.

JavaServer Pages er en form for scriptsprog, hvor man i JSP-filer kan skrive applikationskode og HTML til brugerfladen [Bergsten '01]. Når JSP-filerne kaldes, oversætter servlet-containeren dem til Java-objekter og afvikler dem. Modsat tager man i forbindelse med Servlets direkte udgangspunkt i `HttpServlet`-klassen, som afvikles direkte hos servlet-containeren – primært indeholdende applikationskode, men eventuelt også indeholdende HTML.

`HttpServlet`-klassen bruges som et whitebox-hot spot [Hunter '01]. I den konkrete klasse, der nedarves til, specificeres en overskrivning af metoderne `doGet()` og `doPost()` kendt fra CGI-kommunikation. Alternativt til at nedarve direkte fra `HttpServlet`-klassen, opbygges der ofte et klassehierarki med `HttpServlet`-klassen i top. `HttpServlet`-klassen nedarves først i en generel subklasse og efterfølgende i en række konkrete subklasser og eventuelle yderligere specialiseringer.



Figur 5: *HttpServlet-baseret eksempel*

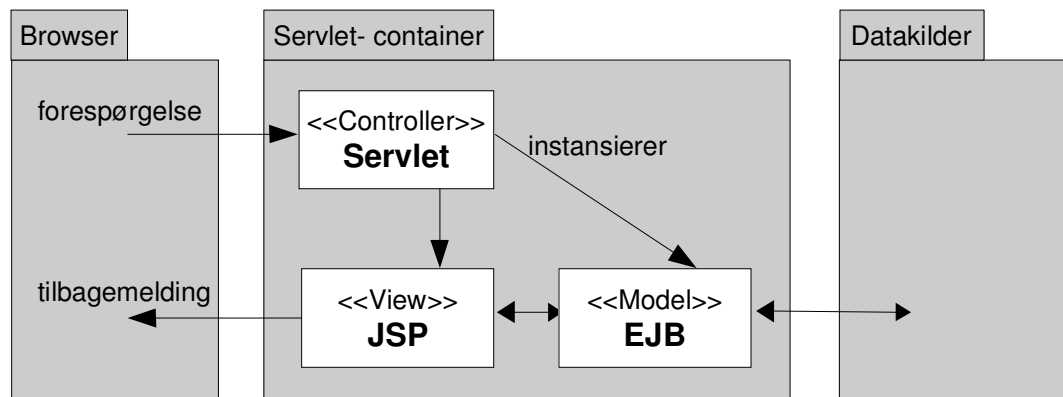
3.2.2 MVC Model 2-arkitekturen

En problemstilling i forbindelse med J2EE er, at det er muligt både at skrive applikationskode i JSP-filerne og HTML-output i Servlet-klasserne. Det giver en arkitekturmæssig udfordring, når de forskellige dele i applikationsstrukturen på denne måde kan blandes ustruktureret.

Som løsning på denne problemstilling er der i open source-miljøet opstået en arkitektur kaldet "**MVC Model 2-arkitekturen**" inspireret af det klassiske **Model View Controller** (MVC) arkitektur-design pattern [Buschmann+ '96, side 125-143]. Begrebet kan ikke entydigt tilegnes en enkelt kilde, et af de første steder det omtales, er i [Seshadri '99]. Siden er det omtalt i [Knight+ '02] [Brown '02] [Malani '02] og implementeret som arkitektur hos blandt andet [Turbine] [Barracuda] og [Struts].

Inspireret af MVC opdeles webapplikationen i tre lag:

- Modellen baseret på Enterprise JavaBeans
- View baseret på JavaServer Pages
- Controlleren baseret på Java Servlets



Figur 6: MVC Model 2, [Seshadri '99]

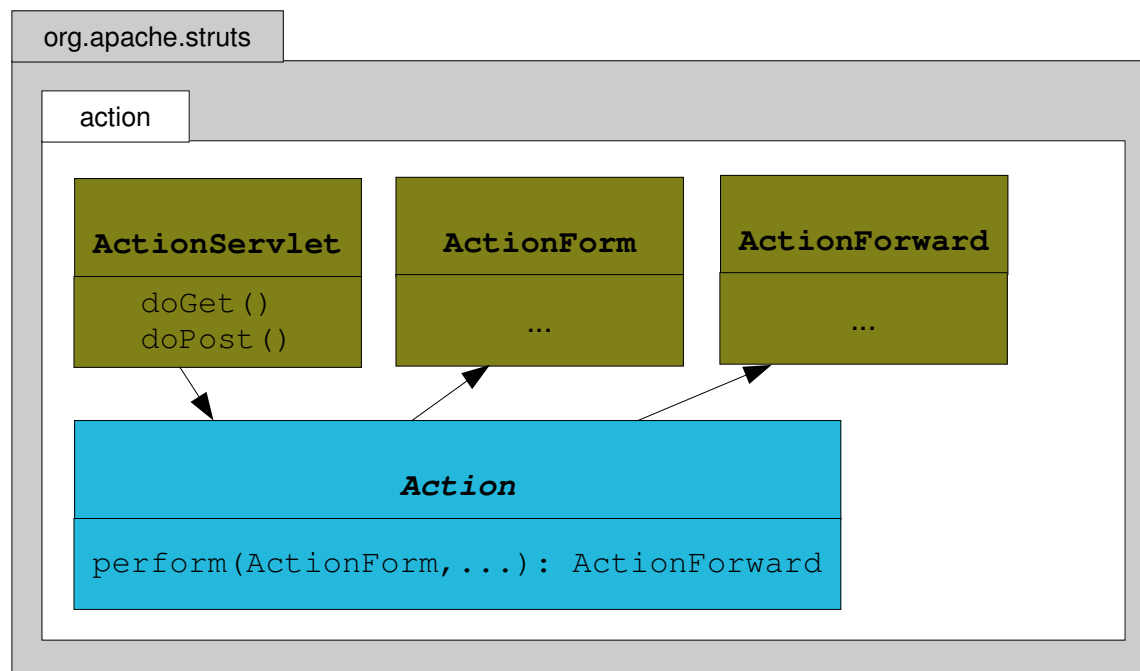
MVC Model 1 [Seshadri '99] [Malani '02] [Brown '02], benyttes som betegnelse, når applikationskontrollen er implementeret direkte i JSP-filer eller hvis Servlet-teknologien bruges til konstruktion af både brugerflade og applikationskontrol.

3.2.3 Jakarta Apache Struts

Open source-frameworket "Jakarta Apache Struts" er et af de steder, hvor MVC Model 2-arkitekturen er implementeret. Formålet med Struts er at stille et framework til rådighed, hvorfra man kan udvikle webapplikationer baseret på MVC Model 2-arkitekturen samt JSP- og Servlet-teknologierne. Det blev oprindeligt startet af en enkelt person i 2000, men er siden blevet et succesfuldt open source-projekt hos Apache Software Foundation, hvor version 1.0 kom i juli 2001 og version 1.1 er fortsat under udarbejdelse.

Via Struts har udvikleren en lang række faciliteter til rådighed til at udvikle webapplikationer med. Udvikleren har blandt andet mulighed for tage udgangspunkt i et whitebox-hot spot, klassen `Action`, som er tiltænkt webapplikationer af mange forskellige slags.

Hos [Larman+ '02] benyttes Rational Unified Process-dokument "Software Architecture and Design" til at give en god og dækkende beskrivelse af motivationen og arkitekturen bag Struts.



Figur 7: Struts arkitektur, [Larman+ '02, Figure 2]

I mange webapplikationer har man behov for at have brugerfladen (View-laget) på forskellige sprog. Dette problem håndterer Struts ved, at man kan placere tekster og andre sprogafhængige dele (som fx. billedreferencer) uafhængigt af JSP-siderne. Tekster og andre sproglige elementer placeres i en række `property`-filer, én for hvert sprog. Med den rigtige navngivning af disse `property`-filer kan Struts ud fra browserens sprogindstilling, vise tekster, billeder, links med videre i den resulterende JSP-side afhængigt af det sprog, som browseren er indstillet til.

I Struts placeres ansvaret for delegeringen og styringen af webapplikationen i en instans af `ActionServlet`-klassen. Den fungerer som Controller og håndterer de indkomne forespørgelser og delegerer ansvaret videre på baggrund af konfigurationen gemt i den tilhørende konfigurationsbeskrivelse. En instans af `ActionServlet` er således udelukkende en delegeringsklasse – en Controller i sin kerneform og uafhængig af den konkrete forespørgelse samt validering af data.

Model-laget i en Struts webapplikation er en instans af `ActionForm`-klassen, som basalt set er en indpakning af data, som findes enten i en database eller er input fra brugerfladens formularer. `ActionForm` er ansvarlig for at stille valideringsrutiner og samt tilgangsmetoder til rådighed på alle de informationer, man ønsker at benytte i afviklingen af forespørgelsen.

Selve afviklingen af forespørgelsen udvikles som en instans af klassen `Action`, som afslutningsvist delegerer ansvaret videre til en JSP-side, som på baggrund af data i det opdaterede Model-lag kan konstruere den resulterende tilbagemelding i HTML.

3.3 Konklusion

Frameworket Struts er et eksempel på et objekt-orienteret framework til udvikling af generelle webapplikationer. Ved at benytte Struts får de konkrete webapplikationer mulighed for at benytte udvidet funktionalitet og en abstraktion af de bagvedliggende teknologier- og teknikker. Struts er baseret på JSP og Servlet-elementerne i J2EE, som ofte også selv bruges som framework for webbaserede applikationer.

Både J2EE og Struts tilbyder deres funktionalitet og abstraktioner i form af forskellige framework-hot spots, som webapplikationen kan benytte. Uanset hvilken variation over Template Method-design patternet [Pree '99], framework-hot spot'et består af, påtvinger frameworket en bestemt struktur på applikationen. Som f.eks. når der i forbindelse med Struts benyttes MVC Model 2-arkitekturen til opdeling af applikationen i logiske dele.

4 Krav til selvbetjeningsframework

Ud fra analysen af egenskaberne ved selvbetjening, redegørelsen for egenskaber ved objekt-orienterede frameworks og beskrivelsen af udvalgte webapplikation-framework kan kravene til det overordnede design og komponenterne i et framework til webbaseret selvbetjening beskrives.

Designet af frameworket til webbaseret selvbetjening (fremover benævnt WS-frameworket) tager udgangspunkt i udfordringerne ved eksisterende webapplikation-frameworks, funktionaliteten i eksisterende selvbetjening. WS-frameworket skal være applikationsnært og målrettet udviklingen af selvbetjening, da det dermed vil passe bedre til selvbetjeningsapplikationer, end generelle teknologi- eller webapplikation-frameworks.

De nedenstående kapitler beskriver motivationen bag WS-frameworkets konstruktion og overordnede design. Designet af WS-frameworkets overordnede komponenter tager udgangspunkt i fire egenskaber, som er egenskaber ved objekt-orienterede frameworks og webbaserede selvbetjening.

4.1 Motivation

Opbygning, ændring og udbygning af webbaseret selvbetjening er i høj grad afhængig af organisationens forretning. Samtidigt er flere organisationers forretning, f.eks. det offentlige serviceniveau, i høj grad afhængig af brugen af selvbetjening. I de kommende år vil både brugere og organisationer forvente at få stillet yderligere webbaserede løsninger til rådighed og dermed yderligere besparelser og effektiviseringer ved digitalisering og automatisering.

Den store efterspørgelse og brug af selvbetjening gør det relevant at undersøge området og diskutere konstruktionen af et framework, som er målrettet udviklingen af selvbetjening. Dette framework skal ikke, som de fleste eksisterende webapplikation-framework, blot fokusere på generelle faciliteter til webapplikationer, men tage udgangspunkt i selvbetjening som applikationsområde.

4.1.1 Vurdering af eksisterende frameworks

Selvbetjening baseret direkte på generelle webapplikation-framework har problemer ved hyppige ændringer og udvidelser. Uanset om J2EE eller Struts bruges som webapplikation-framework for en selvbetjening, vil udviklingen af forespørgelserne umiddelbart blive modelleret i et klassehierarki ud fra de tilbudte whitebox-hot spots.

Klassehierarkiet ændres løbende på grund af ændringer i organisationsprodukter. Nye selvbetjeningsoperationer indplaceres ofte uovervejet i den eksisterende hierarkiske struktur, og som udviklingen og vedligeholdelsen af systemet skrider frem, knopskyder klassehierarkiet både i dybden og i bredden, men altid med J2EE eller Struts whitebox-hot spot som udgangspunkt for hierarkiet. Uden en klar udviklingsstrategi for udbygningen af selvbetjeningen er der høj risiko for at situationen i følgende anti-patterns opstår: **Lava Flow, Spaghetti Code, Cut & Paste, Swiss Army Knife** [Brown+ '98].

Udviklingen af enkeltstående selvbetjeninger fokuserer ofte stift på deadlines og opfyldt funktionalitet i forhold til en kravspecifikation, og når udviklingen af en enkeltstående selvbetjening kommer under tidspres fravælges ofte de faciliteter, der skal bruges til at sikre løsningens vedligeholdelse og udvikling samt sikre løsningens fremtidige brug som framework og udgangspunkt for tilsvarende løsninger.

I forbindelse med selvbetjening er den største ulempe ved en J2EE-baseret applikation, at der ikke umiddelbart er faciliteter til understøttelse af selvbetjening. Udviklerne skal selv sørge for at udvikle og vedligeholde funktioner til eksempelvis, at brugeren er logget ind, og håndtere, at forretningsregler hyppigt kan ændres. Udviklerne skal opfinde faciliteter til selvbetjening igen og igen, uanset om andre udviklere har håndteret tilsvarende problematikker i andre webapplikationer.

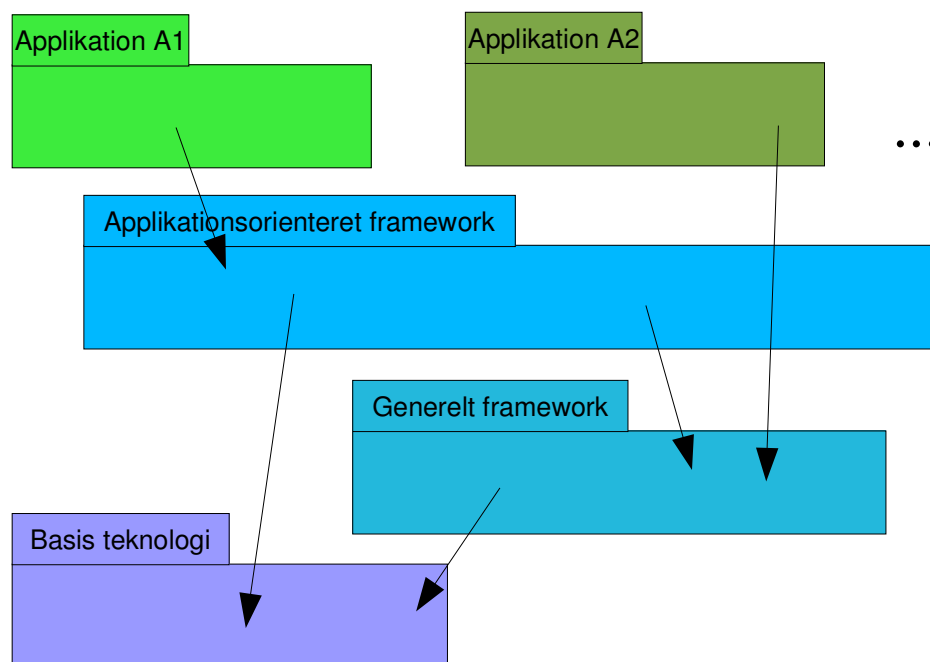
Ved brug af Struts til udvikling af selvbetjening kan udvikleren benytte faciliteter, som er mere målrettede webapplikationer end J2EE. Udgangspunktet for applikationer baseret på Struts er det whitebox-hot spot, som tilbydes i form af `execute()`-metoden i `Action`-klassen. Selvom Struts giver udvikleren nogle gode faciliteter i forbindelse med webapplikationer generelt, er der ingen yderligere beskrivelse eller understøttelse af de konkrete trin, der skal udføres i `execute()`-metoden. Det eneste udvikleren kan tage udgangspunkt i er følgende punkter fra [Struts Users Guide, kapitel 4.4]: *"en typisk Action-klasse implementerer ofte logik som følgende i sin execute()-metode:*

- *Validering af brugerens aktuelle tilstand*
- *Validering af input form egenskaberne*
- *Udførelse af den nødvendige ændring af data*
- *Opdatering af de serverbaserede elementer*
- *Returnering af en reference til en JSP-side"*

4.1.2 Etablering af et nyt framework

Det stigende antal webapplikation-frameworks gør det uoverskueligt for applikationsudvikleren at afgøre, hvilket framework der passer bedst til den konkrete anvendelse. Det framework, der umiddelbart passer bedst, er det framework, der er målrettet mod den pågældende anvendelse - altså et applikationsorienteret framework i højere grad end et generelt framework.

Applikationsorienterede frameworks, der er fokuseret mod et konkret anvendelsesområde, men samtidigt understøtter flere grundteknologier, har bedre potentiale som framework i den konkrete sammenhæng end frameworks med et mere generelt eller teknologiorienteret fokus.



Figur 8: Sammenhæng mellem forskellige frameworks

4.2 Overordnet design

WS-frameworkets overordnede design tager udgangspunkt i følgende begreber, som enten egenskaber ved objekt-orienterede frameworks eller er egenskaber for webbaserede selvbetjeninger. WS-frameworket skal:

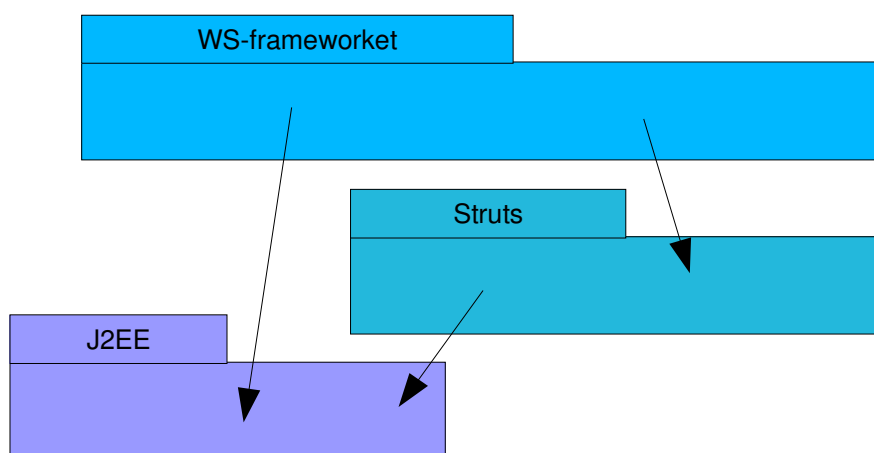
- være applikationsorienteret
- have flere abstraktionslag
- tilbyde direkte og indirekte brugertilpasning
- være opbygget efter MVC Model 2-arkitekturen

4.2.1 Applikationsorienteret

Som tidligere nævnt findes der en lang række forskellige webapplikations frameworks, men ikke umiddelbart et framework som fokuserer netop på faciliteter til selvbetjening. At WS-frameworket skal være applikationsorienteret betyder således, at det skal tage udgangspunkt i de fælles træk og de fælles karakteristika, som der er i selvbetjening, uanset deres konkrete forretningsområde. WS-frameworket skal tage udgangspunkt i begrebet selvbetjening som en passwordbeskyttet, brugertilpasset samling af muligheder.

4.2.2 Flere abstraktionslag

I [Singh+ '02, kapitel 4.3] og [Cann+ '02] beskrives, at intentionen med J2EE er, at det skal benyttes som ét framework ud af flere for at opbygge en applikation. Applikationen skal understøttes direkte af et applikationsnært framework, som derefter håndterer abstraktionen af elementerne i J2EE. Struts er et eksempel på et applikationsnært framework, men det er umiddelbart for generelt til selvbetjening. WS-frameworket skal netop være mere applikationsnært end Struts, og skal derfor bygge ovenpå på de to basisframeworks, J2EE og Struts.



Figur 9: Lagdeling i WS-frameworket

4.2.3 Direkte og indirekte brugertilpasning

Den indirekte brugertilpasning, der ofte findes i selvbetjening er, at selvbetjeningsbrugerfladen har et udseende, der er afhængig af brugerens relation, at teksterne er afhængige af brugerens relation og at brugeren kun får de menupunkter stillet til rådighed som er relevante.

Der stor forskel på hvad der benyttes som direkte brugertilpassede elementer i brugerfladen. Det afhænger i høj grad af den konkrete selvbetjening, hvad der giver mening at brugertilpasse. En ofte set brugertilpasset facilitet, er muligheden for brugeren selv at vælge hvilket sprog (blandt flere) som selvbetjening skal vises på.

4.2.4 MVC Model 2-arkitektur

Der er veldokumenterede erfaringer med at opbygge webapplikationer og webapplikation-frameworks på MVC Model 2-arkitekturen. Struts er blot et eksempel på, hvordan MVC Model 2 er implementeret ved hjælp af JSP- og Servlet-teknologierne fra J2EE. I [Singh+ '02, kapitel 4.4] anbefales det, at applikationer baseret på J2EE-teknologierne implementeres på baggrund af MVC Model 2-arkitekturen. Derfor er MVC Model 2-arkitekturen også en del af det overordnede design af WS-frameworket.

4.3 Komponenter

En af udfordringerne ved WS-frameworket er, at det ikke er tidsmæssigt muligt, at udvikle tre selvbetjeningsapplikationer på baggrund af frameworket, som ellers er den rettesnor, der gives i blandt andet [Roberts+ '98] og [Rüping '00], for, hvornår et framework dels har tjent sig ind og dels har "skudt" sig ind på anvendelsesområdet.

Frameworklitteraturen indeholder diskussioner af, hvordan frameworks og applikationer med udgangspunkt i framework udvikles. I [Rüping '00] fremgår det blandt andet, at hvis et framework skal udvikles samtidigt med applikationerne skal det i hovedtræk bestå af **Framelet**-komponenter [Pree+ '99] og være baseret på **Hollywood Princippet** [Bosch+ '99] for en velafgrænset del af applikationen.

Hollywood Princippet er en betegnelse for det princip, at det er frameworket, der styrer strukturen i applikationen og ikke applikationen selv. Framelet-begrebet, som det beskrives i [Pree+ '99], består af en modulariserbar løsning på maksimum ti klasser med et klart defineret simpelt interface og som ikke overtager programkontrollen.

På baggrund af disse overvejelser og på baggrund af den ønskede MVC Model 2-arkitektur, etableres WS-frameworket af afgrænsede frameworkløsninger til henholdsvis View- og Controller-lagene, da det er i disse lag at de generelle problemstillinger omkring et applikationsnært, brugertilpasset og lagdelt framework til selvbetjening umiddelbart håndteres.

WS-frameworket indeholder ikke komponenter i Model-laget, da dette i høj grad afhænger af det konkrete forretningsområde og den konkrete implementation. Model-laget kan i konkrete sammenhænge være baseret på JavaBeans, SQL-databaseforbindelser eller SOAP Web Services. Det eneste, WS-frameworket vil benytte sig af, er at Controller-laget har adgang til Model-laget ved et givet interface.

4.3.1 Brugertilpasset View

View-laget i en selvbetjening udgøres udover selve operationen ("skift password", "se forbrug" etc) af flere forskellige brugerfladeelementer; grafik, menupunkter, tekster m.v. De grafiske elementer følger den samme grafiske og sproglige stil som den pågældende relations og pågældende organisations øvrige grafiske stil på World Wide Web.

Brugerfladens grafiske elementer og operationerne filtreres nemt sammen, og det bliver omstændeligt at variere delene uafhængigt af hinanden. Det er f.eks. tilfældet, hvis alt vedrørende en enkelt operation er samlet i én enkelt JSP-fil, og hver operations JSP-fil indeholder eksakte kopier af de grafiske elementer i brugerfladen [Malani '02, Solution 1+2].

Tæt sammenkobling af operationer og brugerfladens øvrige elementer kan også give vedligeholdelsesproblemer, når flere operationer skal bruge de samme elementer. I stedet skal de forskellige komponenter i brugerfladen afkobles, så de kan variere uafhængigt og nemt vedligeholdes uafhængigt af hinanden. Udviklerne skal kunne vedligeholde og udvikle operationerne, mens tekstforfatterne kan arbejde med teksterne og brugerfladedesignerne med forskellige grafiske design. Hvert komponent i brugerfladen skal, som argumenteret i "**Pull MVC**" [Turbine], have et velafgrænset og veldefineret ansvarsområde indeholdende elementer, der refererer til bagvedliggende informationsbehandling.

4.3.2 Tilstandsbaseret Controller

En selvbetjening kendetegnes blandt andet ved, at den stiller en række muligheder til rådighed for brugerne. Brugere kan via de forskellige muligheder forespørge på og påvirke de relationer og informationer, de har hos organisationen.

Fra brugerfladen sendes brugerens forespørgelse til Controller-laget i webapplikationen. Afviklingen af forespørgelsen er afhængig af de forretningsmæssige regler og forudsætninger for brugeren. Dels er der sikkerhedsovervejelser, da operationer kun må udføres hvis brugeren er genkendt og eventuelt har gennemgået en bestemt rækkefølge af tidligere operationer, og forespørgelsen skal undersøges for, om den indeholder data og informationer, som er korrekt udfyldt, specielt i situationer, hvor

brugeren selv har mulighed for at indtaste informationer. Alle de forskellige hensyn skal tages, før den bagvedliggende transaktion kan udføres, og den efterfølgende tilbagemelding sendes til brugeren.

Der er mange ens træk i den afviklingen af hver forespørgelse. For at gøre det nemmere at udvikle og vedligeholde en selvbetjening, vil det være en fordel, om der var en generel løsning, som med en systematisk struktur kunne understøtte afviklingen af regler og forudsætninger i Controller-laget, så de løbende kan udbygges og vedligeholdes.

4.4 Konklusion

Strukturen i WS-frameworket skal baseres på MVC Model 2-arkitekturen, og frameworket skal bestå af to velafgrænsede framework-løsninger til henholdsvis View- og Controller-lagene, da det er i disse lag, de generelle problemstillinger omkring et applikationsnært, brugertilpasset og lagdelt framework til selvbetjening håndteres.

Det overordnede design af WS-frameworket skal være applikationsorienteret og tilbyde direkte og indirekte brugertilpasset funktionalitet til udviklingen af webbaseret selvbetjening. Det skal være lagdelt så det er muligt at tage udgangspunkt i både J2EE og Struts som basisapplikationsframework, men konstrueret så de forskellige selvbetjeningsoperationer bliver uafhængig af de generelle hot spots i J2EE og Struts.

5 Et framework over frameworks

Et af kravene til det overordnede design af WS-frameworket er at, det skal være applikations-orienteret og tilbyde hot spots tilpasset webbaserede selvbetjeninger. MVC Model 2-arkitekturen benyttes til at opnå en struktureret opdeling af dels View-laget og dels Controller-laget, hvor sidstnævnte håndterer hver operation ved en trinvis afvikling af forretningsmæssige regler og forudsætninger. WS-frameworkets Controller-lag skal således indeholde hot spots til applikationerne, hvor applikationerne kan koble sig på afviklingen indeholdt i frameworket.

Udover at WS-frameworket skal være applikationsorienteret, skal det indgå som et framework over de to basisframeworks J2EE og Struts, som hver især tilbyder mere generelle hot spots til mange slags webapplikationer. På denne måde kan konkrete selvbetjeninger tage grundlæggende udgangspunkt i enten J2EE eller Struts og bygge videre på WS-frameworkets faciliteter til selvbetjening. WS-frameworket skal således indgå i den konkrete applikation som et supplerende abstraktionslag ovenpå enten J2EE eller Struts og tilbyde en fælles indpakning af de hot spots, der måtte være i de to basisframeworks.

