Interaksjon og Søk i Dynamic Presentation Generator

Tobias Rusås Olsen

Institutt for informatikk
Universitetet i Bergen
Norge

Lang Masteroppgave
2010

Forord

Denne masteroppgaven er resultatet av forfatterens masterstudium i programvareutvikling. Studiet er ledet av Institutt for Informatikk som hører til hos det matematisk-naturvitenskapelige faktultet ved Universitetet i Bergen.

Oppgaven handler om hvordan interaksjon og søk kan integreres i et allerede eksisterende innholdshåndteringsystem med en eksisterende unik arkitektur og oppbygning.

Jeg vil gjerne takke Peder Lång Skeidsvoll, Jostein Bjørge og Ole Henning Vårdal, mine medstudenter på prosjektet. Jeg vil også sende en stor takk til Khalid Azim Mughal og Torill Hamre, mine veiledere som har drevet prosjektet framover med deres gode kommentarer og innspill. De har også kommet med tilbakemeldinger på selve oppgaven, noe som har økt dens kvaliteten med flere hakk. Jeg vil også takke Haakon Nilsen, som har lagt utviklingsforholdene tilrette på prosjektet, i tillegg til å komme med gode tekniske kommentarer. En takk går også til Julie Elisabeth Risnes for god korrekturlesing.

Jeg vil også takke andre med tilknytning til JAFU, hovedsaklig Morten Høiland, Øystein Lund Rolland og Peder Refsnes for deres trivelighet. Stor takk går også til min familie, for både støtte og middager underveis.

Bergen, Juni 2010
Tobias Rusås Olsen

Innhold

Figurer

Akronymer

AJAX - Asynchronous JavaScript and XML
API - Application Programming Interface
CMS - Content Management System
DPG - Dynamisk PresentasjonsGenerator
HTML - HyperText Markup Language
HTTP - Hypertext Transfer Protocol
IDE - Integrated Development Environment
IRC - Internet Relay Chat
JAFU - JAva FjernUndervisning
JAR - Java ARchive
JCR - Java Content Repository
JDOM - Java Document Object Model
JPGEN - Java Presentation Generator
JSON - Javascript Object Notation
MIME - Multipurpose Internet Mail Extensions
MVC - Model-View-Controller
PCE - Presentation Content Editor
PM - Presentation Manager
PV - Presentation Viewer
SAX - Simple API for XML
UiB - Universitetet i Bergen
XHTML - Extensible HyperText Markup Language
XML - Extensible Markup Language
XP - eXtreme Programming
XSLT - Extensible Stylesheet Language Transformations

Kapittel 1

Innledning

1.1 Bakgrunn

JAva FjernUndervisning (JAFU) er et prosjekt tilknyttet Institutt for Informatikk. Prosjektet har som mål å kunne tilby nettbasert undervisning for studenter ved Universitetet i Bergen. JAFU har kunne tilby denne type fjernundervisning siden 1999. Fjernundervisningen har i all hovedsak bestått av to kurs, *INF100-F - Grunnkurs i programmering (Programmering I)* og *INF101F - Videregående programmering (Programmering II)*. Disse kursene har tilsvarende kurs på campus, med samme pensum. Kursene gjennomføres ved at foreleser publiserer nyheter, oppgaver og forelesningsnotater på kurssidene, som da blir tilgjengelig for studentene.

For å realisere målene om nettbasert undervisning er det blitt utviklet en rekke verktøy for å publisere og håndtere kursinformasjonen. Det ble etterhvert et behov for mer enn bare statiske nettsider, og dynamiske løsninger ble utviklet.

I 2004 ble Java Presentation Generator (JPGEN) [10], et system for publisering av kurssider, utviklet. Dette ble overtatt av Dynamic Presentation Generator (DPG) [14] i 2005. DPG ble laget med et spesielt mål for øyet, det å kunne gjenbruke struktur til forskjellige sammenhenger. I forbindelse med fjernundervisningen ble det oppdaget at flere kurs hadde samme struktur, men forskjellig innhold. Menyer og sideoppbygning var lik, men innholdet til for eksempel kurset *INF100F* var helt forskjellig fra *INF101F*. DPG tok hensyn til dette ved å benytte *presentasjonsmønstre* [55], en løsning der struktur og innhold kan gjenbrukes i forskjellige sammenhenger.