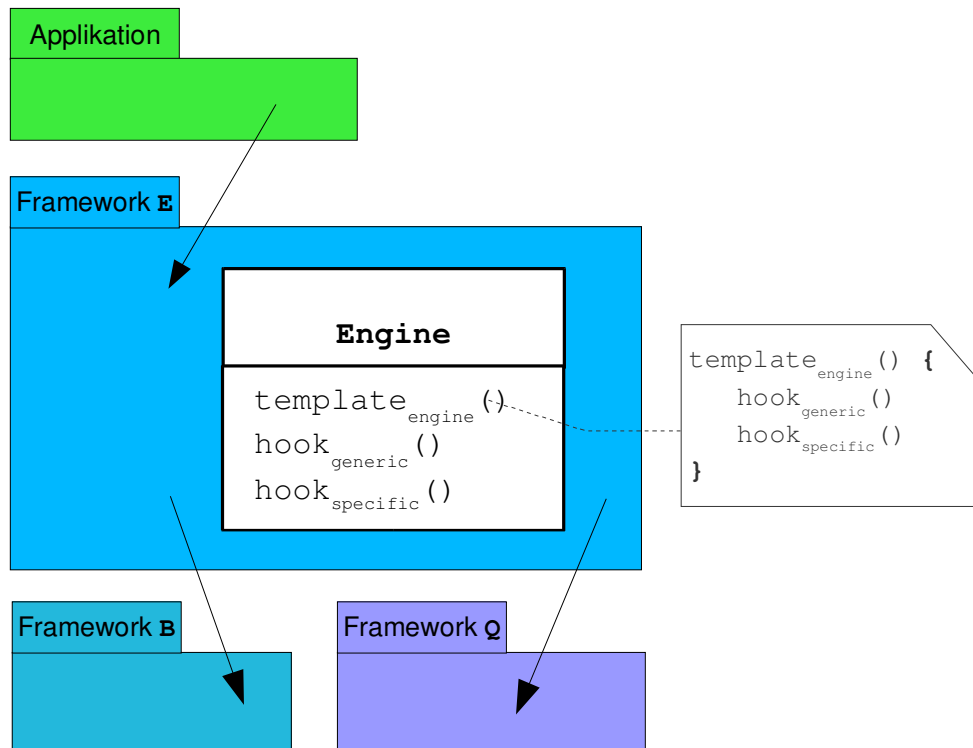
Dette kapitel betragter først analysen og formaliseringen af den generelle problemstilling. I den forbindelse etableres en grundkonstruktion, som diskuteres før designet fastlægges. Designet af et framework over frameworks gennemgås dernæst i detaljer og der implementeres en prototype, som illustreres af en konkret applikation.

5.1 Analyse

For at opnå en ensartet indpakning af de hot spots, der måtte være i de to basisframeworks, J2EE og Struts, diskuteres i det følgende en formaliseret løsning, som er uafhængig af WS-frameworkets konkrete problemstilling. Formaliseringen af to frameworks med whitebox-hot spots leder frem til beskrivelsen af en grundkonstruktion til abstraktion over to frameworks. Grundkonstruktion diskuteres ud fra anvendelsen af de to variationer over Template Method-design patternet og der opstilles en endelig samlet løsning til kombinationen af de to eksisterende whitebox-hot spots.

5.1.1 Formalisering

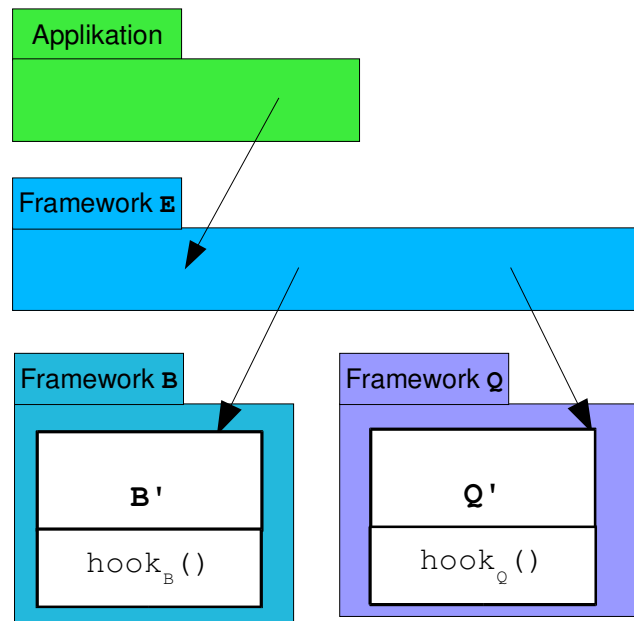
Målet for indpakningen af to basisframework, **Q** og **B**, i et overordnet framework, **E**, er at opnå en indpakning af den generelle funktionalitet på tværs af de forskellige basisframeworks og tilbyde nye hot spots, der er specifikke for applikationer indenfor **E**-frameworkets domæne.



Figur 10: Delene i E

Om frameworket E antages det, at indeholde den generelle funktionalitet i metoden `hook_generic ()` og de applikationsspecifikke trin i metoden `hook_specific ()`, hvor sidstnævnte skal benyttes som hot spot af klasser i applikationen. De to metoder `hook_generic ()` og `hook_specific ()` er hook-metoder i forhold til template-metoden, `template_engine ()`, der indeholder beregningen af sammensætningen af de to hook-metoder. Klassen indeholdende disse tre metoder navngives Engine, da den udfører og sammensætter de forskellige delementer.

Det antages, at basisframeworket Q har et whitebox-hot spot i klassen Q' indeholdende metoden `hook_Q ()`. Tilsvarende har basisframeworket B et whitebox-hot spot i klassen B' indeholdende metoden `hook_B ()`.

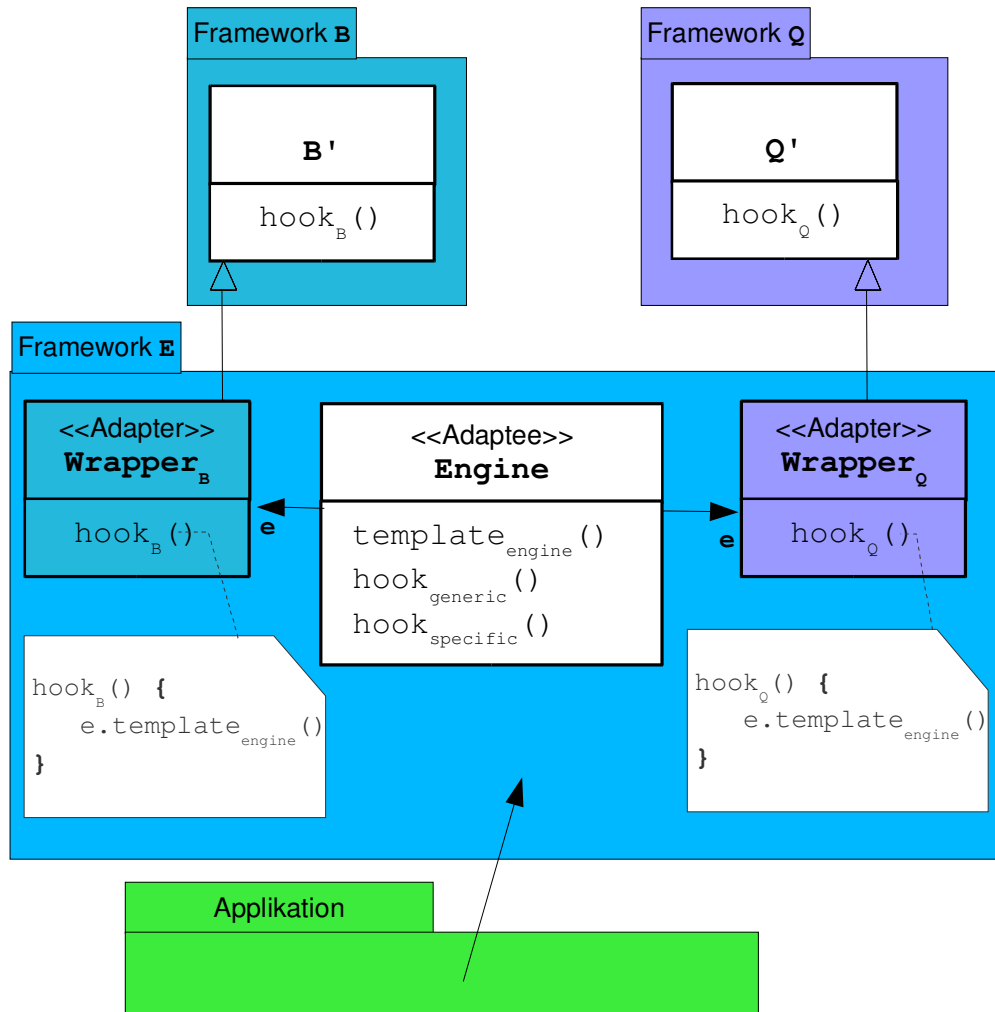


Figur 11: Delene i Q og B

5.1.2 Grundkonstruktion

I frameworket **E** konstrueres ud fra **Adapter**-design patternet [Gamma+'95] to **wrapper**-klasser, $Wrapper_Q$ og $Wrapper_B$, som implementationer af de to whitebox-hot spots i henholdsvis basisframeworket **Q** og **B**. Hver wrapper-klasse indeholder en reference til en `Engine`-klasse med en `template_engine()`-metode som de to hook-metoder i wrapper-klasserne kalder. Se figur 12.

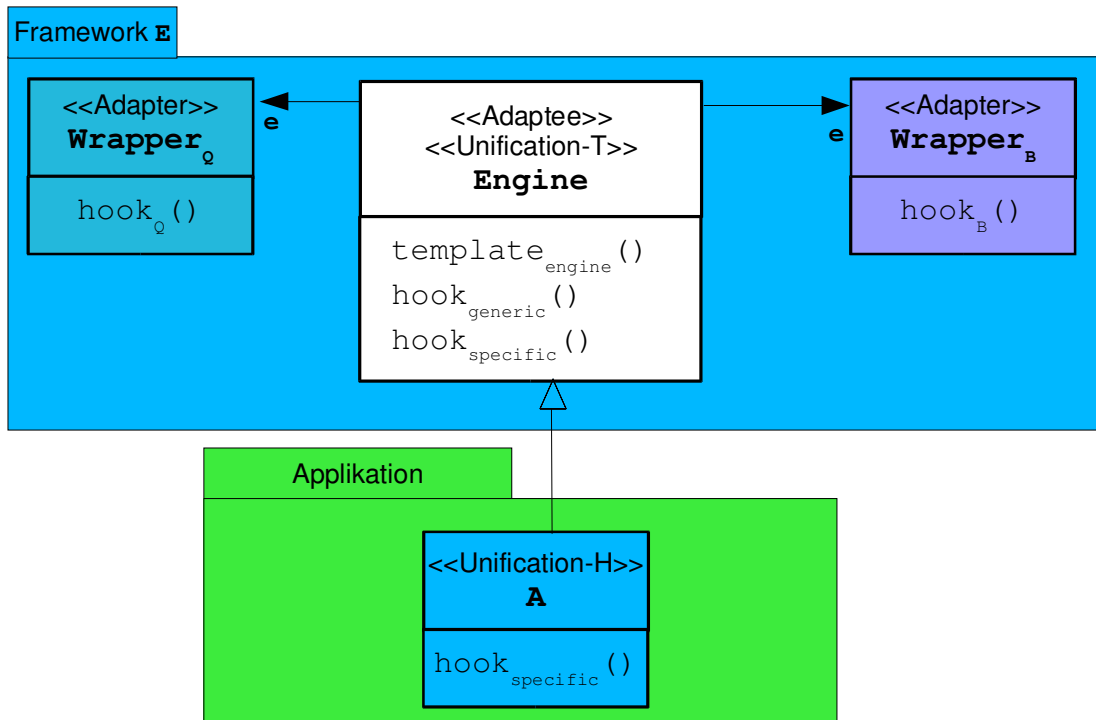
En umiddelbar måde at kombinere de to wrapper-klasser på ville være at tilknytte klassen $Wrapper_Q$ direkte til $Wrapper_B$ -klassen ved komposition (eller omvendt). Så kunne ovennævnte `template`- samt `hook`-metoder implementeres direkte i $Wrapper_Q$ -klassen og bruges direkte som udgangspunkt for konkrete applikationer. Dette ville resultere i en sammenblanding af begreber fra de forskellige basisframeworks og en høj risiko for et design, der genkendes fra anti-patterns som: **The Blob, Swiss Army Knife, Cut & Paste**. Det er ligeledes vigtigt, at hverken `hook_B()` eller `hook_Q()` metoderne er direkte ansvarlige for kald af hook-metoderne i `Engine`-klassen, da det vil give vedligeholdelses- og udvidelsesproblemer, hvis og når `template_engine()`-metoden ændres.



Figur 12: Samleklassen Engine

5.1.3 Unification

Ved brug af Unification-variationen over Template Method-design patternet konstrueres Engine, så den indeholder dels `templateengine()`-metoden, og de to hook-metoder `hookgeneric()` og `hookspecific()`. Metoden `hookQ()` i `WrapperQ` og metoden `hookB()` i `WrapperB` kalder `templateengine()` i klassen Engine, som efterfølgende kalder metoderne `hookgeneric()` og `hookspecific()` på passende vis. Konkrete applikationer kan nu udnytte Engine-klassen som whitebox-hot spot og implementere de konkrete `hookspecific()`-metoder i en konkret klasse A. Se figur 13.

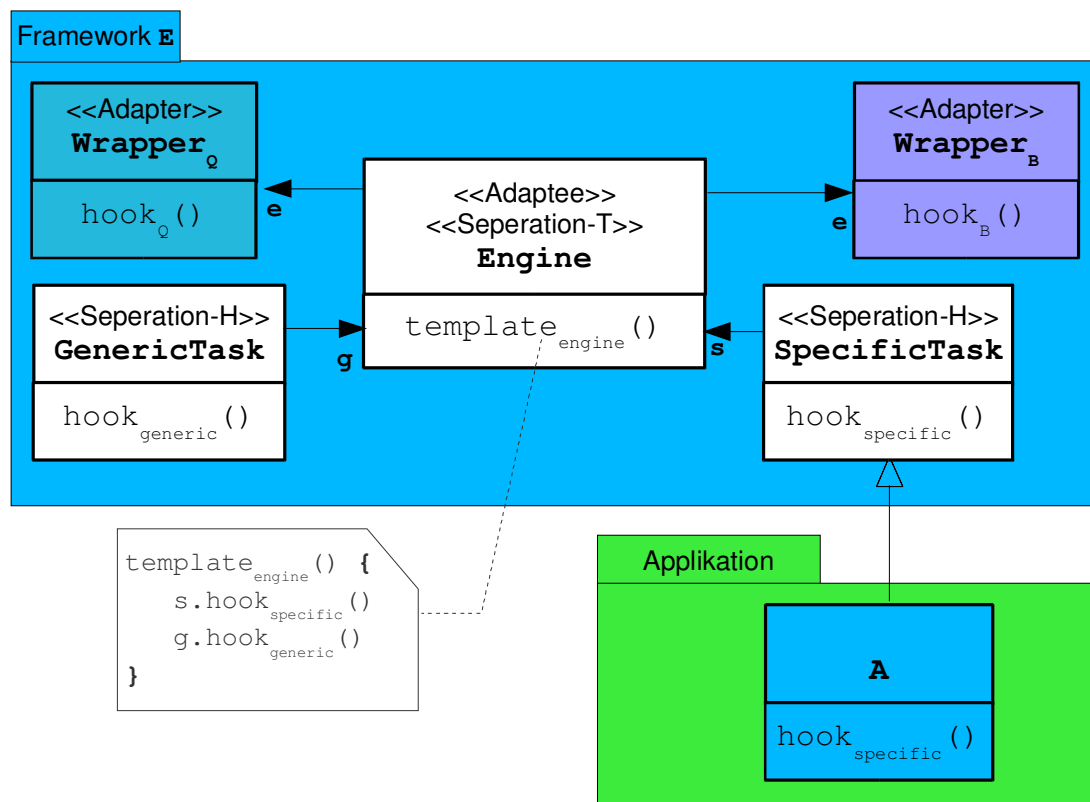


Figur 13: Design ud fra Unification

Da `Engine`-klassen indeholder både de specifikke trin og de generelle trin bliver det omstændeligt at ændre både hvilke generelle og hvilke specifikke trin, der skal benyttes i en konkret subclasse `A`. Der er behov for et design, hvor de generelle trin bliver afkoblet fra de specifikke trin.

5.1.4 Separation

Ved brug af Separation konstrueres `Engine`-klassen med to støtteklasser, `GenericTask` og `SpecificTask`. De to støtteklasser indeholder hver sin `hook`-metode, henholdsvis `hook_generic()` og `hook_specific()`. `Template`-metoden `template_engine()` implementeres fortsat i `Engine`-klassen ligesom den metode, som klasserne `Wrapper_B` og `Wrapper_Q` benytter sig af. Med udgangspunkt i klassen `SpecificTask` tilbydes nu et nyt whitebox-hot spot, og de konkrete `hook_specific()`-metoder implementeres i en konkret klasse `A`. Se figur 14.



Figur 14: Design ud fra Seperation

Da `SpecificTask`-klassen kun indeholder de specifikke trin, er der en klarere snitflade til de konkrete klasser **A** i forhold til brugen af Unification ovenfor. Denne løsning har i stedet en høj kompleksitet, da der dels er mange klasser i løsningen og en højere grad af delegering. Det virker unødvendigt at afkoble de generelle trin fra `Engine`-klassen.

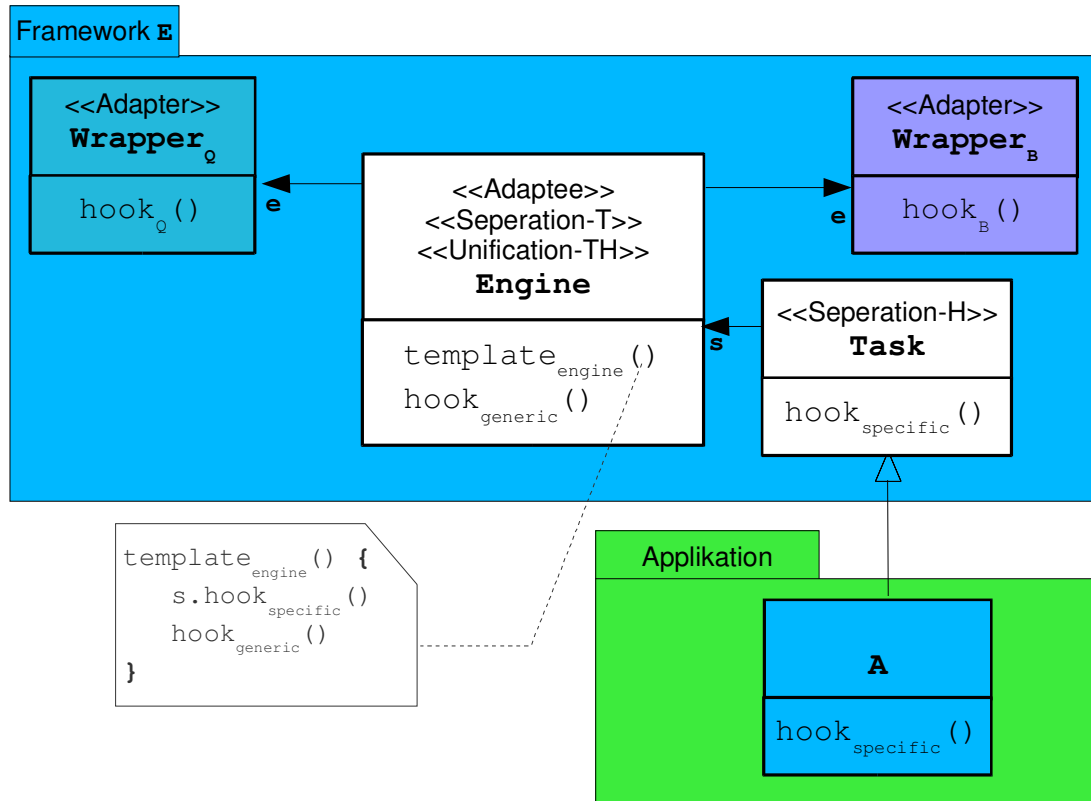
5.2 Design

Ud fra analysen af kombinationen af to basisframeworks i et samlet framework kan der beskrives et generelt design, som benytter en kombination af Unification og Separation. Løsningen suppleres med støtteklasser til parameterhåndtering og støttemetoder til håndtering af delegering mellem de enkelte frameworks.

5.2.1 Kombination

Ved at bruge Unification i forbindelse med de generelle trin og bruge Separation i forbindelse med de specifikke trin kan der opnås et mere fleksibelt og gennemskueligt design. Således placeres `hookspecific()` i en separat klasse, `Task`, der tilknyttes ved komposition til klassen `Engine`, mens `hookgeneric()`-metoden placeres ved Unification

direkte i `Engine`-klassen. Template-metoden implementeres fortsat i `Engine`-klassen som den metode, som klasserne `WrapperB` og `WrapperQ` benytter sig af. Se figur 15.



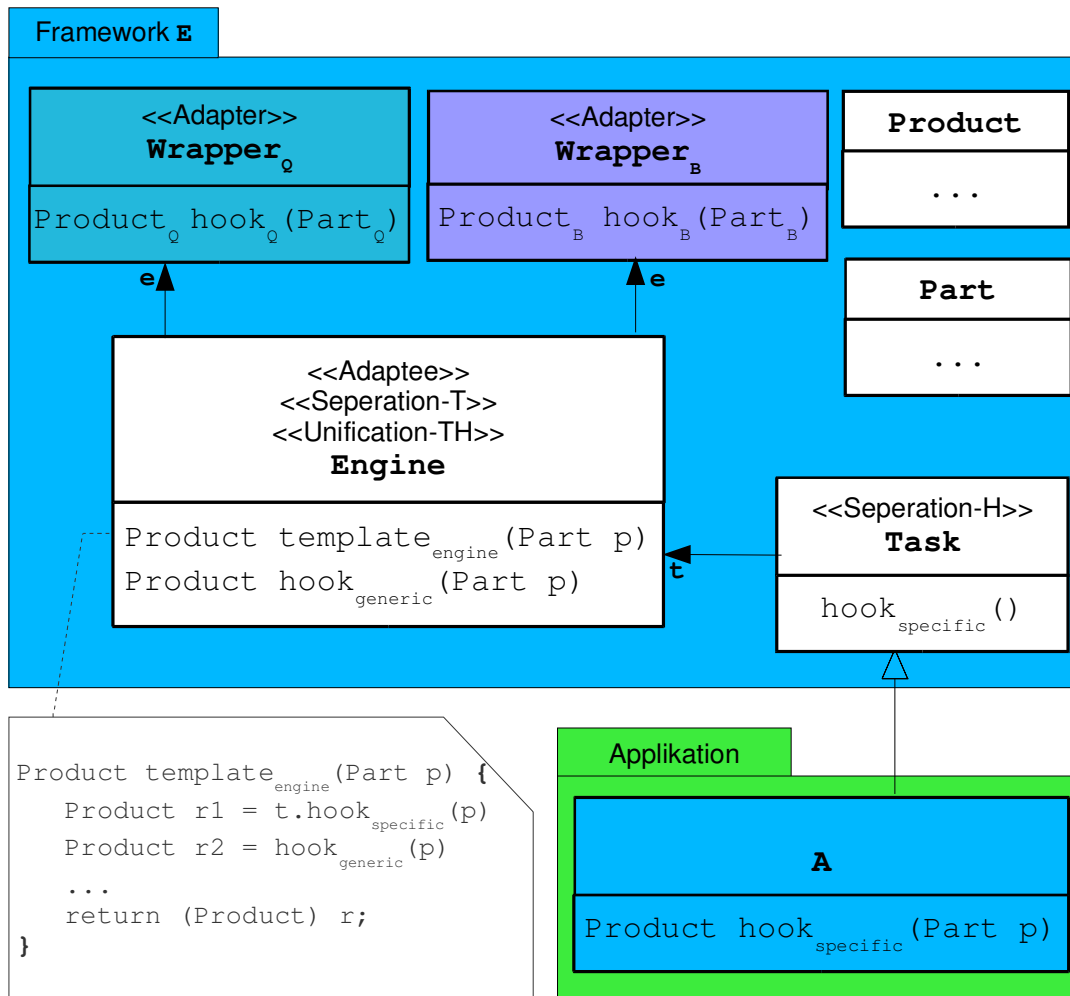
Figur 15: Kombinationsdesign

Ved denne kombination af Unification og Separation tilbyder frameworket **E** netop et hot spot med de specifikke trin i klassen `Task`, mens de generelle trin er indkapslet i `Engine`-klassen. Alt efter konkret sammenhæng kan `Task`-klassen tilbydes som et whitebox-hot spot ved klasse- eller interfacedarvning. Ud fra klassen `Task` som hot spot kan der bygges applikationer med udgangspunkt i domænet for frameworket **E**. Er der eventuelt behov for at tilpasse de generelle trin beskrevet i metoden `hookgeneric()` mere præcist, kan der ud fra `Engine`-klassen etableres en yderligere specialisering til dette formål.

5.2.2 Parameterstyring

Frameworket **E** er konstrueret, så de forskellige basisframeworks ikke længere bruges som udgangspunkt for de konkrete klasser i applikationen. I stedet er der nu kun én enkelt klasse for hvert basisframework, der benytter de oprindelige hot spots. Den konkrete applikations bestanddele konstrueres ud fra **E**-frameworkets hot spot i klassen `Task`.

I de konkrete implementationer af metoden `hookspecific()` skal der udføres nogle trin, som før var udført i en klasse nedarvet fra enten klassen Q' eller klassen B' . For at sikre, at trinene kan udføres i den konkrete `hookspecific()`-metode, skal parametrene til henholdsvis metoden `hookB()` og `hookQ()` videreføres som parameter til `templateengine()`-metoden og dernæst til den konkrete `hookspecific()`-metode.

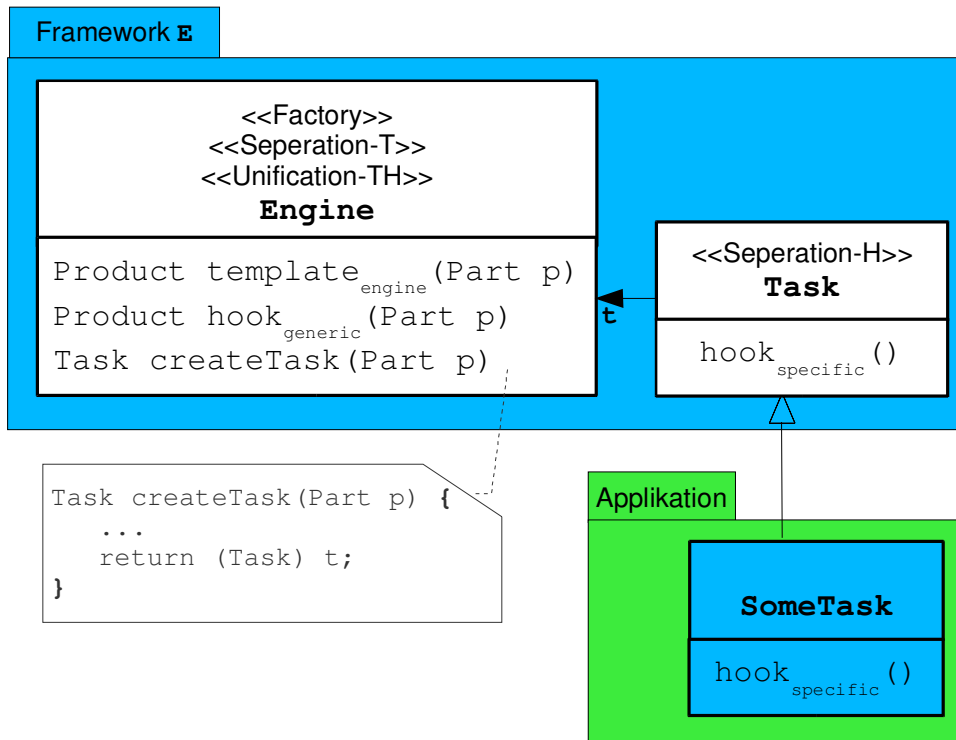


Figur 16: Parameterstyring

For at undgå, at signaturen på metoderne i frameworket kommer til at behandle overordnede typer (f.eks. `java.lang.Object` [J2SE]) beskrives en fælles repræsentation, en klasse `Part`, af de parametre, metoderne `hookB()` og `hookQ()` har til fælles. Ligeledes findes en fælles klasse, `Product`, til returnværdierne for metoderne, så returnværdien ikke skal typecastes efter metoden `templateengine()` er udført.

5.2.3 Kaldstyring

Udover videreførelsen af parametre til udførelsen af `hookspecific()`-metoderne, skal kaldstyringen delegeres fra de forskellige basisframeworks og videre til subclasserne konstrueret på baggrund af klassen `Task`. Oprettelsen af en konkret subklasse `SomeTask` baseret på `Task` udføres i metoden `createTask()`, som er en **Factory Method** [Gamma '95], for `Task`-objekter i den konkrete applikation. Factory-metoden placeres enten i `Engine` eller i en klasse for sig selv.



Figur 17: Kaldstyring

Metoden `createTask()` kaldes enten fra `Engine` eller fra en specialiseret subklasse af `Engine`-klassen. Det bør undgås, at der placeres logik for frameworket `E` i klasserne `WrapperB` eller `WrapperO`, da det vil blande indpakningen af de to basisframeworks med `E`-frameworkets indkapslede funktionalitet. Det er kun i metoden `createTask()`, at der er viden om præcis hvilke variationer af klassen `Task`, der kan oprettes. Den konkrete `Task`-klasse i applikationen har ikke behov for at vide, hvordan den oprettes som en blandt mange.

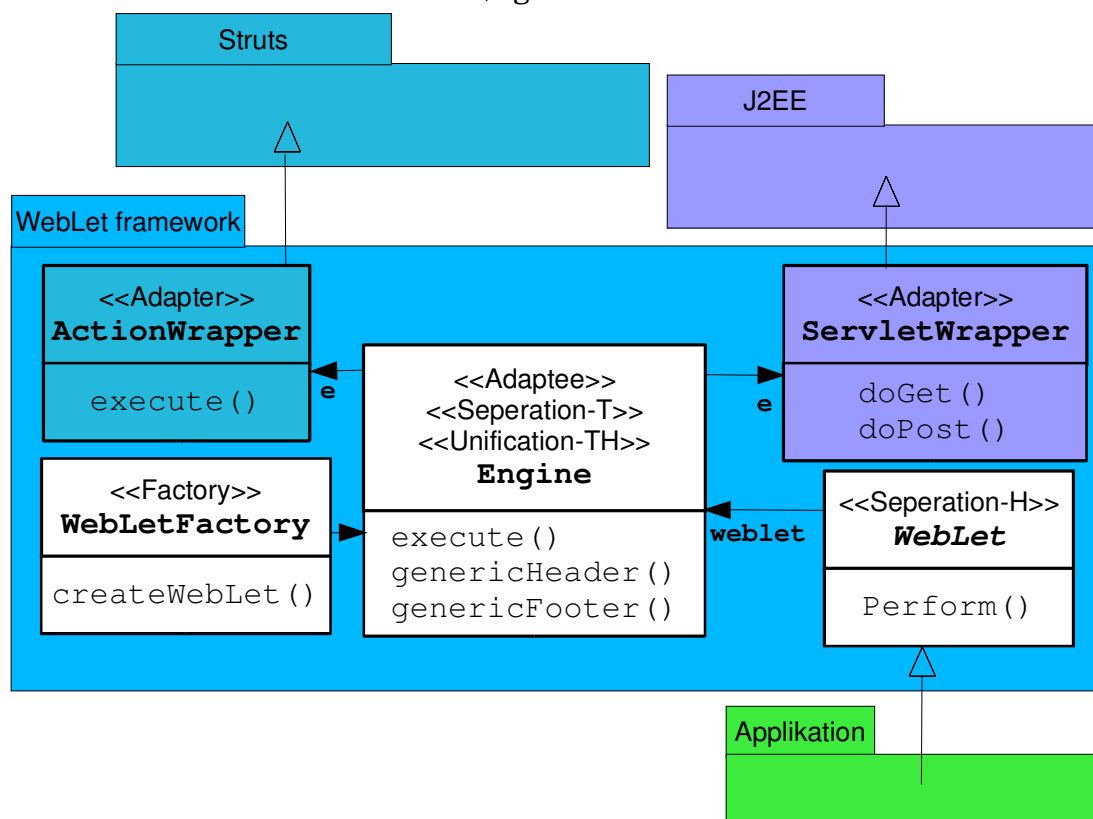
5.3 Implementation

Ovenfor beskrives analysen og designet af en generel løsning til konstruktionen af et nyt overordnet framework, **E**, ovenpå to basisframeworks, **Q** og **B**. Designet kan bruges til at konstruere et nyt framework ovenpå de to webapplikation-frameworks, **J2EE** og **Struts**, som ofte benyttes i forbindelse med udvikling af webbaseret selvbetjening.

Løsningen demonstreres nedenfor i et **WebLet**-framework. Formålet for **WebLet**-frameworket er at stille minimal funktionalitet til rådighed til webapplikationer, uanset om applikationen er baseret på **J2EE** eller **Struts**. **WebLet**-frameworket skal primært kunne benyttes til at lave "Hello World" webapplikationer med, hvor **HTML** i top og bund er forudbestemt.

5.3.1 WebLet-frameworket

WebLet-frameworket består af følgende klasser:



Figur 18: Klassediagram

- **WebLet, whitebox-hot spot mod applikationerne (løsningens Task):**

```

001 package framework;
002 // $Id: WebLet.java,v 1.2 2003/02/12 18:29:46 jottosen Exp $
003
004 [...] //imports
005
006 public abstract class WebLet {
007
008

```

```

009     protected PrintWriter out;
010
011     public abstract void perform(HttpServletRequest req);
012
013     protected final void setPrintWriter(PrintWriter pw) {
014         out=pw;
015     };
016
017 };

```

- Engine, indeholder de generelle hook-metoder og template-metoden:

```

001     package framework;
002     // $Id: Engine.java,v 1.6 2003/02/12 18:29:46 jottosen Exp $
003
004     [...] //imports
005
006     class Engine {
007
008         protected void execute(HttpServletRequest req, HttpServletResponse res)
009             throws ServletException, IOException {
010
011             // Prepare out
012
013             [...] // Extract parameter, create factory and call factory method
014
015             [...] // Call hook methods
016             genericHeader(out);
017             weblet.perform(req);
018             genericFooter(out);
019         };
020
021         protected void genericHeader(PrintWriter out) {
022             out.println("<HTML><HEAD>");
023             out.println("    <TITLE>WebLet Framework</TITLE>");
024             out.println("</HEAD><BODY>");
025         };
026
027         protected void genericFooter(PrintWriter out) {
028             out.println("</BODY></HTML>");
029         };
030
031     };
032
033 };

```

- ServletWrapper, er Wrapper-klassen for J2EE's whitebox-hot spot:

```

001     package framework;
002     // $Id: ServletWrapper.java,v 1.3 2003/02/12 18:29:46 jottosen Exp $
003
004     [...] //imports
005
006     public class ServletWrapper extends HttpServlet {
007
008         Engine engine = new Engine();
009
010         public void doGet(HttpServletRequest req, HttpServletResponse res)
011             throws ServletException, IOException {
012             doPost(req,res);
013         };
014
015         public void doPost(HttpServletRequest req, HttpServletResponse res)
016             throws ServletException, IOException {
017             engine.execute(req, res);
018         };
019
020     };

```

- ActionWrapper, er Wrapper-klassen for Struts' whitebox-hot spot:

```

001     package framework;
002     // $Id: ActionWrapper.java,v 1.2 2003/02/12 18:29:46 jottosen Exp $
003
004     [...] //imports
005
006     public class ActionWrapper extends Action {
007
008         Engine engine = new Engine();
009
010         public ActionForward execute(ActionMapping mapping,
011             ActionForm form,
012             HttpServletRequest req,

```

```

018                                     HttpServletResponse res)
019     throws Exception {
020         engine.execute(req, res);
021         return null;
022     };
023
024 };
    
```

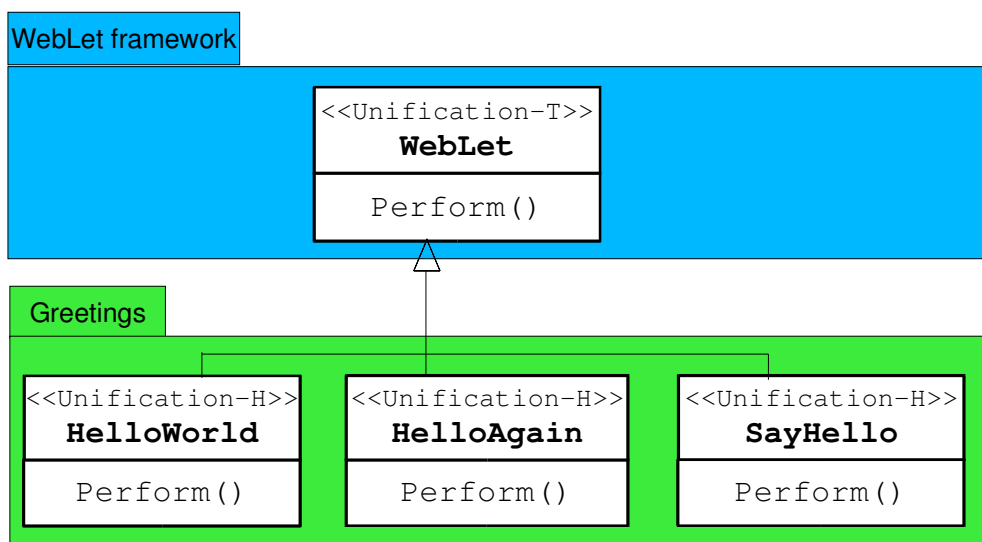
- `WebLetFactory` indeholder `FactoryMethod` for `WebLet`-klasserne og en intern repræsentation af tilgængelige `WebLet`-klasser (Se kapitel 10.1).

Grundkonstruktion i **WebLet**-frameworket i udarbejdet ifølge designet ovenfor, med den undtagelse at `Factory Method`-metoden til `Task`-objekter er placeret eksternt af `Engine`-klassen. Baggrunden for dette er at få en klasse uafhængigt af `Engine`, som har ansvaret for, hvilke `WebLet`-klasser, der findes, og hvordan de oprettes.

I forbindelse med parameterhåndteringen benytter **WebLet**-frameworket en parameter af typen `HttpServletRequest [J2EE]`, som `Part`-element. `HttpServletRequest` er således fællesnævner for, hvad der videreføres til **WebLet**-frameworkets `execute()`-metode fra de to basisframeworks. Resultatet af afviklingen i `execute()`-metoden er i dette tilfælde `null`, altså er `Product`-begrebet fra løsningen ovenfor ikke umiddelbart benyttet.

5.3.2 Greetings-applikationen

Med udgangspunkt i et whitebox-hot spot i **WebLet**-frameworket implementeres en lille "Hello World" applikation, **Greetings**, som består af følgende klasser:



Figur 19: Klassediagram

- HelloWorld, skriver "Hello World" i browseren.
- HelloAgain, tæller antallet af HelloAgain i samme session (udledt af HttpServletRequest-parameteren).
- SayHello, hvor der kan indtastes hvad der skal stå efter "Hello":

```

001 package application;
002 // $Id: SayHello.java,v 1.2 2003/02/12 18:29:46 jottosen Exp $
003
004 [...] //imports
005
006 public class SayHello extends WebLet {
007
008     public void perform(HttpServletRequest req) {
009         String name = req.getParameter("name");
010         if (name == null) {
011             [...]
012             } else {
013                 if (name.length() == 0) { name = "World"; };
014                 out.println(" <h1>Hello "+name+"</H1>");
015             };
016         };
017     };
018 };
019
020 };
021
022 };
023
024 };

```

Resultatet af eksempelvis SayHello er således følgende HTML:

```

<html><head>
  <title>WebLet Framework</title>
</head><body>
  <h1>Hello Jesper</h1>
</body></html>

```

Den komplette dokumentation og kildekode til **Greetings**-applikationen kan ses på den vedlagte cd-rom (Se kapitel 10.1), mens brugen af applikationen installeret på en Tomcat servlet-container er beskrevet i afsnit 10.2.

5.4 Konklusion

I ovenstående beskrives en løsning på hvordan to whitebox basisframeworks kombineres i en samlet frameworkløsning baseret på de to design patterns Template Method og Adapter. udover to støtteklasser (Part og Product) består **WebLet**-frameworket af en abstraktion over de forskellige basisframeworks (Wrapper-klasserne). **WebLet**-frameworket indeholder en indkapsling af generel funktionalitet (Engine-klassen) og der tilbydes et nyt whitebox-hot spot (Task). Designet består af to lag objekt-orienteret delegering, og begge bidrager til løsningens fleksibilitet.

5.4.1 Relateret forskning

Udover Template Method og Adapter har løsningen ovenfor fællestræk med de to design patterns **Bridge** og **Facade** [Gamma '95]. I Bridge beskrives en afkobling af de abstrakte dele fra de konkrete dele. I frameworket **E** bruges en Bridge-lignende konstruktion mellem de (i forhold til frameworkets domæne) abstrakte `Wrapper`-klasser og de konkrete `Task`-klasser (via den mellemliggende klasse `Engine`).

Klassen `Task` er en Facade i forhold til de konkrete applikationer, da den definerer en simpel adgang til subsystemet **E**, der er "godt nok" for de fleste anvendelser [Gamma '95, s.186]. Et hot spot har ofte karakter af en Facade, da den opdeler applikationen i subsystemer, lag eller basisframeworks, alt efter foretrukket anskuelse.

Den kombination af basisframeworks, der benyttes i løsningen, kan betragtes som en arkitekturmæssig overbygning på en række basisframeworks, som ikke benyttes samtidigt, men som giver det samlede framework flere arkitekturmæssige platforme. Dette er til forskel fra de problemer, erfaringer og løsninger til kombination af flere samtidige frameworks i en applikationen, som umiddelbart er i fokus i frameworklitteraturen (bl.a. [Bosch+ '99] [Mattsson+ '99]).

5.4.2 Evaluering

Den beskrevne løsning kan bruges til konstruktionen af WS-frameworkets Controller-lag, da det dels skal være en arkitekturmæssig overbygning på de whitebox-hot spots i J2EE og Struts, og dels skal tilbyde nye hot spots, som er tilpasset WS-frameworkets domæne uafhængigt af Struts og J2EE.

I forbindelse med J2EE, som i sin opbygning er forberedt til at skulle være basisframework, giver ovenstående design en velafgrænset abstraktion af J2EE's basisfunktionalitet. `WebLet`-frameworket tilbyder hot spots ud fra en konstruktion, der behandler input data som abstraktion ud fra `HttpServletRequest` og udprinter færdig HTML.

I forbindelse med Struts forholder det sig anderledes. Struts er i højere grad end J2EE målrettet udvikling af webapplikationer generelt og har i de primære hot spots forskellige nyttige faciliteter til webapplikationer. Den abstraktion over Struts faciliteter, som `WebLet`-frameworket beskriver, er med udgangspunkt i mindste fællesnævner mellem J2EE og Struts. Umiddelbart gør det Struts supplerende faciliteter overflødige, at de først indpakkes og ikke derefter videreføres som faciliteter i løsningens nye hot spots. Ønskes disse faciliteter bevaret, skal der enten videreføres yderligere Struts-faciliteter til `WebLet`-frameworket eller etableres en anden form for arkitekturmæssig overbygning på Struts.

5.4.3 Fremtidigt forskning

I ovenstående er der kun diskuteret en enkelt type kombination af frameworks, yderligere forskning kunne kigge nærmere på hvordan to blackbox-frameworks kombineres og hvordan et blackbox-framework kombineres med et whitebox-framework, eller omvendt.

I kombinationen af frameworks, uanset brugen af blackbox- eller whitebox-hot spots, kan der tages udgangspunkt i, at det overordnede framework skal indeholde en indkapsling af generelle funktionalitet, mens der skal tilbydes applikationsspecifikke elementer som hot spot til de konkrete applikationer indenfor frameworkets anvendelsesområde. Kombinationen af den generelle funktionalitet og de applikationsspecifikke elementer kan etableres ved en kombination af Template Method-begreberne Separation og Unification, således at dele af det nye framework er i form af blackbox-hot spots, mens andre er whitebox-hot spots.

6 Brugertilpasset View-lag

Kravene til det brugertilpassede View-lag i WS-frameworket er, at laget skal udgøre et afgrænset framework, der skal være orienteret mod webbaserede selvbetjeninger og tilbyde direkte og indirekte brugertilpasset funktionalitet. View-laget skal opdele de enkelte elementer i brugerfladen således, at følgende kan variere og vedligeholdes uafhængigt:

- Indirekte brugertilpasning af grafisk udseende og tekster
- Direkte brugertilpasning af aktuelt sprog på tekster og illustrationer
- En brugertilpasset samling af selvbetjeningsmuligheder

Selvbetjeningens brugerflade er en væsentlig del af organisationens identitet og kontaktflade på World Wide Web og er derfor underlagt de samme layoutmæssige retningslinier som organisationens øvrige World Wide Web præsentation. Dette betyder blandt andet, at selvbetjeningen skal kunne skifte brugerflade hyppigt på grund af sammenlægninger, nye grafiske retningslinier og nye faciliteter. Eksempelvis har TDC's internet portal skiftet udseende ca. hver sjette måned de seneste par år, og selvbetjeningen har skullet skifte udseende lige så tit.

I flere selvbetjeninger ses det, at den samme selvbetjening har flere forskellige udseender eller kan benyttes på flere sprog. Hos Danske Bank findes i forskellige versioner af den samme selvbetjening indenfor samme organisation, to på dansk (Danske Bank og Girobank) og en på norsk (Fokus Bank). Hos Nordea afgør relationen ("Basis" eller "Fordel"), hvilke operationer der stilles til rådighed i selvbetjeningen, mens udseende og tekster er ens for alle.

Efter en kort indledende analyse og formalisering af brugerfladens forskellige dimensioner gennemgås de tre komponenter som View-laget i WS-frameworket skal bestå af. Som fast skabelon indeholder gennemgangen af hvert komponent en beskrivelse og diskussion, definition og implementation af TagLib-mærkater, et eksempel, gennemgang af komponentens interaktion og en diskussion af alternativer i implementationen. Afslutningsvist foretages en teknisk evaluering med hensyn til performance og vedligeholdelse.

6.1 Analyse

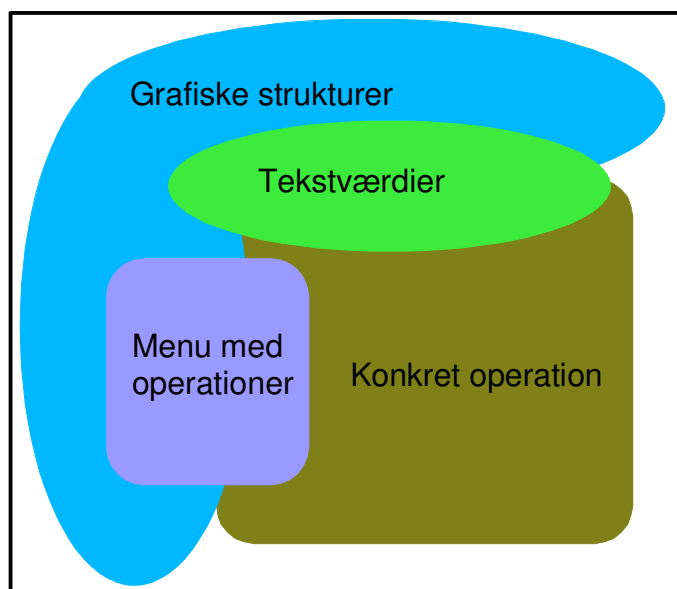
Brugerfladen i en selvbetjening består af grafiske og layoutmæssige strukturer, nogle tekster og illustrationer i et valgt sprog og en samling af tilgængelige operationer. For at opdelingen i grafisk udseende, valg af sprog og menupunkter bliver nem at vedligeholde skal JSP-filerne, som View-laget hovedsageligt består af, være kendetegnet dels ved en høj intern

samhørighed (**high cohesion**, [Larman '98]) med hensyn til indholdet af hver JSP-fil og dels ved en indbyrdes lav kobling (**low coupling**, [Larman '98]) JSP-filerne imellem. JSP-filerne kan vurderes med disse egenskaber, da de ved afviklingen omdannes til udført Java-kode, kan indeholde kald af hinandens indhold og stiller indhold til rådighed. Interfacet til en JSP-fil er ikke så omfangsrigt som ved Java-klasser, men dette begrænser ikke vurderingen af JSP-filens interne samhørighed og eksterne kobling.

6.1.1 Formalisering

View-lagets forskellige "sider" består af:

- Den konkrete operation, der stilles til rådighed
- Grafiske strukturer (layouttemplate, grafik, stylesheets, ...)
- Tekstværdier (tekster, knapper m.v., der afhænger af sproget)
- En menu med de tilgængelige operationer



Figur 20: Brugerfladens fire dele

Hver side i den webbaserede brugerflade består således for en given organisation og en given relation af en operation, nogle grafiske og tekstuelle elementer og en menu med valgmuligheder. Nedenstående definition af siden s_i , beskriver, at den består af operationen o_i , et grafisk design d_i , en konkret samling tekster t_i , og at der på siden er adgang til elementer i \mathcal{P} , mængden af tilgængelige menupunkter i selvbetjeningen.

$$s_i = [o_i, d_i, t_i, \mathcal{P}]$$

En anden side, s_2 , i den samme selvbetjening, med samme grafiske design, tekster og menupunkter kan således beskrives ved:

$$s_2 = [o_2, d_2, t_2, \mathcal{P}]$$

Det konkrete grafiske udseende $d_1 \in \mathcal{D}$ mængden af alle de mulige grafiske design. Tilsvarende er t_1 en konkret samling af tekster i mængden \mathcal{T} , af mulige tekstsamlinger. Operationen $o_2 \in \mathcal{O}$ mængden af mulige operationer.

Da alle tilgængelige operationer skal findes i menuen \mathcal{P} og mængden \mathcal{O} netop består af mængden af operationer, er disse to mængder ens. Den samlede brugerfladen i en selvbetjening, \mathcal{S} , kan beskrives som en funktion, \mathcal{F} , af den konkrete organisation og den konkrete relation:

$$\mathcal{S} = \mathcal{F}(\text{Organisation}, \text{Relation}) = [\mathcal{D}, \mathcal{T}, \mathcal{O}]$$

6.1.2 Dimensionerne

For at kunne opfylde kravene til omskifteligheden i selvbetjeningernes brugerflader skal de tre dimensioner ($\mathcal{D}, \mathcal{T}, \mathcal{O}$) kunne varieres og vedligeholdes uafhængigt. Det grafiske design skal kunne varieres uafhængigt af både teksterne og operationerne, da skift af grafik (nyt logo, nyt grafisk layout etc) ikke må bevirke ændringer i hverken operationer eller de forskellige tekster, der skal vises i brugerfladen.

Ligeledes gælder også, at basale ændringer i teksterne ikke må påvirke det grafiske design eller hvilke operationer, der skal præsenteres i menuen. Operationerne i menuen skal på samme vis kunne varieres uafhængigt af det konkrete tekstuelle indhold på siden, hvilket sprog siden bliver vist på, og hvilket konkret grafisk design siden har.

6.2 Grafiske komponenter

Udgangspunktet for den indirekte brugertilpasning af det grafiske udseende, er at de grafiske komponenter i brugerfladen er uafhængige af brugerfladens tekster og muligheder, men i stedet afhænger af brugerens konkrete relation og den pågældende organisation.

De grafiske komponenter kan beskrives i en layoutmæssig template, som indeholder referencer til de forskellige øvrige grafiske virkemidler (Javascrpts, grafikfiler, cascading stylesheets), og som er afstemt med de gældende grafiske retningslinier. Den grafiske template kan desuden indeholde referencer til de brugertilpassede tekster og illustrationer og indeholde pladser til indplacering af de konkrete operationers indhold. Det konkrete indhold og det grafiske layout skal udgøres af enkeltstående JSP-filer med intern høj samhørighed og indbyrdes lav kobling.

6.2.1 Beskrivelse

De grafiske designere i organisationen skal kunne udarbejde en layoutmæssig template-fil i en samlet JSP-fil og afstemme denne med organisationens og/eller relationens grafiske forskrifter. Da den grafiske template kan indeholde supplerende referencer til de andre grafiske elementer, er det tilstrækkeligt at referere til template for at angive et valgt grafisk design.

Et grafisk komponent d_i angiver således en konkret grafisk template afhængig af enten den pågældende organisation eller relation. Mængden \mathcal{D} af alle grafiske udseender kan dermed beskrives som mængden af alle layoutmæssige templates:

$$\mathcal{D} = \{ \text{template } d_1, \text{template } d_2, \dots \}$$

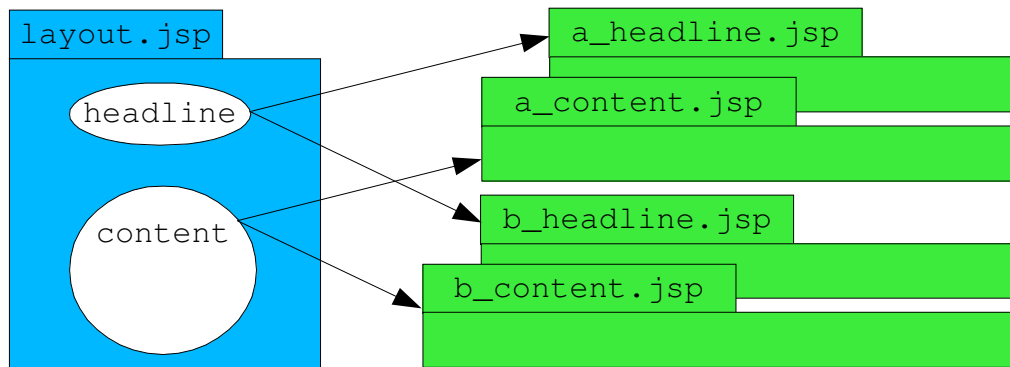
De to næste afsnit diskuterer to mulige tilgange til sammenkobling af JSP-filer med design templates og JSP-filer med indholdet af operationerne. Selvom disse to tilgangsvinkler har nogle pointer, som kan benyttes til en generel afkobling af elementerne, er de ikke tilstrækkelige. I kapitel 6.2.4 beskrives en samlet løsning til afkobling af templates og operationer.

6.2.2 Indhold i det grafiske design

I det grafiske design skal det være muligt at angive den layoutmæssige placering af elementer fra de andre dimensioner i brugerfladen. Det skal være muligt at layoutmæssigt placere dels brugertilpassede tekster og dels det konkrete indhold af den pågældende operation. Der er ofte en fast placering i det grafiske design til operationens overskrift eller indledende beskrivelse, mens operationens aktuelle funktionalitet eller konkrete formular er placeret et andet sted i det grafiske layout.

I den grafiske template er der således behov for at kunne placere operationens enkelte dele, og for at imødekomme dette deles operationen op i mindre JSP-filer, som hver især indeholder netop de dele, som template foreskriver (f.eks. overskrift og indhold). Når hver operation er således delt over flere filer, er det via template, at de enkelte dele af operationen og brugerfladen som helhed sammensættes.

Eksempel: Template `layout.jsp` beskriver placeringen af `headline` og `content`. Operationerne `a` og `b` fordeles på JSP-filerne `a_headline.jsp`, `a_content.jsp`, `b_headline.jsp` og `b_content.jsp`.



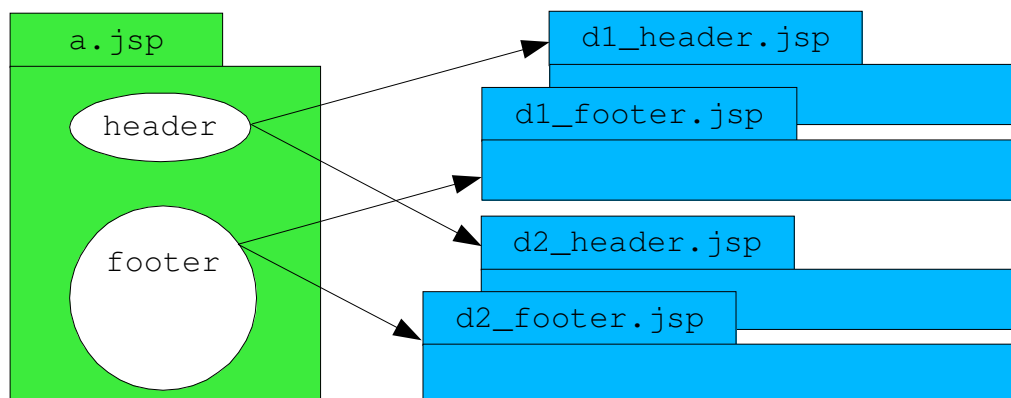
Figur 21: Opdeling af operationer i dele

Opdeles operationerne i selvstændige filer, giver det hurtigt et stort antal filer. Det betyder desuden, at det er template, der har kontrollen over operationen. Operationen kan ikke umiddelbart benyttes selvstændigt, men eksisterer kun udledt fra template.

6.2.3 Grafisk design i indholdet

Det er ikke hensigtsmæssigt at opdele operationerne i flere JSP-filer, da det hurtigt giver stor uoverskuelighed og gør, at operationerne ikke kan tilgås direkte. I stedet ville det være mere hensigtsmæssigt at fastholde operationerne i en fil, så de dermed kan tilgås direkte og uafhængigt af de forskellige templates. Det er således operationen, der skal have kontrollen over den grafiske template. I beskrivelsen af operationen skal der være markeringer af det grafiske designs enkelte dele (f.eks. top og bund). Den layoutmæssige template skal således umiddelbart opdeles i flere JSP-filer, en til hver af disse pladser.

Eksempel: Operationen `a.jsp` indeholder pladser til header og footer. Design templates `d1` og `d2` opdeles i JSP-filerne `d1_header.jsp`, `d1_footer.jsp`, `d2_header.jsp` og `d2_footer.jsp`.



Figur 22: Opdeling af templates i dele

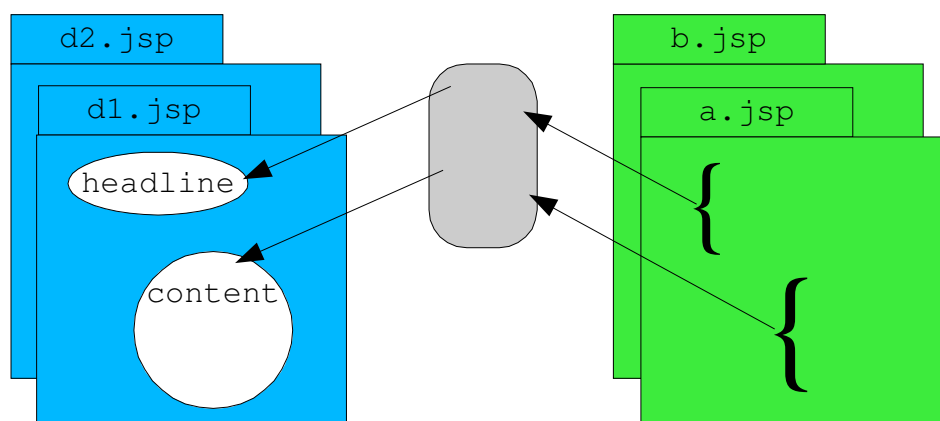
Denne angrebsvinkel gør det grafiske design afhængig af operationen, da det bliver operationerne, der har ansvaret for at sammensætte brugerfladen. Der er lav samhørighed imellem de forskellige grafiske elementer og høj kobling mellem det grafiske designs bestanddele og beskrivelsen af operationen.

6.2.4 Afkobling af grafik og operationer

De to foregående løsninger til sammensætning af grafisk layout og operationernes indhold er ikke ideelle, men har begge nogle pointer, som er væsentlige til en afkobling af elementerne. En pointe i forbindelse med operationerne er, at operationerne kan beskrive, hvilke layoutmæssige dele de består af. Og en pointe ved de grafiske templates er, at de forskellige templates kan beskrive, hvor indholdets forskellige dele placeres layoutmæssigt. Afkoblingen af grafik og operationer skal således beskrives, så både operationer og grafiske templates bevares som enkeltstående filer med intern høj samhørighed og indbyrdes lav kobling.

Løsningen er at beskrive både operationerne og de grafiske design i hver deres samlede JSP-fil. De grafiske design beskriver huller til indsættelse af indhold og operationerne markeringer, der svarer til layoutets huller. Faciliteter uafhængigt af layoutet og operationerne skal sørge for at den rette sammensætning af de to dele foretages. Operationerne kan som ønsket tilgås direkte, men da de kun består af konkret indhold og intet grafisk design, skal den rette layout template aktiveres som det første i opbygningen af brugerfladen.

Eksempel: De to layout templates `d1.jsp` og `d2.jsp` beskriver placeringen af delene `headline` og `content`. Operationerne `a.jsp` og `b.jsp` indeholder markeringer af hvad der konkret skal indgå som `headline` og `content`. Fællesfaciliteter uafhængigt af JSP-filerne sammensætter operationerne med de rette grafiske design.



Figur 23: Opdeling af operationer og templates

6.2.5 TagLib-mærkater

For at sammensætte layoutmæssige templates og beskrivelser af operationer i hver deres JSP-fil, benyttes JSP-teknologien "JSP Tag Library" [TagLib]. Et TagLib er en metode, hvormed der kan udvikles specielle HTML-lignende mærkater til at udvide JSP-filernes funktionalitet.

Følgende JSP-mærkater bruges til de grafiske komponenter:

- `<design:include page="x">`
Placeres øverst i operationernes JSP-filer og angiver, at designet skal aktiveres, og at det initieres fra siden x. Operationens samlede indhold omkranses af mærkaten.
- `<design:markcontent key="y">`
Placeres i operationens JSP-filer og omkranser den del af operationen, som skal pladseres på pladsen y i den grafiske template.
- `<design:placecontent key="z" />`
Placeres i layout templatens JSP-fil og angiver positionen i layoutet, hvor indholdet z skal indsættes.

De tre mærkater beskrives i TagLib [XML]-beskrivelsen `design.tld` ved:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- $Id: design.tld,v 1.6 2003/02/05 20:18:32 jottosen Exp $ -->

[...]

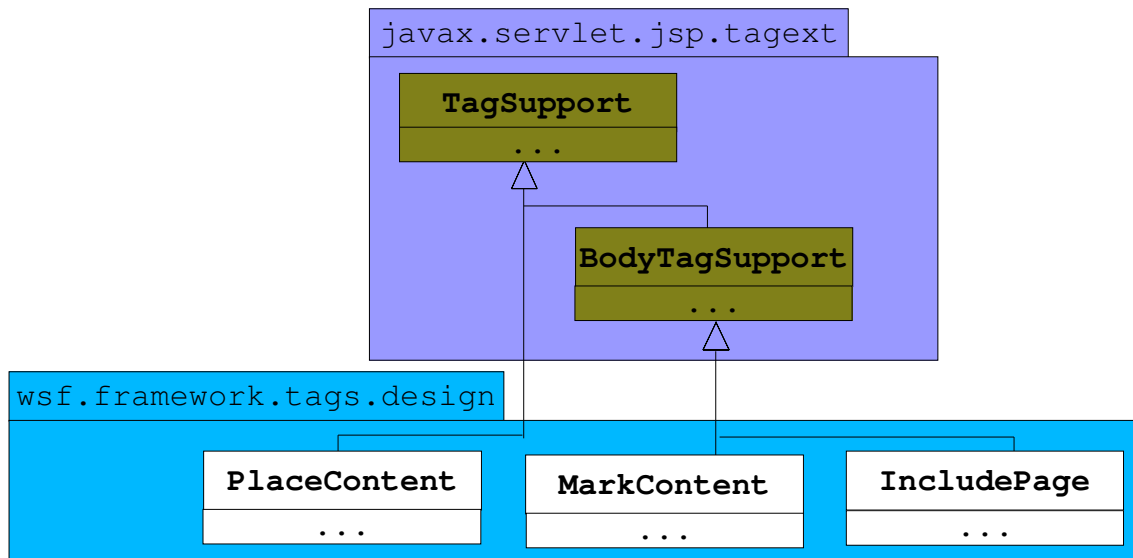
<tag>
  <name>include</name>
  <tag-class>wsf.framework.tags.design.IncludePage</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>page</name>
    <required>false</required>
  </attribute>
</tag>

<tag>
  <name>markcontent</name>
  <tag-class>wsf.framework.tags.design.MarkContent</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>key</name>
    <required>true</required>
  </attribute>
</tag>

<tag>
  <name>placecontent</name>
  <tag-class>wsf.framework.tags.design.PlaceContent</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>key</name>
    <required>true</required>
  </attribute>
</tag>
```

Se afsnit 6.2.7 for eksempler på brug af de forskellige mærkater.

6.2.6 Implementation



Figur 24: Klassediagram

- `wsf.framework.tags.design.IncludePage`
 Når mærkaten aktiveres, evalueres indholdet af mærkaten, og det gemmes i en buffer. Ved afslutning af mærkaten findes den relevante design template, argumentet (page) gemmes, og kontrollen overdrages til design template:

```

001 package wsf.framework.tags.design;
002 // $Id: IncludePage.java,v 1.4 2003/02/12 19:32:38 jottosen Exp $
003
004 [...] //imports
009
010 public class IncludePage extends BodyTagSupport {
011     private String page = "";
012     // Getters & setters
013
014     [...]
021     public int doStartTag() throws JspException {
022         return EVAL_BODY_BUFFERED;
023     };
024
025     public int doEndTag() throws JspException {
026         // Create RessourceReader and extract design
027         ResourceReader rr = new ResourceReader(pageContext);
028         String design = (String) rr.getMapping("designtemplate");
029
030     [...]
034         // include designtemplate in pageContext
035
036     [...]
038         return SKIP_PAGE;
039     };
040
041     [...]
045 }
  
```

- `wsf.framework.tags.design.MarkContent`
 Indholdet af mærkaten gemmes i en buffer, og når mærkaten afsluttes, gemmes bufferen med en nøgle baseret på argumentet (key):

```

001 package wsf.framework.tags.design;
002 // $Id: MarkContent.java,v 1.3 2003/02/12 19:32:38 jottosen Exp $
003
004 [...] //imports
008
009 public class MarkContent extends BodyTagSupport {
010
011     private String key = "";
  
```

```

012
013 // Getters & setters
[... ]
020 public int doStartTag() throws JspException {
021     return EVAL_BODY_BUFFERED;
022 };
023
024 public int doAfterBody() throws JspException {
025     // Create RessourceReader and extract designprefix
026     ResourceReader rr = new ResourceReader(pageContext);
027     String prefix = rr.getDesignPrefix();
028
029     // Remember the bodyContent in Request Scope
030     String pagekey = prefix+"."+key;
031     String body = bodyContent.getString();
032     pageContext.setAttribute(pagekey,body,pageContext.REQUEST_SCOPE);
033     return SKIP_BODY;
034 };
035
036 public int doEndTag() throws JspException {
037     return EVAL_PAGE;
038 };
[... ]
044 }

```

- `wsf.framework.tags.design.PlaceContent`

Når mærkaten afsluttes, benyttes argumentet (key) til at hente en buffer med indhold, som kontrollen overgives til:

```

001 package wsf.framework.tags.design;
002 // $Id: PlaceContent.java,v 1.3 2003/02/12 19:32:38 jottosen Exp $
003
[... ] //imports
009
010 public class PlaceContent extends TagSupport {
011
012     private String key = "";
013
014     // Getters & setters //
[... ]
021 public int doStartTag() throws JspException {
022     return SKIP_BODY;
023 };
024
025 public int doEndTag() throws JspException {
[... ]
030     // Get BodyContent saved in Request Scope
031     String pagekey = prefix+"."+key;
032     String content = (String)
033         pageContext.getAttribute(pagekey,pageContext.REQUEST_SCOPE);
034     if (content == null) { content = "??"+key+"??"; };
035
036     // Get surrounding JspWriter and print the content
[... ]
041     return EVAL_PAGE;
042 };
[... ]
050 }

```

Ved brug af ovenstående tre mærkater afhænger den indirekte tilpasning af brugerfladens grafiske komponenter udelukkende af repræsentationen i klassen `ResourceReader`. Via `ResourceReader` aflæses dels nøglerne til aflæsning af gemte information og dels de konkrete design templates, alt afhængig af den konkrete match i den tilhørende konfiguration. (Den komplette kildekode og dokumentation til mærkaterne, `ResourceReader` og tilhørende støtteklasser findes på den vedlagte cd-rom, se kapitel 10.1.)

6.2.7 Eksempel

Ovenstående mærkater illustreres kort ved nedenstående eksempler. I de to JSP-filer (en template og en operation), er de grafiske mærkater er markeret i **fed**. Yderligere eksempler og detaljer findes i **mdp**-webapplikationen, kapitel 10.2.4.

- `order.jsp`, en formular til indtastning af en ordre:

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/design.tld" prefix="design" %>
<!-- $Id: order.jsp,v 1.3 2003/04/17 11:09:21 jottosen Exp $ -->

<design:include page="order">

<design:markcontent key="headline">
  <h3>Velkommen til din onlinebestilling</H3>
</design:markcontent>

<design:markcontent key="body">
  <table>
  <tr><form action="info.jsp" method="post"><td>
    <b>Bestil</b>
  </td><td>
    <input type="radio" name="select" value="Andemad" checked>Andemad<br>
    [...]
  </td></tr>
  <tr><td colspan="2">
    <input type="submit" value="Indsend">
  </td></form></tr>
  </table>
</design:markcontent>
</design:include>
```

- `design.jsp`, en design template, med placeringer af indhold:

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/design.tld" prefix="design" %>
<HTML><HEAD>
  <!-- $Id: design.jsp,v 1.2 2003/03/01 20:48:41 jottosen Exp $ -->
  [...]
</HEAD><BODY><div align="left">

<design:placecontent key="headline" />

<table border="1" cellpadding="2" cellspacing="2">
<tr><td class="bg0">
  [...]
</td></tr>
<tr><td class="bg2">
  <design:placecontent key="body" />
</td></tr></table>

</div></BODY></HTML>
```

Når disse to JSP-filer kombineres bliver resultatet følgende HTML:

```
<!-- $Id: order.jsp,v 1.3 2003/04/17 11:09:21 jottosen Exp $ -->
<HTML><HEAD>
  <!-- $Id: design.jsp,v 1.2 2003/03/01 20:48:41 jottosen Exp $ -->
  [...]
</HEAD><BODY><div align="left">

<b>Velkommen til din onlinebestilling</b>

<table border="1" cellpadding="2" cellspacing="2">
<tr><td class="bg0">
  [...]
</td></tr>
<tr><td class="bg2">

  <table>
  <tr><form action="info.jsp" method="post"><td>
    <b>Bestil</b>
  </td><td>
    <input type="radio" name="select" value="Andemad" checked>Andemad<br>
```

```
[...]
</td></tr>
<tr><td colspan="2">
  <input type="submit" value="Indsend">
</td></form></tr>
</table>

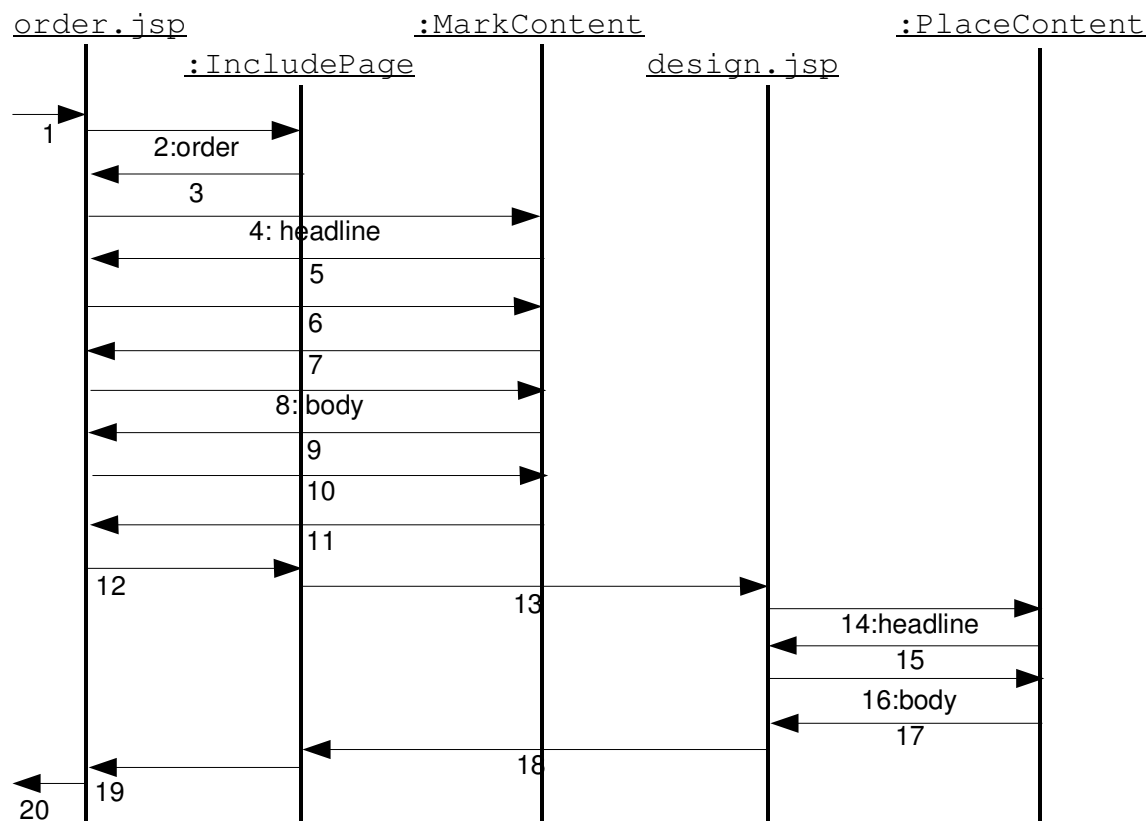
</td></tr></table>
</div></body></html>
```

6.2.8 Interaktion

Den indirekte tilpasning af brugerfladen opnås ved kombinationen af de tre mærkater og en uafhængig aflæsning af tilhørende design template. Mærkaterne har en intern høj samhørighed, en indbyrdes lav kobling og er afhængige af indbyrdes interaktion for at opnå det samlede resultat.

Ovenstående eksempel benyttes til illustration af interaktionen:

1. JSP-filen `order.jsp` aktiveres
2. Mærkaten `<design:include page="order">` initieres
3. Indholdet af mærkaten (2) evalueres
4. Mærkaten `<design:markcontent key="headline">` initieres
5. Indholdet af mærkaten (4) evalueres
6. Indholdet af mærkaten (4) afsluttes
7. Mærkaten (4) afsluttes
8. Mærkaten `<design:markcontent key="body">` initieres
9. Indholdet af mærkaten (10) evalueres
10. Indholdet af mærkaten (10) afsluttes
11. Mærkaten (10) afsluttes
12. Indholdet af mærkaten (2) afsluttes
13. JSP-filen `design.jsp` aktiveres
14. Mærkaten `<design:placecontent key="headline" />` aktiveres
15. Mærkaten (14) afsluttes
16. Mærkaten `<design:placecontent key="body" />` aktiveres
17. Mærkaten (16) afsluttes
18. JSP-filen (13) afsluttes
19. Mærkaten (2) afsluttes
20. JSP-filen (1) afsluttes



Figur 25: Interaktionsdiagram

6.2.9 Alternativer

Opbygningen af brugerfladen i designmæssige og funktionelle dele findes også i de to webapplikation-frameworks Struts og Turbine. Begge tager udgangspunkt i MVC Model 2-arkitekturen og benytter TagLib-mærkater til opbygningen af elementerne i brugerfladen.

Princippet om at brugerfladens grafiske design og operationernes funktionalitet skal være uafhængige ses blandt andet i frameworket Turbine [Turbine]. Turbine benytter en opdeling af brugerfladen i "Page, Layout, Navigation, Screen", men beskriver i sin "MVC 2+1" arkitektur, at arkitekturen er baseret på en sammenlægning af View- og Controller-lag, så der i forbindelse med evalueringen dynamisk kan tages stilling til det resulterende grafiske design og den løbende programkontrol. I modsætning til dette er den beskrevne løsning fokuseret på en indbyrdes lav kobling mellem de JSP-filer, der udgør dels operationerne og dels det grafiske design, under den forudsætning, at det grafiske layout er afhængig af organisationens forretningsområde og ikke en dynamisk evaluering.

Struts indeholder `struts-tiles`, der opdeler det grafiske design og funktionalitet i flere "tiles" (skiver). Tiles kan bruges som alternativ til WS-frameworkets afkobling af det grafiske design og operationer, under forudsætning af at brugerfladen kan opdeles i forskellige "tiles" ("`header.jsp`", "`body.jsp`", m.v.) og at det kun er nødvendigt med et enkelt layout pr. operation. Ved brug af Tiles opdeles operationerne i forskellige dele, som opbevares i hver sin JSP-fil. Ved at bruge Tiles er der således flere JSP-filer at holde rede på end den foreslåede løsning til afkobling af grafisk design og konkret funktionalitet.

Ovenfor gemmes information om den konkrete brugers forretningsmæssige relation i et objekt på servlet-containeren. Alternativt kan denne information gemmes som Enterprise JavaBean som en mere datanær indkapsling af informationerne, specielt hvis data (Model-laget) i selvbetjeningen i øvrigt er baseret på Enterprise JavaBeans. Information om brugerens relationen er data, som aflæses få gange og sjældent ændres indenfor udførelsen af operationen, så derfor kan det være en fordel, at modellere disse informationer mere nært på databasen. Enterprise JavaBeans kan håndteres ressourcemæssigt mere effektivt end objekter i servlet-containeren, som ikke skal være for "tunge" før de sløver afviklingen af servlet-containeren.

6.3 Tekstkomponenter

Tekster og illustrationer i selvbetjeningens brugerflade skal tilpasses brugerens forretningsmæssige relation og brugerens valg af sprog. Alle sproghængige elementer skal kunne indplaceres i beskrivelsen af det grafiske layout og i den konkrete beskrivelse af operationens indhold.

Det er ikke tilstrækkeligt at afgøre brugerens sprog ud fra sproget på browseren eller på operativsystemet. Der er mange brugere, der benytter browsere eller operativ systemer i et sprog, men foretrækker at deres selvbetjening foregår i et andet sprog. De fleste danskere vil umiddelbart foretrække, at deres selvbetjening foregik på dansk, uanset om de brugte engelske operativsystemer eller foretrak engelske browsere uden understøttelse af dansk. Der findes desuden sprog, der næppe vil få deres egne operativsystemer eller sprogpakker til browsere. Sproget i en selvbetjening skal derfor være baseret på brugerens aktive valg.

6.3.1 Beskrivelse

Udvalget af sprog, brugeren kan vælge imellem, er naturligt begrænset af, hvilke sprog organisationen vælger at stille til rådighed for den pågældende forretningsmæssige relation. Men kan brugeren vælge, skal valget være foretaget én gang for alle, og efterfølgende skal det huskes, indtil brugeren aktivt vælger et andet. Dermed også sagt, at hvis der er mulighed for at vælge flere sprog, skal det fremgå af brugerfladen.

Mængden af tekstuelle sammenhænge, \mathcal{T} , udgøres af de tekster, knapper, illustrationer m.v., der skifter afhængigt af de tilbudte sprog. Til hver eneste tekst, knap m.v. kan der tilknyttes en unik nøgle, som benyttes til at referere det specifikke element. På den måde afkobles teksternes konkrete data umiddelbart fra deres placering i andre dimensioner i brugerfladen. Hvert sprog i selvbetjeningen kan repræsenteres ved en ISO sprogkode [ISO '02], og til hver sprogkode et "nøgle/tekst"-par i en sprog-fil, med et par for hvert tekstuel element i brugerfladen.

$$\mathcal{T} = \{ (languagecode_1, languagefile_1), (languagecode_2, languagefile_2), \dots \}$$

6.3.2 TagLib-mærkater

For at stille de tekstuelle komponenter til rådighed i de andre dimensioner benyttes JSP-teknologien "JSP Tag Library". Til repræsentation af sprogkoderne benyttes et `Locale` [J2SE], og til repræsentation af sprogfilerne et `ResourceBundle` [J2SE], med tilhørende property-fil. Property-filerne består netop af "nøgle/værdi"-par, hvor nøglen i denne sammenhæng er referencen til det enkelte tekst element. Værdierne i property-filen er den pågældende tekst, der skal substitueres ind på referencens plads.

Følgende JSP-mærkater bruges til tekstkomponenterne:

- `<text:placetext key="x" />`
Placeres i JSP-filerne som markering af, at værdien af den sprog- og/eller relations-afhængige reference x skal placeres dette sted.
- `<text:changelocale text="y" />`
Placeres i JSP-filerne som markering af, at der på dette sted skal placeres en repræsentation af mulighederne for at skifte sprog. Argumentet y benyttes som nøgle til hvilken tekst eller illustration sprogskiftet skal have.

De to mærkater beskrives i TagLib XML-beskrivelsen `text.tld` ved:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- $Id: text.tld,v 1.7 2003/02/05 20:18:25 jottosen Exp $ -->
[...]
<tag>
  <name>placetext</name>
```

```

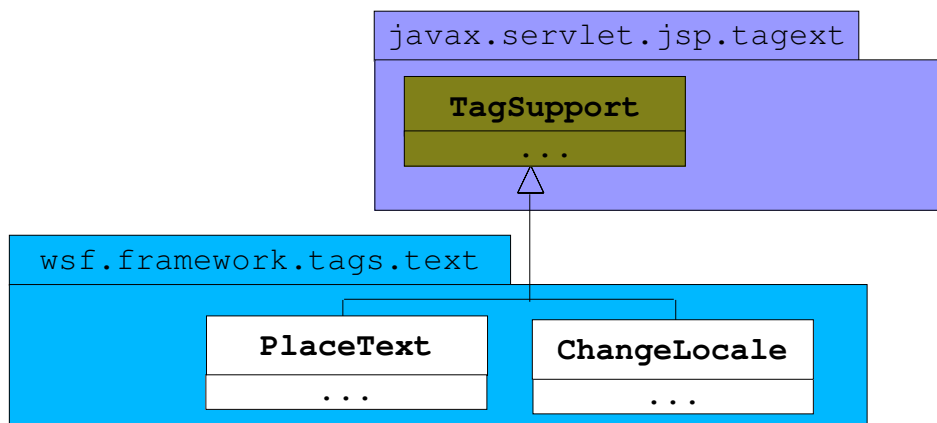
<tag-class>wsf.framework.tags.text.PlaceText</tag-class>
<body-content>empty</body-content>
<attribute>
  <name>key</name>
  <required>true</required>
</attribute>
</tag>

<tag>
  <name>changelocale</name>
  <tag-class>wsf.framework.tags.text.ChangeLocale</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>text</name>
    <required>true</required>
  </attribute>
</tag>

```

Se afsnit 6.3.4 for eksempler på brug af de forskellige mærkater.

6.3.3 Implementation



Figur 26: Klassediagram

- wsf.framework.tags.text.PlaceText
Argumentet til mærkaten (key) benyttes til at slå værdien af argumentet op, og den fundne værdi integreres i brugerfladen:

```

001 package wsf.framework.tags.text;
002 // $Id: PlaceText.java,v 1.2 2003/02/12 19:32:39 jottosen Exp $
003
[...] //imports
008
009 public class PlaceText extends TagSupport {
010
011     private String key = "";
012
013     // Getters & setters
[...]
020     public int doStartTag() throws JspException {
021         return SKIP_BODY;
022     };
023
024     public int doEndTag() throws JspException {
025         // Create RessourceReader and extract message
[...]
029         // Write message to pageContext
[...]
034     };
[...]
040 }

```

- wsf.framework.tags.text.ChangeLocale
Mærkatene henter informationer om de tilgængelige sprog, og for hvert sprog bruges argumentet (text) til at finde teksten på sprogskiftet til sproget. Der udskrives en liste af links til alternative sprog:

```

001 package wsf.framework.tags.text;
002 // $Id: ChangeLocale.java,v 1.3 2003/02/12 19:32:39 jottosen Exp $
003
004 [...] //imports
010
011 public class ChangeLocale extends TagSupport {
012
013     private String text = "";
014
015     // Getters & setters
016
017 [...]
022     public int doStartTag() throws JspException {
023         return SKIP_BODY;
024     };
025
026     public int doEndTag() throws JspException {
027         // Create RessourceReader and extract various info
028
029 [...]
033         // for each possible locale (but not this), make a link
034         Vector v = (Vector) rr.getMapping("locales");
035         for (Enumeration e = v.elements(); e.hasMoreElements();) {
036             String l = (String) e.nextElement();
037             if (!loc.equals(l)) {
038                 // if the l isn't current locale, extract link text
039                 String linktxt = rr.getProperty(text+"."+l);
040                 links = links +
041                     "<a href=\""+linkurl+"?
042                       lang="+l+"\">"+linktxt+"</a><br>";
043             };
044
045         // Write links to pageContext
046
047 [...]
049         return EVAL_PAGE;
050     };
051
052 [...]
056 }

```

Med brug af ovenstående mærkater afhænger den indirekte og direkte tilpasning af brugerfladens tekster udelukkende af repræsentationen i klassen `ResourceReader`. Via `ResourceReader` aflæses den konkrete property-fil, afhængigt af brugerens aktivt valgte sprog. Den komplette kildekode og dokumentation til `ResourceReader`, mærkater, og støtteklasser findes på den vedlagte cd-rom, se kapitel 10.1.

6.3.4 Eksempel

Ovenstående mærkater illustreres kort ved nedenstående eksempel. JSP-filen `demo.jsp` benytter mærkatene `<text:placetext>` (markeret i fed) og property-filen `blue_da.properties`. Yderligere eksempler og detaljer findes i `mdp-webapplikationen`, kapitel 10.2.4.

- `demo.jsp` indeholder tre tekster:

```

<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/text.tld" prefix="text" %>
<HTML><HEAD>
  <!-- $Id: demo.jsp,v 1.8 2003/03/01 20:48:41 jottosen Exp $ -->
  [...]
</HEAD><BODY><div align="left">

<h1><text:placetext key="demo.title" />
  <text:placetext key="demo.flag" />
  <text:placetext key="demo.version" /></h1>

</div></BODY></HTML>

```

- `blue_da.properties` indeholder tekster på dansk:

```
# FRAMEWORK TEXT PROPERTIES danish
# $Id: blue_da.properties,v 1.3 2003/01/29 18:51:21 jottosen Exp $

demo.title      =Velkommen
demo.flag       =
demo.version    =<font color="blue">[Blå]</font>
[...]
```

Når disse filer kombineres, bliver resultatet følgende HTML:

```
<HTML><HEAD>
  <!-- $Id: demo.jsp,v 1.8 2003/03/01 20:48:41 jottosen Exp $ -->
  [...]
</HEAD><BODY><div align="left">

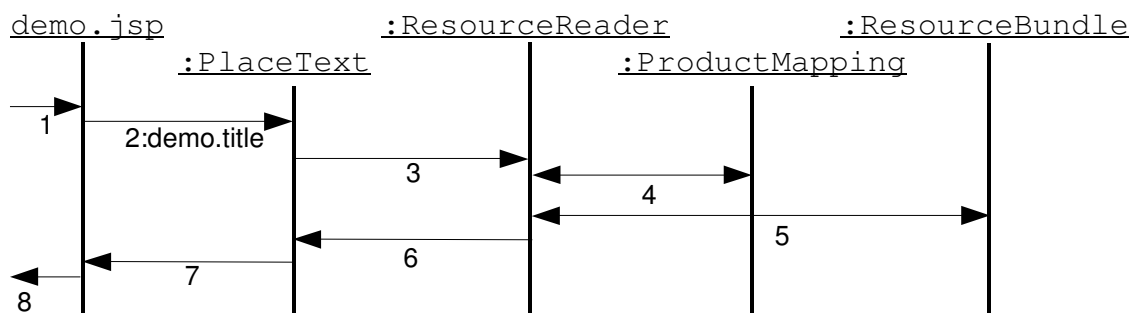
  <h1>Velkommen
    
    <font color="blue">[Blå]</font></h1>

</div></BODY></HTML>
```

6.3.5 Interaktion

Ovenstående eksempel benyttes til illustration af interaktionen:

1. JSP-filen `demo.jsp` aktiveres
2. Mærkaten `<text:placetext key="demo.title">` aktiveres
3. `ResourceReader` kaldes
4. Placeringen af mulige property-filer aflæses i via `ProductMapping`
5. Værdien i den relevante property-fil aflæses via `ResourceBundle`
6. Værdien af `demo.title` returneres
7. Mærkaten (2) afsluttes
8. JSP-filen (1) afsluttes



Figur 27: Interaktionsdiagram

De to andre anvendelser af `<text:placetext>`-mærkaten er tilsvarende, ligeledes er anvendelser af `<text:changelocale>`. Sidstnævnte er en smule mere kompleks på grund af opbygningen af links til mulige sprog.

6.3.6 Alternativer

I stedet for de foreslåede TagLib-mærkater til håndtering af tekstelementerne, er TagLib-mærkaterne i Struts en mulighed. Ved hjælp af en `message-bean` opnås en detaljeret tilgang til property-filerne, som kun kan varieres på baggrund af browserens sprog. Sproget i en selvbetjening baseret på Struts bliver således tilpasset indirekte via browserens sprog og ikke direkte via brugerens eget valg af sprog som i ovenstående løsning. Skal Struts komponenterne håndtere en direkte brugertilpasning af tekster, skal det udvides specifikt til formålet.

Ved teksttunge selvbetjeninger med mange løbende ændringer og tilføjelser i teksterne, bliver håndteringen af property-filer omstændelig. I stedet kan det være mere attraktivt at opbevare teksternes værdier i en database, der kan tilgås af tekstforfattere via et kommercielt artikelsystem eller content management system. Hvis dette er tilfældet, er det i ovenstående løsning begrænset til ændringer i `ResourceReader`-klassen at hente teksternes indhold i databasen.

6.4 Menukomponenter

Den tredje dimension i det brugertilpasset View-lag er opbygningen af en brugertilpasset menu af selvbetjeningsmuligheder. Menuens komponenter kan beskrives ved et navn (forbrug) samt et link til det pågældende komponent (`/vis_forbrug.jsp`).

Præsentationen af den samlede menu skal kunne beskrives uafhængigt af de forskellige grafiske templates og med en mulighed for at angive placering af hvert enkelte menupunkt. Hvilken tekst eller illustration menupunktet skal have håndteres som et tekstuel komponent, da der ofte er en sprogmæssig variation over teksterne og illustrationerne på menupunkterne.

Menuens elementer, mængden O , kan beskrives som en mængde af menupunkter, hvor hvert menupunkt består af et unikt navn og et link til den konkrete forretningsoperation:

$$O = \{ (name_1, link_1), (name_2, link_2), \dots \}$$

6.4.1 Samlet menulayout

Det er umiddelbart nærliggende at placere de enkelte menupunkter og deres indbyrdes strukturering i den grafiske template, men det vil skabe uønsket høj kobling mellem struktureringen af menuerne og det grafiske layout. Det er i stedet tilstrækkeligt, at der i den grafiske template er en måde at angive den layoutmæssige placeringen af menuen som helhed. Layoutet af menuen bør således beskrives i en separat JSP-fil, og det er i

denne fil, at den indbyrdes strukturering af de enkelte menupunkter angives. Layoutet af menuen skal kun beskrives én enkelt gang for hver forretningsmæssige relation uafhængigt af, hvilke menupunkter der aktuelt skal præsenteres for brugeren.

Hvert enkelt menupunkt skal ikke angives eksplicit i layoutet for menuen, men angives ved referencer. På den måde er det op til den bagvedliggende håndtering at identificere, om det pågældende menupunkt skal vises for brugeren. Det giver også den fordel, at der i forbindelse med udarbejdelsen af menuens grafiske layout er overblik over alle menupunkter, uanset hvilken konkret sammenhæng menupunkterne skal bruges i.

6.4.2 TagLib-mærkater

Til opbygningen af brugerfladens menukomponenter udvides dels de tekstuelle TagLib-mærkater og dels de grafiske TagLib-mærkater. Som nævnt skal der i det grafiske layout kunne angives en layoutmæssig placering af menuen som helhed, mens placeringen af hvert enkelt menupunkt kan håndteres som en tekstuel komponent.

Følgende JSP-mærkater bruges til menukomponenterne:

- `<text:menuitem item="x" key="y" />`
Element til placering af de enkelte menupunkter. Argumentet `x` angiver navnet på menupunktet og argumentet `y` angiver en reference til det tekstkomponent, som menupunktet skal benytte. Beskrives i [TagLib XML-beskrivelsen text.tld](#) ved:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- $Id: text.tld,v 1.7 2003/02/05 20:18:25 jottosen Exp $ -->

[...]

<tag>
  <name>menuitem</name>
  <tag-class>wsf.framework.tags.text.MenuItem</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>item</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>key</name>
    <required>true</required>
  </attribute>
</tag>
```

- `<design:placemenu />`
Element til placering af den samlede menu i den grafiske template. Beskrives i [TagLib XML-beskrivelsen design.tld](#) ved:

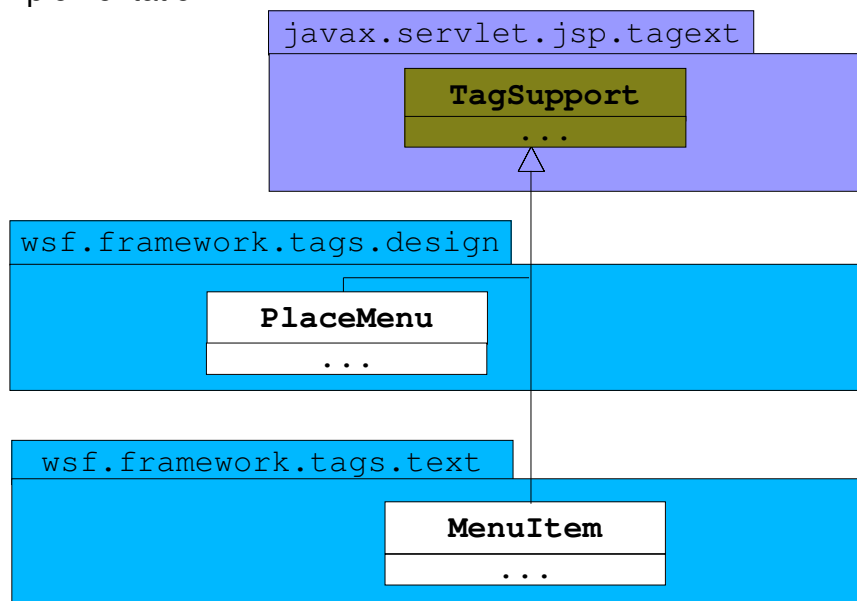
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- $Id: design.tld,v 1.6 2003/02/05 20:18:32 jottosen Exp $ -->

[...]

<tag>
  <name>placemenu</name>
  <tag-class>wsf.framework.tags.design.PlaceMenu</tag-class>
  <body-content>empty</body-content>
</tag>
```

Se afsnit 6.4.4 for eksempler på brug af de forskellige mærkater.

6.4.3 Implementation



Figur 28: Klassediagram

- `wsf.framework.tags.text.MenuItem`
Argumenterne til mærkaten (key og item) benyttes til at slå menupunktet op og finde teksten, der skal integreres i brugerfladen:

```

001 package wsf.framework.tags.text;
002 // $Id: MenuItem.java,v 1.5 2003/02/12 19:32:39 jottosen Exp $
003
[...] //imports
010
011 public class MenuItem extends TagSupport {
012
013     private String item = "";
014     private String key = "";
015
016     // Getters & setters
[...]
029     public int doStartTag() throws JspException {
030         return SKIP_BODY;
031     };
032
033     public int doEndTag() throws JspException {
034         // Create RessourceReader
035         ResourceReader rr = new ResourceReader(pageContext);
036
037         // Get page remembered in Request Scope
[...]
043         // if this page, just the text, else a link from configuration
044         String menuItem = "";
045         if (page.equals(item)) {
046             menuItem = rr.getProperty(key);
047         } else {
048             Hashtable items = (Hashtable)
049                 rr.getMapping("menuitems");
050             String link = (String) items.get(item);
051             String text = rr.getProperty(key);
052             menuItem = "<a href=\"" + link + "\">" + text + "</a>";
053         };
054
055         // include menuItem in pageContext
[...]
059         return EVAL_PAGE;
060     };
[...]
068 }
  
```

- `wsf.framework.tags.design.PlaceMenu`

Find det menulayout, relationen bruger og inkluder i brugerfladen:

```

001 package wsf.framework.tags.design;
002 // $Id: PlaceMenu.java,v 1.4 2003/02/12 19:32:38 jottosen Exp $
003
004 [...] //imports
005
006 public class PlaceMenu extends TagSupport {
007
008     public int doStartTag() throws JspException {
009         return SKIP_BODY;
010     };
011
012     public int doEndTag() throws JspException {
013         // Create RessourceReader and extract menulayout
014         ResourceReader rr = new ResourceReader(pageContext);
015         String menulayout = (String) rr.getMapping("menulayout");
016
017         // include menulayout in pageContext
018         [...]
019         return EVAL_PAGE;
020     };
021
022 [...]
023 }

```

6.4.4 Eksempel

Ovenstående mærkater illustreres kort ved nedenstående eksempel. JSP-filen `demo.jsp` benytter placeringen af menulayoutet. Menulayoutet er beskrevet i `menu.jsp` og indeholder et menupunkt, som trækker på property-filen `blue_da.properties`. Yderligere eksempler og detaljer findes i `mdp-webapplikationen`, se kapitel 10.2.4.

- `demo.jsp`, indeholder en placering af menuen:

```

<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/design.tld" prefix="text" %>
<HTML><HEAD>
  <!-- $Id: demo.jsp,v 1.8 2003/03/01 20:48:41 jottosen Exp $ -->
  [...]
</HEAD><BODY><div align="left">
  <table cellpadding="5">
    <tr><td width="100" class="greybg">
      <design:placemenu />
    </td></tr>
  </table>
</div></BODY></HTML>

```

- `menu.jsp`, indeholder et enkelt menupunkt:

```

<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/text.tld" prefix="text" %>
<!-- $Id: menu.jsp,v 1.2 2003/02/05 20:16:11 jottosen Exp $ -->
<li><text:menuitem item="demo" key="menulayout.demo" />
<br>&nbsp;<br>

```

- `blue_da.properties`, indeholder tekster på dansk:

```

# FRAMEWORK TEXT PROPERTIES danish
# $Id: blue_da.properties,v 1.3 2003/01/29 18:51:21 jottosen Exp $

menulayout.demo      =Prøv demoen
[...]

```

Når disse filer kombineres, bliver resultatet følgende HTML:

```

<HTML><HEAD>
  <!-- $Id: demo.jsp,v 1.8 2003/03/01 20:48:41 jottosen Exp $ -->
  [...]
</HEAD><BODY><div align="left">

```

```

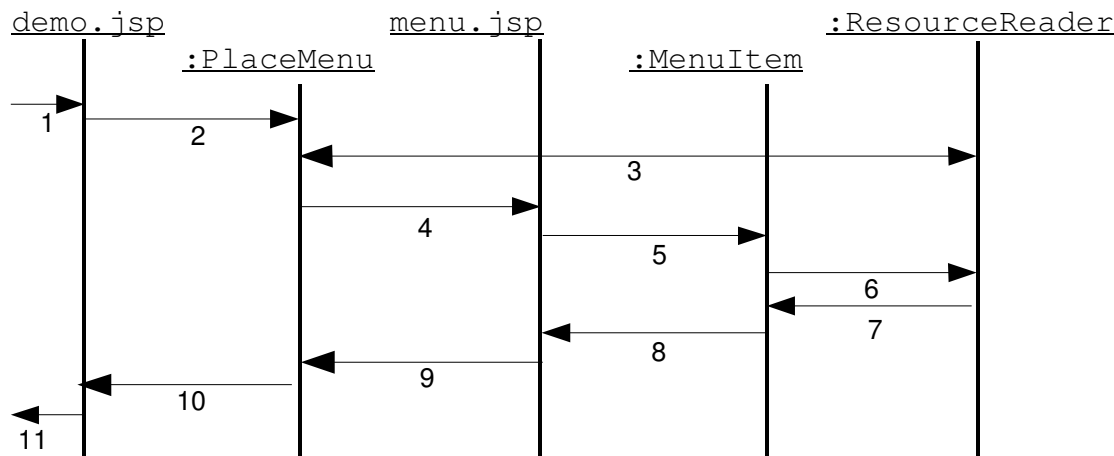
<table cellpadding="5">
  <tr><td width="100" class="greybg">
    <!-- $Id: menu.jsp,v 1.2 2003/02/05 20:16:11 jottosen Exp $ -->
    <li>Prøv demoen
    <br>&nbsp;<br>
    </td></tr>
</table>
</div></BODY></HTML>

```

6.4.5 Interaktion

Ovenstående eksempel benyttes til illustration af interaktionen:

1. JSP-filen `demo.jsp` aktiveres
2. Mærkatens `<design:placemenu />` aktiveres
3. Menulayoutet aflæses via `ResourceReader`
4. JSP-filen `menu.jsp` aktiveres
5. Mærkatens `<text:menuitem ... />` aktiveres
6. `ResourceReader` kaldes
7. Værdien af `menulayout.demo` returneres
8. Mærkatens (5) afsluttes
9. JSP-filen `menu.jsp` afsluttes
10. Mærkatens (2) afsluttes
11. JSP-filen `demo.jsp` afsluttes



Figur 29: Interaktionsdiagram

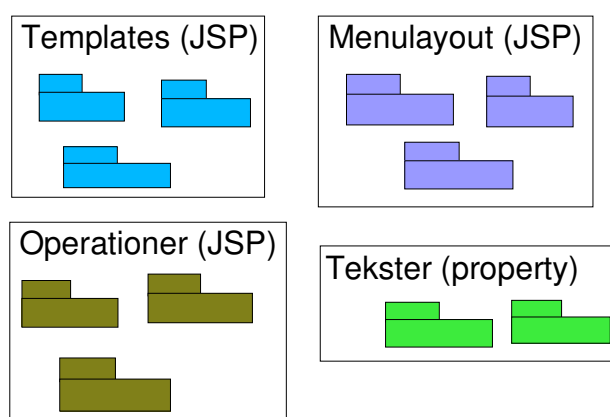
6.4.6 Alternativer

I nogle selvbetjeninger er menuens enkelte punkter så afhængig af detaljer i den forretningsmæssige relation, at der skal opstilles en større matrice til afkodning af relationen og menuelementerne. I det tilfælde kan det være en fordel at gemme denne information i Model-laget og ikke i en konfiguration i webapplikationen, som der er tilfældet i ovenstående `ResourceReader`. Ovenstående mærkater til udvidelse af JSP-filernes funktionalitet er ikke

umiddelbart afhængige af, hvor den bagvedliggende funktionalitet henter informationerne, og de ændringer, der skal til for at hente informationer andetsteds, er begrænset til en ændring i `ResourceReader`.

6.5 Teknisk evaluering

Brugertilpasningen af View-laget er opnået ved at opdele brugerfladen i en række logiske komponenter (operationer, templates og menulayout), som alle implementeres i JSP og af servlet-containeren omdannes til Java-klasser. Tekstkomponenterne i property-filerne er konfigurationsindhold, men skal aflæses af en passende Java-klasse. De forskellige TagLib-mærkater i View-laget kobler de forskellige JSP-filer sammen.



Figur 30: Brugerfladens delkomponenter

6.5.1 Performance

Det er en generel problemstilling i forbindelse med lav kobling, at den opnåede fleksibilitet sker på bekostning af flere funktionskald i afviklingen af applikationen. Selvom flere kald er omkostningsfulde, er problemstillingen vedrørende performance ikke umiddelbart afklaret.

Performanceproblemer ved de mange funktionskald kan blandt andet nedbringes ved fokus på performance i aflæsning af parametriseringen. Performance ved parametriseringsaflæsning er velkendt problem i forbindelse med konfigurationsstyring. Løsningen er ofte at indlæse parametriseringen i hukommelsen ved opstart, hvilket resulterer i en forøgelse af loadtid ved opstart. I ovenstående implementation indlæses parametriseringen i en hashtabel i et **Singleton**-objekt [Gamma '95, s.127].

Performance afhænger udover parametriseringen af servlet-containerens konkrete implementation af kald mellem TagLibs og JSP-filer og af den initiale fortolkning og udførelse af JSP-filerne. I forhold til øget starttid er denne faktor mere væsentlig, men den er afhængig af implementationen af den aktuelle servlet-container.

Ved at fokusere på forbedringer af performance i aflæsningen af parametrisering opnås en begrænset belastning på grund af det øgede antal funktionskald. Generelt kan det desuden konstateres, at de foreslåede JSP-filer vil have et bedre struktureret indhold og være mindre omfangsrige (på grund af den høje samhørighed og lave kobling) i forhold til løsninger, hvor JSP-filerne indholder mere blandet indhold (højere kobling og lav samhørighed).

6.5.2 Vedligeholdelse

For at opnå en tilpasning af brugerfladen opdeles denne i forskellige dimensioner ved hjælp af en række JSP-filer. Antallet af filer og omkostningen ved at vedligeholde disse skal overvejes. Umiddelbart bevirker opdelingen i de forskellige dimensioner og filer, at vedligeholdelsen af de enkelte dimensioner er begrænset til vedligeholdelse af et fåtal af filer uafhængigt af de andre dimensioner.

Hvis det på forhånd er sikkert, at applikationen har et lavt genbrug af grafisk design og kun har ganske få faste tekster, introducerer den beskrevne løsning umiddelbart en række filer for meget. Løsningens fokus er ikke på applikationer med lavt genbrug, men på problematikken ved en høj variation af brugerfladens komponenter, og den beskriver en lavt koblet opdeling af JSP-filerne, som er lineært skalerbar. Antallet af enkeltstående filer er af summen af antallet af forskellige operationer, antallet af forskellige grafiske designs, antallet af forskellige menuer og antallet af mulige sprog.

6.5.3 Anvendelse

For at benytte ovenstående komponenter i en selvbetjening, skal man tilføje de to TagLib XML-beskrivelser, samt tilhørende klassefiler, til webapplikationens filstruktur på servlet-containeren (Se fx. [Bergsten '01]). Udgangspunktet for de tilføjede mærkater er, at brugerfladens elementer dokumenteres i den tilhørende konfiguration. Konfigurationen kan med fordel dokumenteres i forbindelse med det overordnede design og kravsbeskrivelse for selvbetjeningen og ellers justeres løbende gennem udviklingsforløbet. Det kan være en fordel at beskrive selvbetjeningens brugerflade dels ved brugerfladefunktionen, \mathcal{F} , og dels ved de forskellige operationer, templates, menulayout og sprogfiler.

Med udgangspunkt i beskrivelsen af selvbetjeningens kan udviklerne arbejde med operationerne, tekstforfattere skrive sprogfilerne, og brugerfladedesignerne arbejde med de forskellige templates og beskrivelsen af menuens layout. Det er væsentligt, at der er en åben og hyppig kommunikation mellem de involverede parter. Udviklerne og de grafiske designere skal være enige om, hvilke pladser i templateen, de forskellige operationer skal udfylde. Ligeledes skal der være enighed om hvilke nøgler, der bruges i sprogfilerne.

Når ellers dokumentationen og kommunikationen er på plads, kan udviklingsforløbet fortsætte med efterfølgende kvalitetssikring, deployment og afvikling. Hvorledes udviklingsforløbet, kvalitetssikring og deployment foregår, er op til den pågældende udviklingsorganisation. Afvikling af selvbetjeningen overlades til den valgte Java servlet-container.

Selvom løsningen tog udgangspunkt i MVC Model 2-arkitekturen, er der ikke umiddelbart noget i løsningen, der kræver at selvbetjeningen i øvrigt skal bestå af Java Servlets og Enterprise JavaBeans. Om ønsket kan alle operationerne beskrives i JSP og følge MVC Model 1 (Se kapitel 3.2.2).

6.6 Konklusion

I ovenstående løsning stilles TagLib-mærkater til rådighed, som giver en samlet løsning, et letvægtsframework, for, hvordan man opdeler de forskellige dimensioner i brugerfladen i forskellige uafhængige dele. Det brugertilpassede View-lag er beskrevet ved komponenter, der har intern høj samhørighed og indbyrdes lav kobling. Den lave kobling er opnået på bekostning af en stigning i antallet af funktionskald i forbindelse med afviklingen, og performanceproblemer i denne forbindelse kan nedbringes blandt andet med fokus på parameter aflæsningen og valg af servlet-container.

6.6.1 Relaterede design patterns

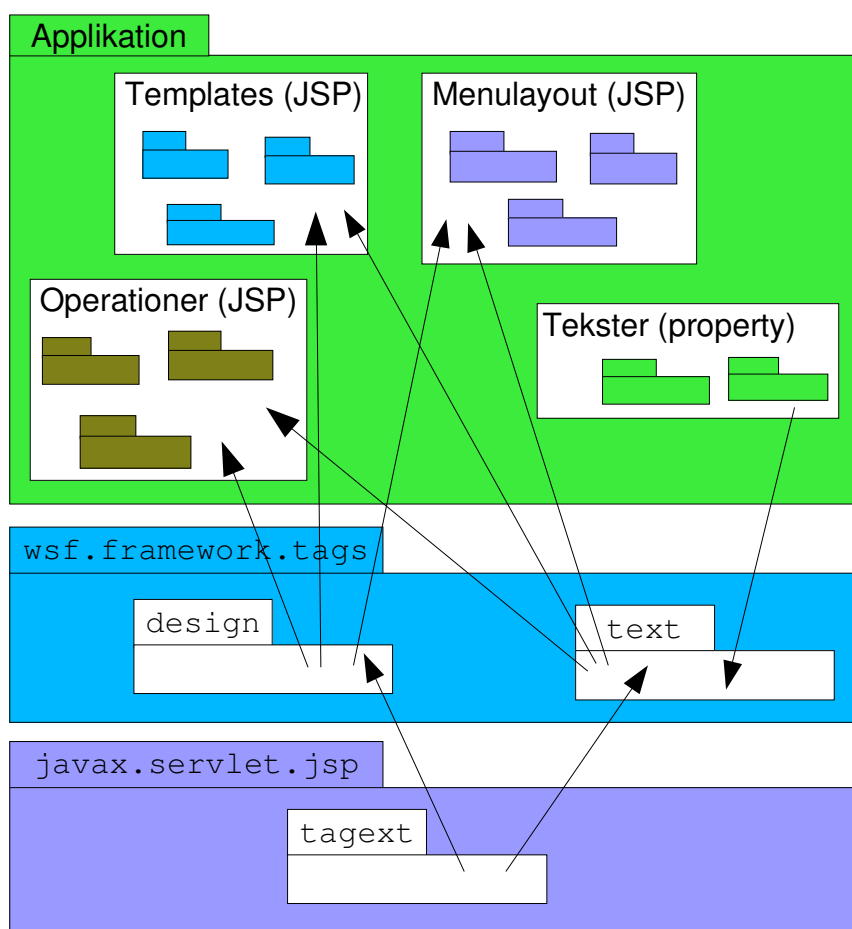
Løsningens design er relateret til følgende design patterns:

- Low coupling: Indbyrdes lav kobling mellem filer og dimensioner.
- High cohesion: Høj intern samhørighed indenfor filer og dimensioner.
- Bridge: Opdeling af anvendelsen i en JSP-fil og den konkrete repræsentation af elementerne i andre filer.
- View Helper [Alur+ '01]: Opdeling af brugerfladen i præsentation og hjælpefunktioner, brug af **Custom Tag Helper Strategy**.
- Composite View: View-laget består af sammensatte elementer, der ændrer sig ofte. Brug af flere atomare subviews som i **JSP View Strategy**.

6.6.2 Evaluering som framework

WS-frameworkets View-lag er ikke et framework i klassisk forstand. Der er hverken klasser at nedarve efter eller et interface at implementere. Alligevel kan løsningen betragtes som et framework, da den indkapsler rutiner til en opdeling af brugerfladen i uafhængige dele og pålægger applikationen en bestemt struktur. View-lagets framework-hot spots er TagLib-mærkaterne, der stiller funktionalitet til rådighed for den konkrete applikation. Hvorledes resultatet af funktionen frembringes er ikke tilgængeligt for applikationen, men kan konfigureres i parametriseringen.

View-laget har fuld kontrol over udførelsen af applikationen og stiller krav til hvordan applikationen i JSP-filerne skal opbygges, og det uden at indeholde skabeloner i forhold til applikationen. Den konkrete applikation bruger i stedet frameworket via komposition (har/består af-relation) og parametrisering. Det brugertilpassede View-lag er dermed karakteriseret ved flere blackboxframework-egenskaber, blandt andet ved at have fuld kontrol over udførelsen og benytter kompositions-relationer [Roberts '96] [Gurph+ '01].



Figur 31: Brugen af View-laget i WS-frameworket

6.6.3 Fremtidigt forskning

J2EE-teknologien Java Server Faces [JSF] fokuserer på at tilbyde funktionelle komponenter til brugerfladen, således at brugerfladen kan sammensættes af en struktureret opbygning af brugerfladens Javascripts, formularvalidering, aflæsninger af data fra Enterprise JavaBeans m.v. Denne tilgang til opbygning af brugerfladen genfindes i flere frameworks inden for webapplikationer. En opdeling af View-laget i uafhængige dele i forbindelse med JSF kunne undersøges nærmere.

Opdelingen af en webbaserede brugerflade i flere uafhængige elementer er ikke begrænset til selvbetjening, men kan genbruges i andre webbaserede sammenhænge, hvor der er behov for en indbyrdes lav kobling mellem brugerfladens forskellige komponenter. Yderligere forskning kunne afdække en lav koblet opdeling af brugerfladen i [JWIG], som umiddelbart er baseret på, at applikationskoden og XML-præsentation beskrives i samme JSP-fil.

Løsningen er umiddelbart udarbejdet med J2EE for øje, yderligere forskningsresultater kunne afprøve om principperne kan genbruges indenfor andre programmeringsplatforme. Det kunne blandt andet afprøves i forbindelse med Microsoft .NET-teknologien WebControls [WebControls], da den netop definerer TagLib-lignende mærkater, der stiller information, funktionalitet og databehandling til rådighed i brugerfladen.

7 Tilstandsbaseret Controller-lag

Kravene til Controller-laget i WS-frameworket er, at det skal udgøre et afgrænset framework, der tilbyder en struktureret afvikling af selvbetjeningsoperationer. Da selvbetjeninger håndterer økonomiske, personlige og forretningsmæssige informationer er det væsentligt, at hver eneste selvbetjeningsoperation udføres med udgangspunkt i en trinvis evaluering af forretningsreglerne for operationen.

Den omhyggelige afvikling af operationerne kan baseres på et tilstandsbegreb i Controller-laget, som konkrete selvbetjeninger kan tage udgangspunkt i ved hjælp af applikationsorienterede framework-hot spots. Controller-laget kan dermed indgå som et supplerende abstraktionslag ovenpå en række basisframeworks, og kan medvirke til en velafgrænset udvikling og vedligeholdelse af selvbetjeningsoperationer.

De følgende kapitler omhandler analysen og formaliseringen af en den trinvis evaluering af forretningsregler, der skal foretages i Controller-laget. Designet af Controller-lagets komponenter beskrives direkte ud fra analysen og en vurdering af hvilke hot spots, der skal stilles til rådighed. Dernæst følger en gennemgang af hvordan de enkelte delkomponenter implementeres og der afsluttes med to eksempler på applikationer, der benytter det tilstandsbaserede Controller-lag som applikationsframework.

7.1 Analyse

WS-frameworket er baseret på MVC Model 2-arkitekturen og består af View-laget, Controller-laget samt en abstraktion over Model-laget. Det er i Controller-laget, at abstraktionen af Model-laget etableres, og View-lagets repræsentation af den webbaserede brugerflade aktiveres. I følge MVC Model 2-arkitekturen bør Controller-laget hverken indeholde brugerflade, præsentation eller direkte manipulation af forretningsmæssige data. Controller-laget er i stedet ansvarlig for at behandle de forskellige operationer og aktivere den relevante tilbagemelding i View-laget på baggrund af informationer i Model-laget.

De forretningsmæssige regler og forudsætninger skal undersøges, selvom den ønskede operation blot er at få fremvist en forside, en oversigt eller lignende brugerfladeorienteret handling. Disse operationer har også forskellige forretningsregler tilknyttet, der bestemmer hvilke forudsætninger, der gælder for at forsiden eller oversigten kan benyttes.

7.1.1 Beskrivelse

En selvbetjening kan beskrives som en passwordbeskyttet samling af muligheder for, at brugeren kan vedligeholde egenskaber ved sin personlige relation med den konkrete organisationen. Derfor skal det første der undersøges i afviklingen af en forespørgelse være, om brugeren er logget ind og dermed er blevet entydigt identificeret. Er brugeren allerede identificeret kan afviklingen af forespørgelsen fortsætte. Men er brugeren ikke logged ind skal afviklingen af operationen afbrydes og brugeren skal præsenteres for en side i View-laget, hvor der kan logges ind.

Er brugeren korrekt identificeret, skal det efterfølgende undersøges, om brugeren har de forretningsmæssige rettigheder til at aktivere den pågældende forespørgelse. En forudsætning for, at forespørgelsen kan gennemføres kan eksempelvis være, at brugeren skal acceptere nogle supplerende vilkår første gang forespørgelsen gennemføres. Hvis brugerens rettigheder, ifølge Model-laget, er i orden, kan afviklingen fortsætte, ellers må afviklingen afbrydes og sende brugeren til en side i View-laget, der indeholder en relevant besked. En relevant besked kan f.eks. være en godkendelse af supplerende vilkår eller en mulighed for erhvervelse af rette forretningsmæssige rettigheder til at få forespørgelsen udført, eksempelvis ved tilkøb eller anden aktivering af den ønskede rettighed.

En del selvbetjeningsoperationer er opbygget som i en guide: en fast række trin, der skal gennemføres i en bestemt række for at fungere. f.eks. består onlineregistreringen hos TDC af følgende faste trin:

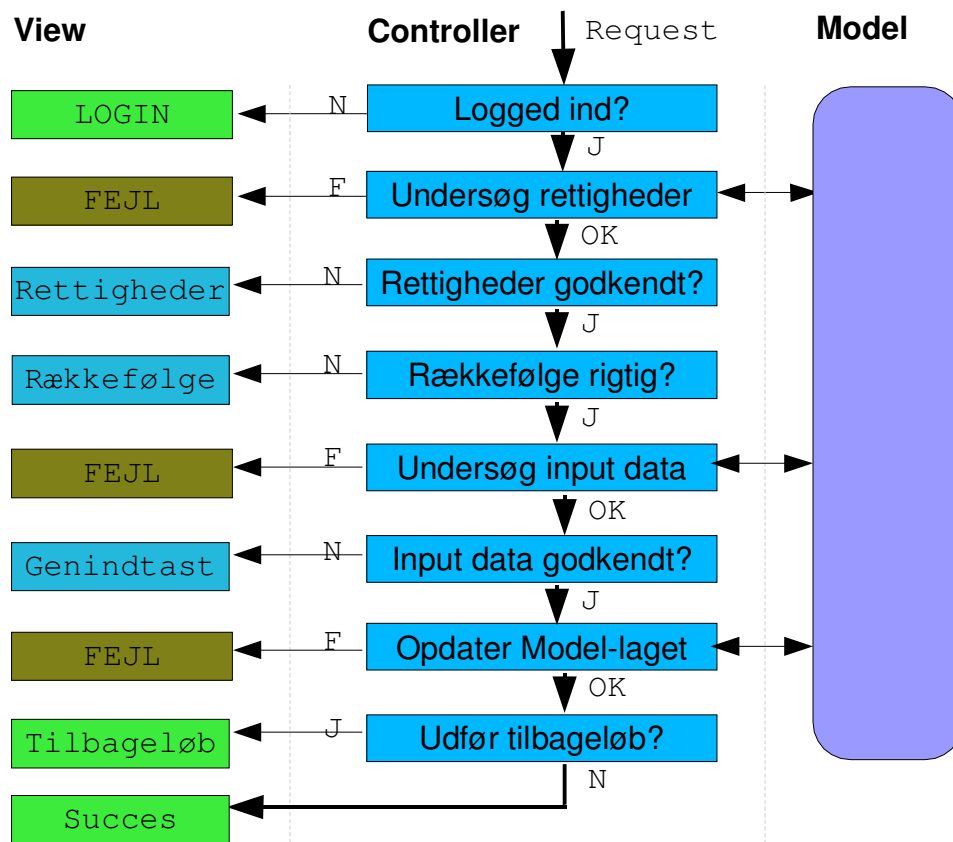
- Produktbeskrivelse
- Accept af kundevilkår
- Indtastning af personlige oplysninger
- Valg af emailadresse
- Kvitteringsside.

I afviklingen af forespørgelserne i Controller-laget skal der være en mulighed for at evaluere rækkefølgen af forespørgelserne og, hvis ikke rækkefølgen er korrekt, funktionalitet til at sende brugeren til en tidligere forespørgelse.

En sidste del af forudsætningerne for, at forespørgelsen kan gennemføres, er, om de påkrævede felter er udfyldt, og at alle formulardata er i de forventede formater. Selvom basal validering af påkrævede felter og inputformater kan foregå i brugerfladen (som ved Jwig) er det i sidste ende op til Model-laget at afgøre om input data kan accepteres. Input data er ofte tæt koblet til den konkrete forretningsoperation, der ønskes udført, eller er baseret på en beregning eller databaseopslag, som brugerfladen ikke umiddelbart kan gennemføre. Har brugeren glemt et påkrævet felt eller udfyldt et felt med forkerte informationer, skal de ukorrekte informationer rettes.

Hvis ellers brugeren er logget ind og har ret til at indsende forespørgslen, aktiverer forespørgslerne i den rigtige rækkefølge og derudover udfylder felterne korrekt, kan selve den datamæssige aflæsning og opdatering af Model-laget gennemføres, og brugeren kan sendes hen til den næste rette side i View-laget. Det kan hænde, at der opstår fejl i denne datamæssige opdatering af Model-laget, og i de tilfælde skal brugeren sendes til en passende fejlside. I forbindelse med ovenstående aflæsning af brugerens rettigheder og validering af indtastet information kan der, på samme måde, også risikeres fejl i kommunikationen med Model-laget.

I de tre ovenstående tilfælde, hvor brugeren sendes til en anden side uden at den ønskede forespørgelse udføres, skal der være mulighed for, at næste trin, brugeren udfører, er netop den forespørgelse, der oprindeligt blev kaldt. Brugeren ønsker at gennemføre en forespørgelse, men skal eventuelt først forbi en side med supplerende information eller logge ind først. I stedet for at få det normale resultat af eksempelvis at logge ind, skal brugeren sendes direkte hen til den tidligere ønskede og forventede operation. Dette "tilbageløb" er medvirkende til, at der kan refereres til selvbetjeningens enkeltstående forespørgelser fra offentlige sider, da det i forbindelse med hver eneste forespørgelse altid undersøges om nødvendige forudsætninger (identifikation, rettigheder m.v.) er til stede.



Figur 32: Flow for Controller-laget

En nærmere gennemgang af de trin, Controller-laget skal evaluere, viser at der er mulighed for otte forskellige udfald, udover den reelt forventede succesfulde tilbagemelding. Der er mulighed for fejl i kommunikationen med Model-laget tre steder (opslag af rettigheder, validering af input data og dataopdatering) og risiko for, at brugeren skal sendes til en anden side i fem situationer (tilbageløb, ikke logged ind, har ikke rette rettigheder, ikke udført i korrekt rækkefølge og invalide input data). Disse otte forskellige udfald er udtryk for, at der er otte forskellige forudsætninger, der skal undersøges, før en opdatering af Model-laget giver mening og den succesfulde tilbagemelding i View-laget kan aktiveres.

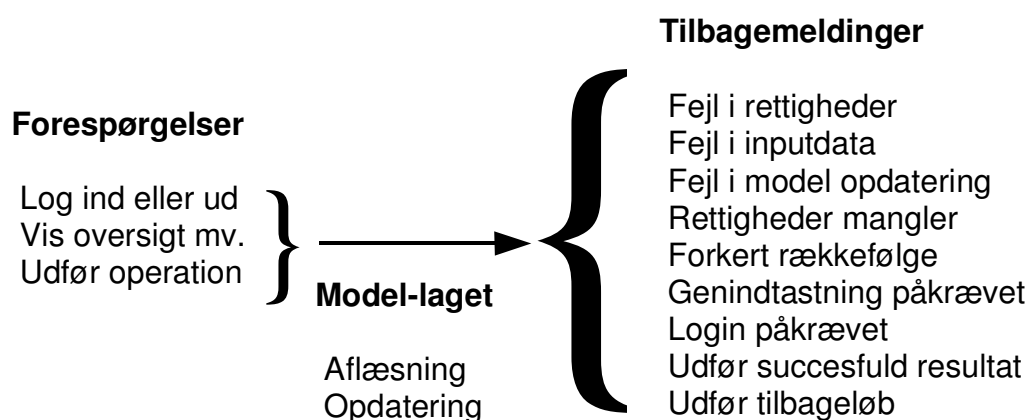
7.1.2 Forespørgelserne

Forespørgelserne svarer til de handlinger, som brugeren rekvirerer: "se forbrug", "udfør overførelse", "vis en oversigt" osv. Specielt er også operationerne "log ind" og "log ud" forespørgelser. Tilbage meldingen på "log ind" er forsiden og tilbage meldingen på "log ud" at komme ud af selvbetjeningen. Forespørgelserne kan listes som disse tre typer:

- Log ind eller ud
- Vis forside, oversigt eller lignende
- Udfør en konkret operation

Resultatet af en forespørgelse er en af følgende tilbage meldinger:

- Fejlet kommunikation ved aflæsning af rettigheder.
- Fejlet kommunikation ved validering af input data.
- Fejlet kommunikation ved dataopdatering.
- Rettigheder mangler.
- Rækkefølge forkert.
- Genindtastning påkrævet.
- Login påkrævet.
- Vis forespørgelsens succesfulde udfald.
- Foretag et tilbageløb.



Figur 33: Forespørgelser og tilbage meldinger

Således kan afviklingen af en selvbetjeningsoperation beskrives som bestående af en forespørgelse, en aflæsning/opdatering af Model-laget og en tilbagemelding. Forespørgelsen udgøres af en af tre muligheder og tilbagemeldingen af en af ni forskellige muligheder.

7.1.3 Formalisering

Analysen viser, at de enkelte operationer i en selvbetjening kan beskrives som bestående af en forespørgelse, en aflæsning/opdatering af Model-laget og en aktivering af en tilbagemelding i View-laget. Mængden \mathcal{R} af mulige forespørgelser (requests) kan beskrives ved følgende definition:

$$\mathcal{R} = \{ \text{forside}, \text{log ind}, \text{log ud}, \text{operation}_1, \text{operation}_2, \dots \}.$$

Hver eneste forespørgelse $r \in \mathcal{R}$ kan resultere i en række forskellige tilbagemeldinger i View-laget alt efter aflæsningen og opdateringen af Model-laget. Model-laget benævnes i det følgende ved \mathcal{M} , mens \mathcal{V} betegner mængden af alle tilbagemeldinger i View-laget. Elementerne i \mathcal{R} er mere end blot funktioner af \mathcal{M} og \mathcal{V} , men er rettere transitionsfunktioner, da de involverede argumenter ændres i forbindelse med beregningen.

Afviklingen af en forespørgelse afhænger udover \mathcal{M} og \mathcal{V} af seneste tilbagemelding ($v \in \mathcal{V}$), om der skal udføres et tilbageløb og om brugeren er identificeret. Om brugeren er identificeret beskrives ved $i \in I = \{ "true", "false" \}$. Et tilbageløb w beskrives ved enten det elementet i \mathcal{V} , der skal tilbageløbes til eller ved at være nul:

$$w \in \mathcal{W} = \{ w \mid w = 0 \vee w \in \mathcal{V} \} = \mathcal{V} \cup \{ 0 \}$$

En forespørgelse $r \in \mathcal{R}$ kan samlet beskrives som:

$$r: \mathcal{M} \times \mathcal{V} \times I \times \mathcal{W} \rightarrow \mathcal{M} \times \mathcal{V} \times I \times \mathcal{W}.$$

Formel beskrivelse af r :

$$\begin{aligned} r(\mathcal{M}, v, i, w) = & \text{ If } w \neq 0 & (\mathcal{M}, w, i, 0) \\ & \text{Elsif } (i = \text{false}) \wedge (r \neq \text{Login}) & (\mathcal{M}, \text{Login}, i, v) \\ & \text{Elsif } \text{Rights}(\mathcal{M}, r) = \text{Fail} & (\mathcal{M}, \text{Fail}, i, v) \\ & \text{Elsif } \text{Rights}(\mathcal{M}, r) = \text{false} & (\mathcal{M}, \text{Missing}(r), i, v) \\ & \text{Elsif } v \neq \text{Required}(r, v) & (\mathcal{M}, \text{Required}(r, v), i, v) \\ & \text{Elsif } \text{Input}(\mathcal{M}, r) = \text{Fail} & (\mathcal{M}, \text{Fail}, i, v) \\ & \text{Elsif } \text{Input}(\mathcal{M}, r) = \text{false} & (\mathcal{M}, v, i, w) \\ & \text{Elsif } \text{Update}(\mathcal{M}, r) = \text{Fail} & (\mathcal{M}, \text{Fail}, i, v) \\ & \text{Else } \mathcal{M}' = \text{Update}(\mathcal{M}, r) & \\ & \quad i' = \text{DoLogin}(r, i) & (\mathcal{M}', \text{Success}(r), i', w) \end{aligned}$$

Funktioner på \mathcal{M} :

$Rights(\mathcal{M}, r) =$	<i>If user has right to activate r</i>	<i>true</i>
	<i>Else</i>	<i>false</i>
	<i>Exception</i>	<i>Fail</i>

$Input(\mathcal{M}, r) =$	<i>If input data of r is correct</i>	<i>true</i>
	<i>Else</i>	<i>false</i>
	<i>Exception</i>	<i>Fail</i>

$Update(\mathcal{M}, r) =$	<i>\mathcal{M}' is the updated Model</i>	<i>\mathcal{M}'</i>
	<i>Exception</i>	<i>Fail</i>

Funktioner på \mathcal{R} :

$Required(r, v) =$	<i>If $\exists v' \in \mathcal{V}$: r requires v'</i>	<i>v'</i>
	<i>Else</i>	<i>v</i>

$Missing(r) =$	<i>$v \in \mathcal{V}$: v gives access to r</i>	<i>v'</i>
----------------	--	------------------------

$Succes(r) =$	<i>$v \in \mathcal{V}$: v is the succes of r</i>	<i>v</i>
---------------	---	-----------------------

Faste værdier i \mathcal{V} :

Login, Fail $\in \mathcal{V}$

Funktioner på I :

$DoLogin(r, i) =$	<i>If $r = login$</i>	<i>true</i>
	<i>Elsif $r = logout$</i>	<i>false</i>
	<i>Else</i>	<i>i</i>

7.2 Design

Designet af WS-frameworkets trinvis afvikling af de forretningsmæssige regler og forudsætninger kan tage direkte udgangspunkt i ovenstående analyse og formelle beskrivelse. Ud fra den formelle beskrivelse kan løsningens komponenter beskrives, og der kan konstrueres en indkapsling af de to basisframework J2EE og Struts.

7.2.1 Komponenter

Controller-lagets forespørgelser kan beskrives som transitionsfunktioner, dels på Model-laget og dels på tre statusværdier (seneste forespørgelse, log ind og tilbageløb). Controller-lagets adgang til at opdatere og aflæse Model-laget, er blevet beskrevet i tre funktioner (*Rights*, *Input*, *Update*), som skal konkretiseres i forbindelse med hver forespørgelse og som indgår i en fast tilstandsbaseret evaluering, hvor trinene i evalueringen består af:

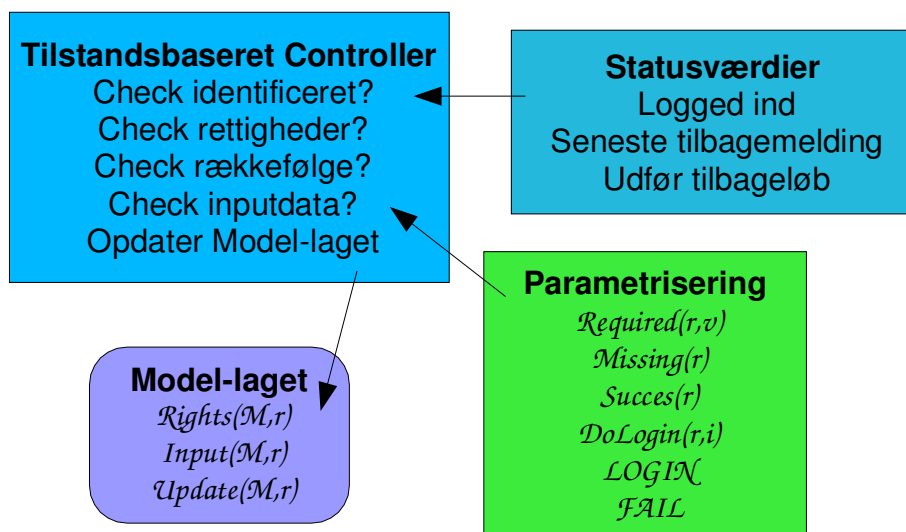
- Er brugeren identificeret?
- Har brugeren rette rettigheder?
- Er rækkefølgen af forespørgelserne korrekt?
- Indeholder input data korrekte data?
- Kan Model-laget opdateres?
- Skal der udføres et tilbageløb?

Ud over de tre funktioner på Model-laget beskriver formaliseringen af forespørgelserne, at hver forespørgelse er afhængig af fire faste funktioner. Det unikke i hver forespørgelse er beskrevet ved de tre støttefunktioner (*Required*, *Missing*, *Success*) på \mathcal{R} . Ud fra disse tre faste funktioner kan det for hver forespørgelse beskrives, hvilken tilbagemelding i View-laget der:

- er påkrævet før gennemførelse af forespørgelsen
- indeholder besked om manglende rettigheder for forespørgelsen
- beskriver det succesfulde resultat af forespørgelsen

Disse tre faste funktioner kan, sammen med de faste tilbagemeldinger i View-laget (*Login*, *Fail*) og den trivielle aflæsning (*DoLogin*), beskrives i en parametrisering til selvbetjeningen. I parametriseringen beskrives hver forespørgelse ved hvilken tilbagemelding, der eventuelt er påkrævet, hvilken tilbagemelding der beskriver manglende brugerrettigheder, og hvilken tilbagemelding der beskriver det succesfulde resultat.

Ved at ændre i parametriseringen er det fleksibelt at vedligeholde forespørgelsernes regler og forudsætninger. Afhængigheden af Model-laget kan også nemt vedligeholdes, da Model-laget har fået en klar afgrænsning uafhængigt af den øvrige opbygning og evaluering af forespørgelserne. Udvikleren af selvbetjeningen kan således fokusere på at udvikle og vedligeholde funktionerne på Model-laget, mens der via WS-frameworket er understøttelse af tilbagemeldinger i form af tilbageløb, check aflogin, genindtastning ved inputfejl og fejl i kommunikationen med Model-laget.



Figur 34: Komponenter

Controller-laget består af en evaluering af en række deltrin og skal have adgang til statusværdierne, fast adgang til parametrisering af dels de faste tilbagemeldinger og dels de faste funktioner. Kommunikationen med Model-laget er blevet afgrænset til et fåtal af funktioner. Se figur 34.

7.2.2 Indkapsling

Før forespørgelserne kommer så langt, at de kan evalueres på baggrund af formaliseringen, er forespørgelsen initieret i et af de to basisframeworks J2EE eller Struts. Der skal designes en indpakning og abstraktion af de to basisframeworks, så de har mulighed for at delegere evalueringen videre til Controller-laget i WS-frameworket.

Struts og J2EE har, som beskrevet i kapitel 3, hver sine whitebox-hot spot, som Controller-laget kan baseres på ved hjælp af designet beskrevet i kapitel 5. Ved brug af de to patterns, Template Method og Adapter, opbygges en løsning, hvor J2EE's og Struts' whitebox-framework indkapsles i hver sin `Wrapper`-klasse, og som efterfølgende aktiverer en fast `Engine`-klasse indeholdende dels en håndtering af frameworkets funktionalitet `hook_generic()` og dels nye framework-hot spots `hook_specific()`.

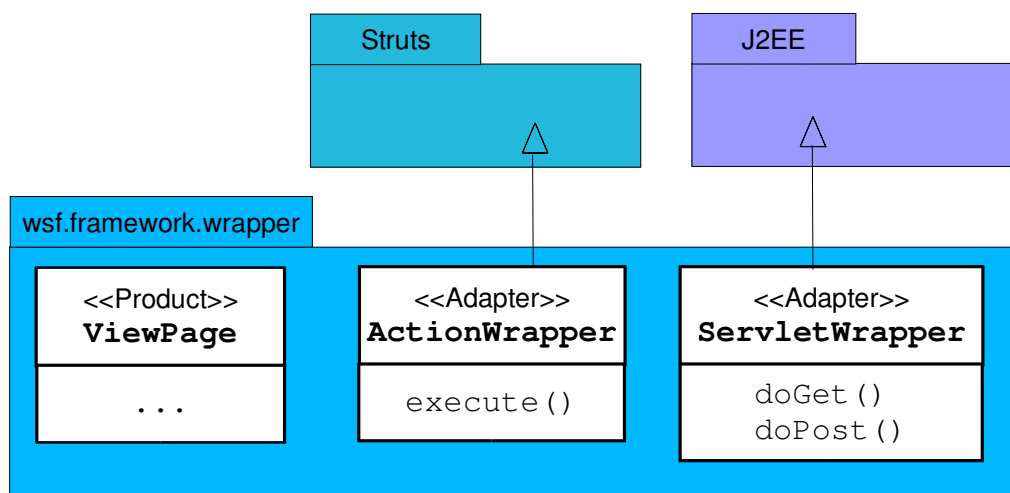
Som nævnt i konklusionen for kapitel 5, skjuler denne indkapsling mange nyttige faciliteter i Struts. Hver forretningsoperation implementeret med Struts' hot spots skal i stedet kunne benytte WS-frameworkets Controller-lag som evalueringsfacilitet betinget af, at frameworkets kommunikation med Model-laget overholdes - med andre ord som blackbox-framework.

7.3 Implementation

Implementationen af Controller-laget kan foretages ud fra ovenstående analyse, formelle beskrivelse og designovervejelser. Løsningen består af en indpakning af to basisframeworks, beskrivelse af Controller-lagets nye applikations-orienterede hot spots og en tilstandsmaskine med eksternt definerede tilstande og parametrisering. Den komplette kildekode og dokumentation til klasserne samt supplerende støtteklasser forefindes på den vedlagte cd-rom, se kapitel 10.1.

7.3.1 Indpakning

Indpakningen af whitebox-hot spot i J2EE udføres som beskrevet i kapitel 5. Som argument (Part) til evalueringen i frameworket benyttes en repræsentation af inputværdier i en Map-struktur [J2SE]. Som resultat af afviklingen (Product) benyttes klassen `ViewPage`, som repræsentation af en JSP-side i View-laget.



Figur 35: Klassediagram

- `wsf.framework.wrapper.ViewPage`
Indholder navnet og/eller stien til JSP-siden i View-laget. Et `ViewPage`-objekt kan sammenlignes med andre `ViewPage`-objekter:

```

001 package wsf.framework.wrapper;
002 // $Id: ViewPage.java,v 1.5 2003/02/17 23:02:45 jottosen Exp $
003
004 public class ViewPage {
005
006     private String name;
007     private String path;
008
009     // Getters and setters
010     [...]
022     // equals
023     public boolean equals(ViewPage vp) {
024         return ((name.equals(vp.getName())) &&
025                (path.equals(vp.getPath())));
026     };
027
028     // Constructors
029     public ViewPage (String n, String p) {

```

```

029         setPath(p);
030         setName(n);
031     };
[... ]
038 };

```

- `wsf.framework.wrapper.ServletWrapper`

Er Wrapper-klassen for J2EE's whitebox-hot spot:

```

001 package wsf.framework.wrapper;
002 // $Id: ServletWrapper.java,v 1.4 2003/02/15 17:31:42 jottosen Exp $
003
[... ] //imports
012
013 public class ServletWrapper extends HttpServlet {
014
015     Engine engine;
016
017     public void doGet(HttpServletRequest req, HttpServletResponse res)
018         throws ServletException, IOException {
019         doPost(req, res);
020     };
021
022     public void doPost(HttpServletRequest req, HttpServletResponse res)
023         throws ServletException, IOException {
024
025         // Get the next page from the engine
026         engine = new Engine(req.getSession());
027         ViewPage vp = engine.execute(req.getPathInfo(),
028             req.getParameterMap());
029
030         // Dispatch control to the next page
031         RequestDispatcher disp = req.getRequestDispatcher(vp.getPath());
032         disp.forward(req, res);
033     };
034 };

```

- `wsf.framework.wrapper.ActionWrapper`

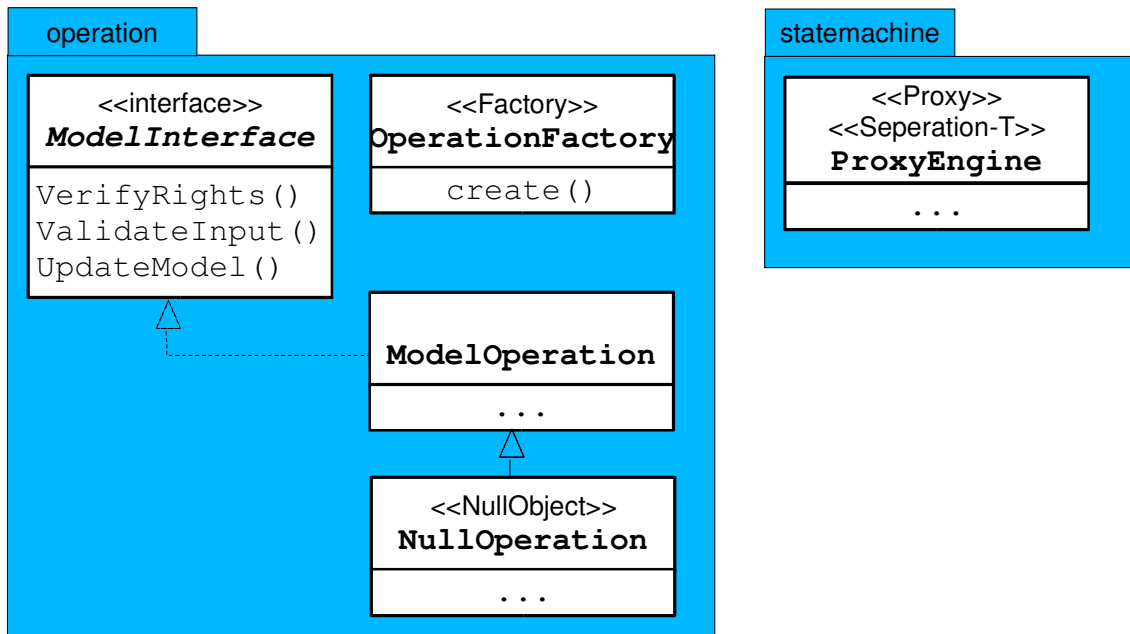
Supplerende Wrapper-klassen for Struts's whitebox-hot spot, konstruert på samme vis som ServletWrapper.

7.3.2 Hot spots

Analysens funksjoner på Model-laget beskrives ved et interface, `ModelInterface`, som benyttes som utgangspunkt for Controller-lagets følgende blackbox- og whitebox-hot spots:

- `ModelOperation`, er et whitebox-hot spot, der implementerer interfacet `ModelInterface`.
- `EngineProxy`, er et blackbox-hot spot, som er basert på en **Proxy** [Gamma+ '95] i forhold til `Engine`-klassen i tilstandsmaskinen (se neste afsnit). `EngineProxy` og `Engine` er begge baserte på at behandle objekter, der overholder `ModelInterface`-interfacet.

En `FactoryMethod` for `ModelInterface`-objekter beskrives i en separat støtteklasse `OperationFactory`, mens støtteklassen `NullOperation`, er et **NullObject** [Grand '98] basert på `ModelOperation`.



Figur 36: Klassediagram

- wsf.framework.operation.ModelInterface

Er interface for beskrivelse af kommunikationen med Model-laget:

```

001 package wsf.framework.operation;
002 // $Id: ModelInterface.java,v 1.3 2003/03/04 19:11:02 jottosen Exp $
003
004 [...] //imports
005
006 public interface ModelInterface {
007
008     public String getName();
009     public void setName(String n);
010
011     public boolean VerifyRights(Map param) throws Exception;
012
013     public boolean ValidateInput(Map param) throws Exception;
014
015     public void UpdateModel(Map param) throws Exception;
016
017 };
  
```

- wsf.framework.operation.ModelOperation

Whitebox-hot spot, der implementerer interfacet:

```

001 package wsf.framework.operation;
002 // $Id: ModelOperation.java,v 1.7 2003/03/04 19:11:02 jottosen Exp $
003
004 [...] //imports
005
006 public abstract class ModelOperation implements ModelInterface {
007
008     private String name;
009
010     // Getters & Setters
011     public final String getName() { return name; };
012     public final void setName(String n) { name = n; };
013
014     // Abstract model operations
015     public abstract boolean VerifyRights(Map param) throws Exception;
016
017     public abstract boolean ValidateInput(Map param) throws Exception;
018
019     public abstract void UpdateModel(Map param) throws Exception;
020
021 };
  
```

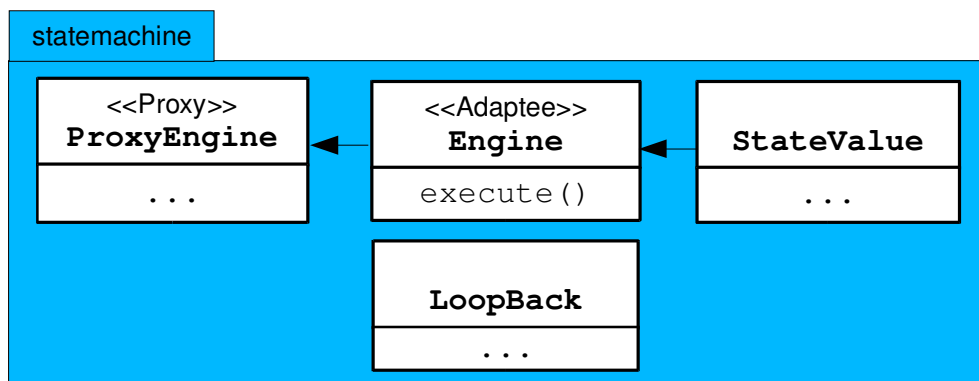
- wsf.framework.statemachine.ProxyEngine
Blackbox-hot spot baseret på ModelInterface, er Proxy for Engine:

```

001 package wsf.framework.statemachine;
002 // $Id: ProxyEngine.java,v 1.1 2003/03/03 21:33:14 jottosen Exp $
003
004 [...] //imports
010
011 public class ProxyEngine {
012
013     Engine e;
014
015     public ViewPage execute(ModelInterface mo, Map parameters) {
016         return e.execute(mo,parameters);
017     };
018
019     public ProxyEngine(HttpSession ses) {
020         e = new Engine(ses);
021     };
022
023 }
    
```

7.3.3 Tilstandsmaskine

Tilstandsmaskinen i Controller-laget indeholder mekanismerne til tilstandsbaseret evaluering af de forretningsmæssige regler og forudsætninger. Tilstanden i maskinen, Engine, beregnes ud fra en tilstandsværdi, StateValue. Om der skal foretages et tilbageløb vurderes til sidst i evalueringen, og i så fald delegeres kontrollen til en LoopBack-klasse, som aktiverer en rekursiv evaluering af tilstanden.



Figur 37: Klassediagram

- wsf.framework.statemachine.Engine
Indeholder hook-metoden til den generelle funktionalitet:

```

001 package wsf.framework.statemachine;
002 // $Id: Engine.java,v 1.13 2003/03/05 15:46:51 jottosen Exp $
003
004 [...] //imports
013
014 public class Engine {
015
016     private StateValue state;
017     private HttpSession session;
018     private ArrayList steps;
019
020     private SubStep createSubStep(Object c, Map m) {
021     [...]}
031     };
032
033     // Wrapper hook
034     public ViewPage execute(String path, Map param) {
035         // Create factory, and create mo using factory method
036     [...]}
    
```

```

039         // Use execute(ModelInterface)
040         return execute(mo,param);
041     };
042
043     // Proxy hook
044     protected ViewPage execute(ModelInterface mo, Map param){
045         // Test validity of mo
046         BufferedReader rr = new BufferedReader();
047     [...]}
048
049         // Find steps in the ResourceReader
050         Iterator iter = rr.getSubSteps().iterator();
051
052         // Use iterator to traverse steps, evaluating each step
053         while (!state.isChanged() && (iter.hasNext())) {
054             SubStep step = createSubStep(iter.next(),param);
055             state = step.evaluate(mo,state);
056         };
057
058         // Take LoopBack into consideration
059         String lb = state.getLoopBack();
060         if ((lb != null) && (!lb.equals(""))){
061             // Create LoopBack handler & return result
062             LoopBack handler = new LoopBack(session);
063             state = handler.evaluate(state,param);
064         };
065
066         // Store state in session and return the resulting page
067         state.saveStateValue(session);
068         return state.getPage();
069     };
070
071     // Constructor
072     public Engine( HttpSession ses) {
073     [...]}
074
075     };
076
077     };
078
079     };

```

- wsf.framework.statemachine.StateValue

Repræsentation af tilstanden i tilstandsmaskinen:

```

001 package wsf.framework.statemachine;
002 // $Id: StateValue.java,v 1.2 2003/02/17 23:02:44 jottosen Exp $
003
004 [...] //imports
005
006 public class StateValue {
007
008     private ViewPage page;
009     private String loopback;
010     private boolean loggedin;
011     private boolean ischanged;
012
013     // Getters and setters
014 [...]}
015
016     public void setStateValue(ViewPage p, String b, boolean l) {
017         page = p; loopback = b; loggedin = l; ischanged = true;
018     };
019
020     protected void saveStateValue(HttpSession ses) {
021         // Get Session attribute key from ResourceReader
022         BufferedReader rr = new BufferedReader();
023         String key = rr.getKey("stateinfo");
024
025         // Save this StateValue in session
026         ses.setAttribute(key,this);
027     };
028
029     // Constructors
030     public StateValue(HttpSession ses) {
031         // Get Session attribute key from ResourceReader
032     [...]}
033
034         // Try to read StateValue from Session
035     [...]}
036
037         // If previous act was to logout, reset to LOGIN
038     [...]}
039
040         // Assign extracted data to attributes
041         loopback = si.getLoopBack();
042         loggedin = si.getLoggedIn();
043         ischanged = false;
044     };
045
046     };
047
048     };
049
050     };

```

```

061     private StateValue() {
[...]
067     };
068
069 }

```

- wsf.framework.statemachine.LoopBack

Udførelse af tilbageløb i en rekursiv evaluering:

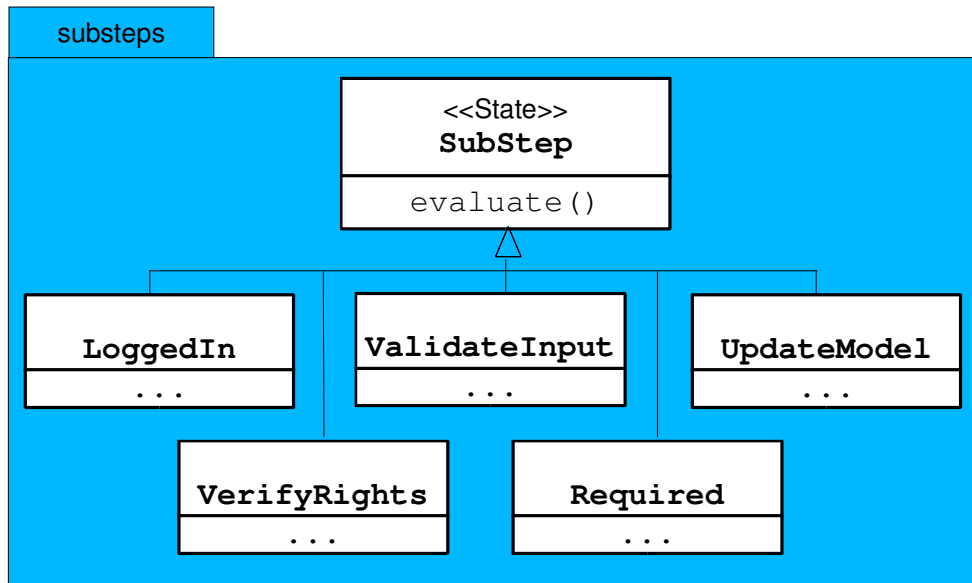
```

001 package wsf.framework.statemachine;
002 // $Id: LoopBack.java,v 1.3 2003/03/05 15:48:47 jottosen Exp $
003
[...] //imports
009
010 class LoopBack {
011
012     HttpSession ses;
013     ResourceReader rr;
014
015     protected StateValue evaluate(StateValue state, Map param) {
016         String prev = state.getLoopBack();
017         ViewPage init = state.getPage();
018         ViewPage login = rr.getPage("LOGIN");
019         boolean useloop;
020         String path = "";
021
022         // If LOGIN, use environment /login and LOGIN
023         // else use the loopback and page given
024         if (init.equals(login)) {
[...]
035         };
036
037         // Recursive evaluation under new environment
038         state.setStateValue(init,"",state.getLoggedIn());
039         state.saveStateValue(ses);
040         Engine newloop = new Engine(ses);
041         ViewPage loopresult = newloop.execute(path, param);
042
043         // Recursive result is not applicable if LOGIN
044         // else the initial environment applies
045         if (useloop) {
046             state.setStateValue(loopresult,"",state.getLoggedIn());
047         } else {
048             state.setStateValue(init,prev,state.getLoggedIn());
049         };
050         return state;
051     };
052
053     protected Map getParam(String p) {
054         // Try reading the map from session
[...]
068     };
069
070     protected void setParam(String p, Map m) {
071         // Save this StateValue in session
[...]
075     };
076
077     // Constructor
078     protected LoopBack(HttpSession s) {
[...]
081     };
082
083 };

```

7.3.4 Tilstande

I tilstandsmaskinen drives den trinvis evaluering ved hjælp af en **Iterator** [Gamma+ '95][J2SE], over enkelte deltrin baseret på superklassen `SubStep`. Hvert deltrin er, efter **State** [Gamma+ '95], ansvarlig for evalueringen af hvert sit trin, baseret på viden om den konkrete forespørgelse og tilstandsværdi. Det er tilstandsmaskinen, `Engine`, der opretter de enkelte deltrin ved `FactoryMethod`-metoden `CreateSubStep()` og kender næste trin. Der er initielt oprettet fem deltrin med mulighed for yderligere konfiguration via parametriseringen.



Figur 38: Klassediagram

- wsf.framework.substep.SubStep
Indeholder superklassen for deltrin i tilstandsmaskinen.

```

001 package wsf.framework.substeps;
002 // $Id: SubStep.java,v 1.3 2003/03/03 21:37:38 jottosen Exp $
003
004 [...] //imports
005
006 public class SubStep {
007     protected ResourceReader resources;
008     protected Map param;
009     // Setters & getters
010     [...]
011     public StateValue evaluate(ModelInterface mo, StateValue state) {
012         return state;
013     };
014     // Constructor
015     public SubStep() {
016         resources = new ResourceReader();
017     };
018 };
  
```

- wsf.framework.substep.LoggedIn
Evaluering af om brugeren er logged ind:

```

001 package wsf.framework.substeps;
002 // $Id: LoggedIn.java,v 1.3 2003/02/19 19:09:55 jottosen Exp $
003
004 [...] //imports
005
006 public class LoggedIn extends SubStep {
007     public StateValue evaluate(ModelInterface mo, StateValue state) {
008         // IF (i="false") || (r!="LOGIN"): r(M,v,i,w)=<M,LOGIN,i,v>
009         if (isLogin(mo.getName())) { return state; };
010         if (!state.getLoggedIn()) {
011             ViewPage login = resources.getPage("LOGIN");
012             state.setStateValue(login,mo.getName(),state.getLoggedIn());
013         };
014         return state;
015     };
016 };
017 [...]
018 };
  
```

- wsf.framework.substep.VerifyRights

Om brugeren har rettigheder ud fra Model-lagets VerifyRights().

```

001 package wsf.framework.substeps;
002 // $Id: VerifyRights.java,v 1.4 2003/03/03 21:37:38 jottosen Exp $
003
004 [...] //imports
005
006 public class VerifyRights extends SubStep {
007
008     public StateValue evaluate(ModelInterface mo, StateValue state) {
009         boolean rightsok = false;
010
011         // IF Rights(M,r)="FAIL": r(M,v,i,w)=<M,FAIL,i,v>
012         try {
013             rightsok = mo.VerifyRights(param);
014         } catch (Exception e) {
015             ViewPager fail = resources.getPage("FAIL");
016             state.setStateValue(fail,state.getLoopBack(),
017                               state.getLoggedIn());
018
019             return state;
020         };
021
022         // IF Rights(M,r)="false": r(M,v,i,w)=<M,Missing(r),i,v>
023         if (!(rightsok)) {
024             Descriptor desc = resources.getOperation(mo.getName());
025             String missing = desc.getMissing();
026             ViewPager succes = resources.getPage(desc.getSuccess());
027             state.setStateValue(succes,missing,state.getLoggedIn());
028         };
029         return state;
030     };
031 };
032
033 };

```

- wsf.framework.substep.ValidateInput

Tilsvarende VerifyRights, blot baseret på mo.ValidateInput().

- wsf.framework.substep.Required

Om forespørgelsen har rette foregående trin:

```

001 package wsf.framework.substeps;
002 // $Id: Required.java,v 1.3 2003/02/19 19:09:55 jottosen Exp $
003
004 [...] //imports
005
006 public class Required extends SubStep {
007
008     public StateValue evaluate(ModelInterface mo, StateValue state) {
009         String req = resources.getOperation(mo.getName()).getRequired();
010         if (req != null) {
011             String key = resources.getOperation(req).getSuccess();
012             ViewPager required = resources.getPage(key);
013             ViewPager page = state.getPage();
014
015             // IF v!=Required(r,v): r(M,v,i,w)=<M,Required(r,v),i,v>
016             if (!page.equals(required)) {
017                 state.setStateValue(required,req,state.getLoggedIn());
018             };
019         };
020         return state;
021     };
022 };
023
024 };
025 };

```

- wsf.framework.substep.UpdateModel

Udfør forespørgelsen baseret på Model-lagets UpdateModel():

```

001 package wsf.framework.substeps;
002 // $Id: UpdateModel.java,v 1.5 2003/03/04 19:15:27 jottosen Exp $
003
004 [...] //imports
005
006 public class UpdateModel extends SubStep {
007
008     public StateValue evaluate(ModelInterface mo, StateValue state) {

```

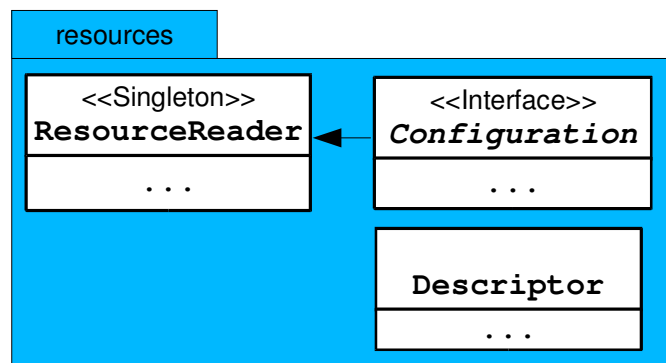
```

011 // IF Update(M,r)="FAIL": r(M,v,i,w)=<M,FAIL,i,v>
012
013 try {
014     mo.UpdateModel(param);
015 } catch (Exception e) {
016     ViewPage fail = resources.getPage("FAIL");
017     state.setStateValue(fail, state.getLoopBack(),
018                         state.getLoggedIn());
019     return state;
020 };
021 // ELSE M'=Update(M,r) l'=DoLogin(r,i):
022 //     r(M,v,i,w)=<M',Success(r),i',w>
023 String key = resources.getOperation(mo.getName()).getSuccess();
024 ViewPage succes = resources.getPage(key);
025 boolean loggedin = DoLogin(mo.getName(), state.getLoggedIn());
026 state.setStateValue(succes, state.getLoopBack(), loggedin);
027 return state;
028 };
029
030 private boolean DoLogin(String n, boolean b) {
031     Boolean o = resources.getOperation(n).getChangeLoggedIn();
032     if (o == null) { return b; }
033     else { return o.booleanValue(); }
034 };
035
036
037 };

```

7.3.5 Parametrisering

Parametriseringen af Controller-laget tager udgangspunkt i en fælles `ResourceReader`, som henter informationer i et Singleton-dataobjekt [Gamma+ '95], som overholder et givet `Configuration`-interface. Egenskaberne ved Controller-lagets forespørgelser implementeres ved en `Descriptor`-klasse.



Figur 39: Klassediagram

- `wsf.framework.resources.ResourceReader`
Aflæsning af konfigurationen fra en Singleton.

```

001 package wsf.framework.resources;
002 // $Id: ResourceReader.java,v 1.7 2003/03/02 17:11:18 jottosen Exp $
003
004 [...] //imports
008
009 public class ResourceReader {
010
011     private static Configuration conf = null;
012
013     // Getters
014     public String getKey(String key) {
015         return conf.getKey(key);
016     };

```

```

017
018     public ViewPage getPage(String key) {
019         return new ViewPage(key, conf.getPage(key));
020     };
021
022     public Descriptor getOperation(String key) {
023         Hashtable op = conf.getOperation(key);
024         return new Descriptor(op);
025     };
026
027     public ArrayList getSubSteps() {
028         return conf.getSubSteps();
029     };
030
031     // Constructor
032     public ResourceReader() {
033         if (conf == null) {
034             conf = (Configuration) new Config();
035         };
036     };
037
038 };

```

(Constructor'en burde have været synchronized)

- wsf.framework.resources.Configuration

Interface til beskrivelse af aflæsningen af parametriseringen:

```

001 package wsf.framework.resources;
002 // $Id: Configuration.java,v 1.11 2003/03/02 17:10:20 jottosen Exp $
003
004 [...] //imports
005
006 public interface Configuration {
007     public String getKey(String key);
008
009     public String getPage(String key);
010
011     public Hashtable getOperation(String key);
012
013     public ArrayList getSubSteps();
014
015 };

```

- wsf.framework.resources.Descriptor

Beskrivelse af forespørgelserne i parametriseringen:

```

001 package wsf.framework.resources;
002 // $Id: Descriptor.java,v 1.6 2003/03/02 17:10:53 jottosen Exp $
003
004 [...] //imports
005
006 public class Descriptor {
007     private String fullclass;
008     private String missing;
009     private String succes;
010     private String required;
011     private Boolean changeloggedin;
012
013     // Getters & setters
014
015     [...]
016
017     // Constructor
018     protected Descriptor(Hashtable h) {
019         setFullClass((String) h.get("fullclass"));
020         setSucces((String) h.get("succes"));
021         setMissing((String) h.get("missing"));
022         setRequired((String) h.get("required"));
023         setChangeLoggedIn((String) h.get("changeloggedin"));
024     };
025
026 };

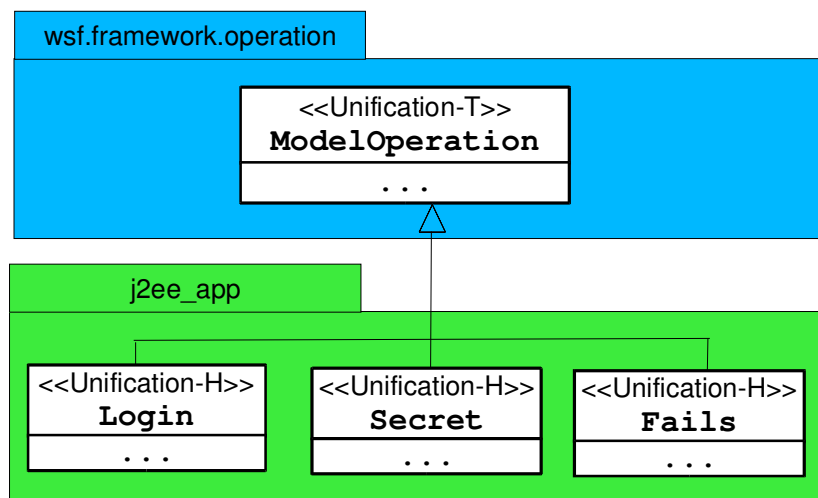
```

7.4 Eksempel

Som eksempel på anvendelse af Controller-lagets forskellige framework-hot spot og abstraktion ovenpå de to basisframeworks J2EE og Struts, implementeres to demonstrations applikationer. Yderligere eksempler og detaljer findes i **sbc-webapplikationen**, kapitel 10.2.3.

7.4.1 J2EE-applikation

Applikationen tager udgangspunkt i Controller-lagets whitebox-hot spot `ModelOperation`, som specialiseres i klasserne: `Login`, `Secret` og `Secret`.



Figur 40: Klassediagram

- `wsf.framework.j2ee_app.Fails`

Altid fejl i opdateringen af Model-laget (brug af `UpdateModel()`):

```

001 package wsf.j2ee_app;
002 // $Id: Fails.java,v 1.2 2003/03/03 21:37:39 jottosen Exp $
003
[...] //imports
006
007 public class Fails extends ModelOperation {
008
009     public boolean VerifyRights(Map param) throws Exception {
010         return true;
011     };
012
013     public boolean ValidateInput(Map param) throws Exception {
014         return true;
015     };
016
017     public void UpdateModel(Map param) throws Exception {
018         throw new Exception();
019     };
020
021 };
  
```

- `wsf.framework.j2ee_app.Login`

Kun brugeren abc kan logge ind (brug af `ValidateInput()`):

```

001 package wsf.j2ee_app;
002 // $Id: Login.java,v 1.2 2003/03/03 21:37:39 jottosen Exp $
003
[...] //imports
006
007 public class Login extends ModelOperation {
  
```

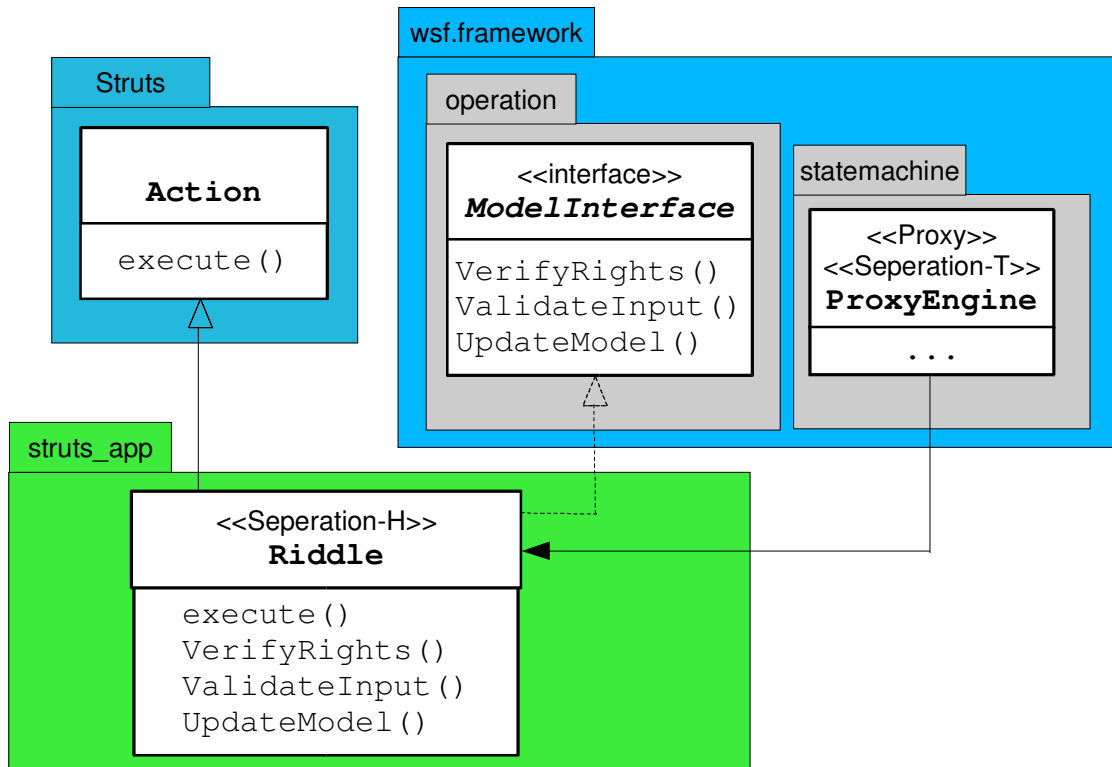
```
008     public boolean VerifyRights(Map param) throws Exception {
009         return true;
010     };
011
012
013     public boolean ValidateInput(Map param) throws Exception {
014         // Get user from parameters
015         String[] user;
016         try {
017             user = (String[]) param.get("user");
018         } catch (Exception e) {
019             return false;
020         };
021         if (user == null) { return false; }
022
023         // only the user "abc" is allowed
024         if (user[0].equals("abc")) {
025             return true;
026         } else {
027             return false;
028         }
029     };
030
031     public void UpdateModel(Map param) throws Exception {
032
033     };
034
035     };
```

- wsf.framework.j2ee_app.Secret
Rettighederne accepteres (VerifyRights()), hvis der er svaret ven:

```
001     package wsf.j2ee_app;
002     // $Id: Secret.java,v 1.2 2003/03/03 21:37:39 jottosen Exp $
003
004     [...] //imports
005
006     public class Secret extends ModelOperation {
007
008         public boolean VerifyRights(Map param) throws Exception {
009             // get phrase form parameters
010             String[] phrase = (String[]) param.get("phrase");
011             if (phrase == null) { throw new Exception(); };
012
013             // only the right phrase shall pass
014             if (phrase[0].equals("ven")) {
015                 return true;
016             } else {
017                 return false;
018             }
019         };
020
021         public boolean ValidateInput(Map param) throws Exception {
022             return true;
023         };
024
025         public void UpdateModel(Map param) throws Exception {
026
027         };
028
029     };
030     };
```

7.4.2 Struts-applikation

Applikationen tager udgangspunkt i Controller-lagets blackbox-hot spot og implementerer i klassen Riddle interfacet ModelInterface samt aktiverer ProxyEngine.



Figur 41: Klasediagram

- wsf.framework.struts_app.Riddle

Input er kun korrekt (ValidateInput()), hvis svaret er ur:

```

001 package wsf.struts_app;
002 // $Id: Riddle.java,v 1.3 2003/03/04 19:14:23 jottosen Exp $
003
004 [...] //imports
005
006 public class Riddle extends Action implements ModelInterface {
007     // From Action
008     public ActionForward execute(ActionMapping mapping,
009                                 ActionForm form,
010                                 HttpServletRequest req,
011                                 HttpServletResponse res)
012         throws Exception {
013
014         // Get the next page from the ProxyEngine
015         ProxyEngine proxy = new ProxyEngine(req.getSession());
016         ViewPage vp = proxy.execute(this, req.getParameterMap());
017
018         // Redirect to the next page
019         return mapping.findForward(vp.getName());
020     };
021
022     // From ModelInterface
023     public String getName() { return "/riddle"; };
024     public void setName(String n) { };
025
026     // From ModelInterface
027     public boolean VerifyRights(Map param) throws Exception {
028         return true;
029     };
030
031     // From ModelInterface
032     public boolean ValidateInput(Map param) throws Exception {
033         // Get answer from parameters
034         [...]
035         // only the answer "ur" is allowed
036         if (answer[0].equals("ur")) {
037             return true;
038         }
039     };
040 }
  
```

```
051         } else {
052             return false;
053         }
054     };
055
056     // From ModelInterface
057     public void UpdateModel(Map param) throws Exception {
058     };
059
060
061 }
```

7.5 Konklusion

Controller-laget beskriver et afgrænset framework, som sikrer en ensartet håndtering af de forretningsmæssige regler og forudsætninger baseret på en beregning af tilstande i et applikationsorienteret framework ovenpå J2EE eller Struts. Ovenstående løsning gør det nemt at vedligeholde og udvikle webbaseret selvbetjening baseret på interfaces og parametriseringer, og det er muligt at specialisere frameworket yderligere ved at nedarve fra støtteklasser og hot spots.

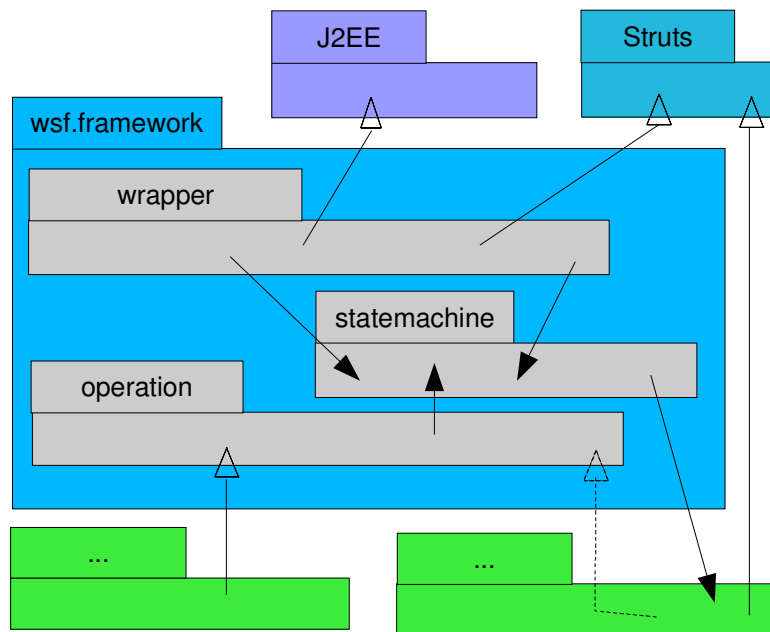
7.5.1 Relateret forskning

Ovenstående implementation af Controller-laget i en MVC Model 2-arkitektur har mange fællestræk med de to J2EE-design patterns **Intercepting Filter** [Alur+ '01] og **Front Controller** [Alur+ '01]. Førstnævnte beskriver en løsning til etablering af pre- og postevaluering af forespørgelser på baggrund af J2EE's filterfaciliteter, og sidstnævnte beskriver en løsning til centraliseret håndtering af de indkomne forespørgelser i lighed med **Command** [Gamma+ '95] [Alur+ '01].

Controller-laget opbevarer tilstanden for brugerens session i form af et `StateValue`-objekt på servlet-containeren. Alternativt kunne dette repræsenteres i form af enten en Enterprise JavaBeans eller ved et XML-dokument. Sidstnævnte fremgangsmåde vil få løsningen til i høj grad at ligne den tilstandsbaserede teknik, der benyttes i **work flow engines**, f.eks. IBM Holosofx [Holosofx] eller Microsoft Workflow Engine [MSWE].

7.5.2 Evaluering som framework

Controller-laget består af et afgrænset framework indeholdende få kernemoduler og er specifikt målrettet håndteringen af forretningsmæssige regler og forudsætninger i en selvbetjening. WS-frameworket tilbyder dels et whitebox-hot spot og dels et blackbox-hot spot til en trinvis evaluering af funktionaliteten. De to typer hot spots forøger frameworkets kompleksitet en anelse, men giver frameworket en øget anvendelighed. Frameworket laver en indkapsling af de to basisframeworks J2EE og Struts.



Figur 42: Lagdeling i Controller-laget

Anbefalingen er, at bruge whitebox-hot spot til J2EE-applikationer og blackbox-hot spots til Struts-baserede webapplikationer. Benyttes omvendt det indeholdte blackbox-hot spot i forbindelse med J2EE kommer forretningsoperationen nedarvet fra `HttpServlet`-klassen, hvilket som nævnt i kapitel 3.2.1 kan give vedligeholdelsesproblemer.

Indkapsles Struts ved brug af det indeholdte whitebox-hot spot, bliver det som nævnt i kapitel 5.4.2, ikke muligt, at anvende faciliteterne i Struts til en udbygning af webapplikationen. En løsning på denne indkapsling af Struts faciliteter er at benytte WS-frameworkets blackbox-hot spot som ovenfor, alternativt at udvide WS-frameworkets kerne med yderligere faciliteter fra Struts, eksempelvis `ActionForm`.

7.5.3 Fremtidigt forskning

Indpakningen af J2EE som basisframework foregår efter princippet beskrevet i kapitel 5, desuden indeholder Controller-laget en delløsning i form af et blackbox-hot spot til Struts-baserede applikationer. Supplerende forskning kunne undersøge om denne delløsning er generelt brugbar ved etableringen af blackbox-hot spots ud fra eksisterende whitebox-hot spots og ved en arkitekturmæssig indpakningen af eksisterende whitebox- og blackbox-hot spots.

8 Konklusion

I dette speciale er begrebet webbaseret selvbetjening blevet beskrevet, afgrænset og analyseret med fokus på de fælles egenskaber på tværs af de forskellige forretningsområder. Webbaseret selvbetjening kan beskrives som en World Wide Web-brugerflade, hvor brugeren har en password-beskyttet brugertilpasset samling af muligheder for at manipulere egenskaberne ved en langvarig relation til en organisation.

Flere organisationers forretning, f.eks. det offentlige serviceniveau, er i høj grad afhængig af brugen af selvbetjening. I de kommende år vil både brugere og organisationer forvente at få stillet yderligere webbaserede løsninger til rådighed og dermed yderligere besparelser og effektiviseringer ved digitalisering og automatisering. Den store efterspørgelse og brug af selvbetjening gør det relevant at undersøge området og diskutere konstruktionen af et framework, som er målrettet udviklingen af selvbetjening.

8.1 WS-frameworket

I dette speciale er der designet og implementeret et framework (WS-frameworket), der løser nogle af de overordnede problemstillinger i udviklingen af selvbetjening. WS-frameworket udgør et arkitekturmæssig abstraktionslag på to eksisterende webapplikation-frameworks (J2EE og Struts). WS-frameworket er applikations-orienteret mod selvbetjening og tilbyder direkte og indirekte brugertilpasset funktionalitet.

8.1.1 Komponenter

WS-frameworket er baseret på MVC Model 2-arkitekturen og består af to velafgrænsede løsninger til henholdsvis View- og Controller-lagene. Det er primært i disse lag, at problemstillingerne omkring et applikationsnært, brugertilpasset og lagdelt framework til selvbetjening håndteres.

I det brugertilpassede View-Lag stilles TagLib-mærkater til rådighed, som giver en samlet løsning, et letvægtsframework, for, hvordan man opdeler de forskellige dimensioner i brugerfladen i forskellige uafhængige dele. Det brugertilpassede View-lag er beskrevet ved komponenter, der har intern høj samhørighed og indbyrdes lav kobling.

Controller-laget indgår direkte som et abstraktionslag ovenpå udvalgte basisframeworks og kan medvirke til en velafgrænset udvikling og vedligeholdelse af selvbetjeningens operationer. Controller-lagets primære opgave er at sikre en ensartet håndtering af de forretningsmæssige regler og forudsætninger baseret på applikationsorienterede hot spots.

8.1.2 Vurdering af Struts og J2EE

Webbaserede selvbetjeninger bliver i fremtiden en væsentlig del af organisationernes forretningsgrundlag, og vil på den baggrund altid være under løbende tilpasning i takt med organisationernes forretningsområder. Webbaserede selvbetjeninger bør baseres på et forretningsmæssigt og applikationsnært framework, som kan variere efter forretningsområdets behov og bør ikke være afhængig af generelle webapplikation-frameworks.

J2EE er en samling af Java-specifikationer og Java-klasser målrettet mod implementation af forretningssystemer, deriblandt webapplikationer. Flere kilder påpeger, at der ikke bør etableres applikationer direkte med udgangspunkt i J2EE, men at det primært er et udgangspunkt for yderligere arkitekturmæssige applikationslag, som f.eks. Struts.

Selvom Struts giver udvikleren nogle gode faciliteter i forbindelse med udvikling af webapplikationer generelt, er der ingen understøttelse af de konkrete trin, der skal implementeres i dets primære whitebox-hot spot. WS-frameworkets Controller-lag tilbyder netop faciliteter til en trinvis og konfigurerbar evaluering af disse trin i forbindelse med webbaseret selvbetjening.

8.2 Fremtidig forskning

Udover at implementere et konkret framework til selvbetjening beskriver specialet nogle generelle resultater, som er uafhængige af det konkrete område. Disse resultater kan undersøges yderligere i forbindelse med fremtidig forskning og konkrete anvendelser.

I forbindelse med designet af View-laget fokuseres på en opdeling af brugerfladen i uafhængige elementer med intern høj samhørighed og indbyrdes lav kobling. Yderligere forskning kunne afdække egenskaberne ved denne opdeling i forbindelse med andre applikationer og frameworks, eksempelvis JMWIG.

WS-frameworket er etableret ovenpå to eksisterende frameworks, og der er designet en løsning til kombination af de indeholdte whitebox-hot spots. Løsningen giver et bud på omdannelsen af et whitebox-hot spot til et blackbox-hot spot, som kunne undersøges yderligere.

Yderligere forskning kunne evaluere resultaterne i forbindelse med en generel lagdeling og indkapsling af eksisterende frameworks, og bl.a. diskutere hvorledes to blackbox-frameworks kombineres og hvordan et blackbox-framework kombineres med et whitebox-framework.

9 Referencer

- [Alur+ '01] Deepak Alur, John Crupi & Dan Malks: "Core J2EE Patterns: Best Practices and Design Strategies", Prentice Hall / Sun Microsystems Press 2001,
<http://java.sun.com/blueprints/corej2eepatterns/index.html>
- [Barracuda] Enhydra Barracuda: "Surveying the Landscape", February 2002, http://barracuda.enhydra.org/cvs_source/Barracuda/docs/landscape.html
- [Bergsten '01] Hans Bergsten: "Java Server Pages", O'Reilly 2001.
- [Bosch+ '99] Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson: "Object-Oriented Frameworks -Problems & Experiences" in "Object-Oriented Application Frameworks" af Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson, John Wiley & Sons 1999.
- [Buschmann+ '96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns", John Wiley & Sons 1996.
- [Brown '02] Simon Brown: "Designing Web Applications and Servlet Patterns" chapter 12 in "Professional Java Servlets 2.3", Wrox 2002.
- [Brown+ '98] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. III McCormick, Thomas J. Mowbray: "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis", John Wiley & Sons 1998.
- [Cann '02] Sandra Cann, Adam Rossi & Peter Pilgrim: "Frameworks for building Component Based Applications" Jcorporate 2002
<http://www.jcorporate.com/econtent/Content.do?state=resource&resource=702>.
- [CGI] Common Gateway Interface, <http://www.w3.org/CGI>.

- [DB] **Danske Bank Netbank:**
<http://www.danskebank.dk/Danskenetbank>.
- [Davis '01] **Malcom Davis: "Struts, an open-source MVC implementation", IBM DeveloperWorks 2001**
<http://www-106.ibm.com/developerworks/library/j-struts/index.html>
- [e-blanket] **e-blanket.dk: "Besparelsespotentialer ved brug af digitale blanketter"** <http://www.e-blanket.dk/wm1949>
- [Eberhard+ '02] **Martin Eberhard, Anders Birch, Marianne Pedersen, Allan Jepsen og Lars Andersen: "Den Digitale Borger 2002", PLS Rambøll Management 2002.**
- [Fayad+ '99] **Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson: "Building Application Frameworks: Object-Oriented Foundations of Framework Design", John Wiley & Sons 1999.**
- [Fontura+ '02] **Marcus Fontura, Wolfgang Pree & Bernhard Rumpe: "The UML Profile for Framework Architectures", Addison-Wesley 2002.**
- [Finansrådet] **Finansrådet:** <http://www.finansraadet.dk>.
- [Gamma+ '95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Design Patterns - Elements of Reuseable Object-Oriented Software", Addison-Wesley 1995.**
- [Grand '98] **Mark Grand: "Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML", John Wiley & Sons 1998.**
- [Gurph+ '01] **J. van Gorp & J. Bosch: "Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines", Software Practice & Experience no 33(3), side 277-300, March 2001.**

-
- [Holosofx] **IBM Holosofx:** <http://www-3.ibm.com/software/integration/holosofx/>
- [J2EE] **Sun Microsystems Java™ 2 Enterprise Edition 1.3 FCS**
<http://java.sun.com/j2ee>.
- [J2SE] **Sun Microsystems Java™ 2 Standard Edition 1.4.1**
<http://java.sun.com/j2se>.
- [JSF] **Sun Microsystems JavaServer™ Faces 1.0 Early Acces Draft** <http://java.sun.com/j2se/javaserverfaces>.
- [JSP] **Sun Microsystems JavaServer™ Pages 1.2**
<http://java.sun.com/products/jsp>.
- [JWIG] **Java™ Extensions for High-Level Web Service Development,** <http://www.brics.dk/JWIG>.
- [Hunter '01] **Jason Hunter: "Java Servlet Programming" 2. edition,** O'Reilly 2001.
- [ISO '02] **International Organization for Standardization: "ISO 639: Codes for the representation of names of languages" 2002.**
- [Knight+ '02] **Alan Knight & Naci Dai: "Objects and the Web",** IEEE Software March/April 2002.
- [Larman '98] **Craig Larman: "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design",** Prentice Hall PTR 1998.
- [Larman+ '02] **Craig Larman & Frederic Bellier: "Web Presentation Frameworks, a SAD example",** IEEE Software March 2002.

- [Malani '02] Prakash Malani: "UI design with Tiles and Struts", **JavaWorld 2002**, <http://www.javaworld.com/javaworld/jw-01-2002/jw-0104-tilestrut.html>
- [Markiewicz+ '01] Marcus Eduardo Markiewicz & Carlos J.P. Lucena: "Object Oriented Framework Development", **ACM Crossroads Student Magazine April 2001**, <http://www.acm.org/crossroads/xrds7-4/frameworks.html>
- [Mattsson+ '99] Michael Mattsson, Jan Bosch & Mohamed E. Fayad: "Framework Integration: Problems, Causes and Solutions" **Communications of the ACM**, vol.42, No. 10 October 1999.
- [Mellor+ '02] W. Mellor & V. Parr: "Government Online a national a national perspective, 2002 annual report", Taylor Nelson Sofres, Januar 2003.
- [WebControls] **Microsoft .NET WebControls**: <http://msdn.microsoft.com/msdnmag/issues/01/09/asp/default.aspx>.
- [MSWE] **Microsoft Workflow Engine**: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wss/wss/_cdo_workflow_overview.asp.
- [Nordea] **Nordea Netbank**: <http://www.nordea.com>.
- [OWASP] **Open Web Application Security Project**: <http://www.owasp.org>.
- [Pedersen '02] Dekan Tom Lautrup-Pedersen: "Oplæg til tværfagligt forskningsprojekt om digital forvaltning", **Det Samfundsvidenskabelige fakultet, Aarhus Universitet 2002**, http://www.samfundsvidenskab.au.dk/nyt/nr32_2002.html#1

-
- [Pree '94] Wolfgang Pree: "Meta Patterns A Means For Capturing the Essentials of Reusable Object-Oriented Design", 8th European Conference on Object-Oriented Programming (ECOOP) 1994.
- [Pree '99] Wolfgang Pree: "Hot-Spot-Driven Development" in "Building application frameworks: object-oriented foundations of framework design", [Fayad+ '99].
- [Pree+ '99] Wolfgang Pree & Kai Koskimies: "Framelets Small and Loosely Coupled Frameworks", ACM Computing Survey Symposium on Application Frameworks, December 1999.
- [Roberts+ '96] Don Roberts and Ralph Johnson: "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", Proceedings of Pattern Languages of Programming (PLoP) 1996.
- [Rüping '00] Andreas Rüping: "Building Frameworks and Applications Simultaneously" Proceedings of Pattern Languages of Programming (PLoP) 2000.
- [Seshadri '99] Govind Seshadri: "Understanding JavaServer Pages Model 2 architecture", JavaWorld 1999
<http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html>
- [Servlet] Sun Microsystems Java™ Servlet 2.3,
<http://java.sun.com/products/servlet>.
- [Singh+ '02] Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team: "Designing Enterprise Applications with the J2EE™ Platform" 2. edition. Addison-Wesley 2002, http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/titlepage.html
- [Struts] Apache Jakarta Struts 1.1 beta 3,
<http://jakarta.apache.org/struts>.
-

- [TagLib] **Sun Microsystems JSP Tag Library,**
<http://java.sun.com/products/jsp/taglibraries.html>.
- [TDC] **TDC's selvbetjening,**
<http://selvbetjening.tdconline.dk>
- [Telestyrelsen] **IT & Telestyrelsen: "Kontrol med forbruget"**
<http://www.itst.dk/wimpdoc.asp?page=tema&objno=95024615>.
- [Tomcat] **Apache Jakarta Tomcat 4.1.18,**
<http://jakarta.apache.org/tomcat>.
- [Turbine] **Apache Jakarta Turbine 2.2,**
<http://jakarta.apache.org/turbine>.
- [VTU '03] **Ministeriet for Videnskab, Teknologi og Udvikling,
Pressemeddelelse vedr. Digital offentlig signatur, 6.
februar 2003, <http://www.vtu.dk>.**
- [Watson+ '01] **James Watson Jr & Joe Fenner: "The High Price of
Self-Service", Information Week, Januar 2001.**
<http://www.informationweek.com/821/rbprovinggrounds.htm>.
- [XML] **eXtensible Markup Language, <http://www.w3.org/XML>.**
- [Yankee '00] **Yankee Group: "The Users Speak: Trends in Call
Centers and Web-Based Customer Care" EuroScope
Communications, Report vol. 5, March 2000.**
- [Zurek '01] **Bob Zurek: "Making Self-Service Pay Off", Forrester
Research, Maj 2001.**

10 Indhold på vedlagt cd-rom

Vedlagt specialet findes en cd-rom indeholdende dels kildekoden og dels dokumentationen af komponenterne i WS-frameworket. Cd-rom'en indeholder desuden de systemer, der er blevet brugt i implementationen. Nedenfor gives en kort oversigt til cd-rom'ens indhold og en vejledning i brugen af de udviklede applikationer. Kontakt mig venligst, hvis cd-rom'en mangler eller mod forventning ikke fungerer.

10.1 Oversigt

Cd-rom'en indeholder fire biblioteker, med følgende indhold:

/dokumenter	Dokumenter og illustrationer.
/javadoc	Dokumentation af kildekoden.
/kildekode	Kildekoden til de tre applikationer.
/systemer	Binære filer til benyttede systemer.

10.1.1 Dokumenter

Vedlagt på cd-rom'en findes specialet i dels Adobe PDF og StarOffice™ samt figurer og illustrationer udarbejdet i StarOffice™.

[CD]/speciale.pdf	Specialet i Adobe PDF
/ dokumenter	Specialets dokumenter
/speciale.sxw	Specialet i StarOffice™
/figurer/*	Figurer og illustrationer

10.1.2 Dokumentation

Biblioteket indeholder dokumentationen af kildekoden til de tre applikationer. Til hver applikation findes dels javadoc dokumentation og dels en oversigt over indeholdte JSP-, XML og TLD-filer.

[CD]/javadoc		
/ greetings	Dokumentation på Greetings-applikationen	
/index.html	Javadoc dokumentation	
/files.html	Oversigt over JSP-, XML og TLD-filer	
/ mdp	Dokumentation på MDP-applikationen	
/index.html	Javadoc dokumentation	
/files.html	Oversigt over JSP-, XML og TLD-filer	
/ sbc	Dokumentation på SBC-applikationen	
/index.html	Javadoc dokumentation	
/files.html	Oversigt over JSP-, XML og TLD-filer	

Dokumentationen omfatter i alt 9389 linier HTML.

10.1.3 Kildekode

Til afprøvning og demonstration af WS-frameworket er der udviklet følgende tre webapplikationer:

- **greetings** baseret på J2EE og Struts til Kapitel 5.
- **mdp** baseret på J2EE til Kapitel 6.
- **sbc** baseret på J2EE og Struts til Kapitel 7.

Hver applikation er struktureret som på servlet-containeren, se beskrivelse i blandt andet [Hunter '01]. Til hver applikation forefindes WAR-filer, Class-filer, shell-scripts til kompilering (`jc.sh`), generering af dokumentation (`doc.sh`) og indplacering af applikationen på servlet-containeren (`deploy.sh`). Det burde være muligt, at indsætte WAR-filerne direkte i en allerede etableret servlet-container.

[CD]/kildekode	
/greetings.war	WAR-fil med greetings-applikationen
/ greetings	Kildekoden til greetings-applikationen
/index.html	Applikationens forside
/struts-jsp/	JSP-filer til Struts eksempler
/WEB-INF	XML- og TLD-filer
/classes/	Java- og Class-filer
/mdp.war	WAR-fil med mdp-applikationen
/ mdp	Kildekoden til mdp-applikationen
/index.html	Applikationens forside
/jsp/	JSP-filer
/WEB-INF	XML- og TLD-filer
/classes	(ingenting)
/wsf/	Java- og Class-filer
/sbc.war	WAR-fil med sbc-applikationen
/ sbc	Kildekoden til sbc-applikationen
/index.html	Applikationens forside
/jsp/	JSP-filer
/WEB-INF	XML- og TLD-filer
/classes	(ingenting)
/wsf/	Java- og Class-filer

Kildekoden omfatter i alt 1237 linier XML, 292 linier HTML, 2255 linier Java, 468 linier JSP og 3448 linier i TLD-filer.

10.1.4 Systemer

Følgende systemer er blevet brugt i udviklingen af WS-frameworket:

- Sun Microsystems Java 2 Standard Edition 1.4.1 [J2SE]
- Sun Microsystems Java 2 Enterprise Edition 1.3 FCS [J2EE]
- Apache Jakarta Tomcat 4.1.18 [Tomcat]
- Apache Jakarta Struts 1.1 beta 3 [Struts]

Alle fire systemer er vedlagt som binære filer beregnet til Linux 2.4.x (som på `horse*.daimi.au.dk`). Hos henholdsvis Sun og Apache kan der hentes tilsvarende systemfiler til Unix og Microsoft Windows 2000/XP.

10.2 Vejledninger

Systemerne installeres i ovennævnte rækkefølge og ifølge de medfølgende beskrivelser. Efterfølgende konfigureres miljø variable som beskrevet nedenfor, afhængigt af det pågældende operativ system:

\$JAVA_HOME	=	/sti_til/Linux-j2sdk1.4.1
\$J2EE_HOME	=	/sti_til/j2sde1.3.1
\$CATALINA_HOME	=	/sti_til/jakarta-tomcat-4.1.18
\$STRUTS_HOME	=	/sti_til/jakarta-struts-1.1-b3
\$JAVADIR	=	JAVA_HOME
\$PATH	=	PATH J2EE_HOME JAVA_HOME

Tomcat servlet-containeren startes ved

```
$CATALINA_HOME/bin/startup.sh
```

og stoppes ved

```
$CATALINA_HOME/bin/shutdown.sh
```

Når Tomcat er startet med standard indstillingerne findes webapplikationerne på følgende url:

```
http://localhost:8080/<navn>/
```

10.2.1 Greetings-applikationen

greetings-webapplikationen indeholder eksempler på simple "Hello World"-applikationer implementeret ved brug af J2EE, Struts og **WebLet**-frameworket omtalt i kapitel 5. Det indeholder fire implementationer (ud fra `HttpServlet`-klassen, ud fra `Struts Action`-klassen, **WebLet** ovenpå J2EE samt **WebLet** ovenpå Struts) af de samme tre små programmer:

- `HelloWorld`, skriver "Hello World" i browseren
- `HelloAgain`, tæller antallet af kald til den i samme session
- `SayHello`, hvor der kan indtastes hvad der skal stå efter "Hello"

10.2.2 MDP-applikationen

mdp-webapplikationen illustrerer brugen af WS-frameworkets brugertilpassede View-lag, som beskrevet i kapitel 6. Webapplikationen benytter sig af de tilbudte JSP mærkater til at illustrere fire forskellige design/produkter:

- Blåt produkt: Egne tekster, menu, og sprogsift
- Rødt produkt: Egne tekster og menu, egen operation
- Grønt produkt: Egne tekster og menu, brug af eget design
- Lilla produkt: Egen opbygning af menu, brug af eget design

10.2.3 SBC-applikationen

sbc-webapplikationen indeholder brug af WS-frameworkets tilstandsbaserede Controller-lag, som beskrevet i kapitel 7. Applikationen indeholder syv små operationer baseret på frameworkets whitebox-hot spot `ModelOperation` og **J2EE**, en enkelt operation baseret på Struts via whitebox-hot spot samt en enkelt operation via frameworkets blackbox-hot spot.

Operationer baseret på Controller-laget og **J2EE** :

- **Login** (`j2ee_app.Login`), kun brugeren `abc` kan logge ind
- **Logout** (`operation.NullOperation`), logger brugeren ud
- **Hemligt** (`j2ee_app.Secret`) fejler, hvis den ikke aktiveres fra menuen og for at få hemmeligheden via menuen skal der svares "ven"
- **Kodeordet** (`operation.NullOperation`), er i parametriseringen konfigureret til at give rettigheder til Hemligt-operationen
- **Fails** (`j2ee_app.Fails`) fejler altid i kommunikationen med modellen
- **Side1** (`operation.NullOperation`) påkrævet for at udføre side 2
- **Side2** (`operation.NullOperation`) kræver side 1

Operation baseret på Controller-lagets whitebox-hot spot og Struts:

- **Login** (`j2ee_app.Login`), kun brugeren `abc` kan logge ind

Operation baseret på Controller-lagets blackbox-hot spot og Struts:

- **Riddle** (`struts_app.Riddle`), svar ur på en gåde