I 2008 ble den andre versjonen av DPG ferdigstilt, med versjonsnavnet DPG 2.0. Ideen om å bruke presentasjonsmønstre var bevart, men systemet var implementert på nytt der målet var å få en mer modulær utgave med flere delsystem, god testdekning, en trygg databaseløsning og en forbedret utgave av presentasjonsmønsterideen. For å implementere

presentasjonsmønsterideen ble det laget en presentasjonsmønsterspesifikasjon som definerer de forskjellige elementene i et presentasjonsmønster. DPG 2.0 er som nevnt oppdelt i flere delsystem og blant annet er det et delsystem som i kurssammenheng vil brukes av studentene. I DPG 2.0 kan studenter kan lese informasjonen som er lagt ut på nettsiden, og laste ned ressurser som forelesningsnotater og oppgaver.

1.2 Motivasjon

Motivasjonen bak oppgaven er å få involvert brukerne til å ta del i utformingen av innholdet på siden. Brukere skal ikke bare kunne lese av innhold som er lagt ut, men også kunne bidra med eget innhold i form av kommentarer og andre former for tilbakemelding på innholdet.

En annen motivasjonsfaktor er søk. I et innholdshåndteringssystem som DPG 2.0 er er det viktig å kunne finne innholdet på en enkel måte, men DPG 2.0 har ingen søkefunksjonalitet.

Det er også viktig at både DPG og presentasjonsmønsterspesifikasjonen holdes oppdatert, og dersom ønsket funksjonalitet ikke er støttet, må spesifikasjonen utvides.

1.3 Målsetninger

Målet for oppgaven er å gi brukeren flere interaksjonsmuligheter i Dynamic Presentasjon Generator 2.0. For å oppnå dette ble det satt opp følgende delmål:

1. Tilrettelegge arkitekturen i DPG for interaksjon
2. Legge til interaksjonsmuligheter i DPG
3. Evaluere forskjellige søkealternativer
4. Implementere søk i DPG ut fra valgt søketeknologi
5. Øke uttrykkskraften til presentasjonsmønsterspesifikasjonen

Disse målene gjennomgås i kapittel 3: Problembeskrivelse med forslag til fremgangsmåte og detaljer om hvordan målene kan oppnås.

1.4 Teknologier

For utvikling vil det benyttes en rekke teknologier som muliggjør komplettering av målsetninger, blant annet vil det behøves et *Integrert Utviklingsmiljø* (eng. *Integrated Development Environment* - IDE), et versjonskontrollsystem (eng. *Revision Control System*), et system

for *kontinuerlig integrasjon* (eng. *Continuous Integration*) i tillegg til alle eksisterende teknologier som benyttes av DPG 2.0. Disse vil bli forklart etter hvert som de blir aktuell i oppgaven. Forfatteren valgte å benytte *Eclipse* [32] som IDE fordi dette var et velkjent verktøy for undertegnede, *subversion* [25] ble valgt for versjonshåndtering og underveis i prosjektet ble også den kontinuerlige utviklingsserveren *Hudson* [57] lagt til.

1.5 Utviklingsmetode

Systemet vil videreutvikles i et team med fire individer, hver med sitt ansvarsområde, og det vil derfor være klokt å finne en felles utviklingsmetodologi, som vil minske problemene som oppstår når det jobbes på samme prosjekt. En del arbeid er felles, men det vil også være mye individuelt arbeid. Alle på prosjektet har tatt kurs som frontet agil systemutvikling og da spesielt *ekstremprogrammering* (eng. *eXtreme programming* - XP). Det er derfor ønskelig å benytte en del av prinsippene derfra, som refaktorering, test-drevet utvikling, parprogrammering, kollektivt eierskap og lignende.

1.6 Organisering av oppgaven

Kapittel 2: Bakgrunn

I bakgrunnskapittelet vil viktige relaterte system, komponenter og teknologier bli forklart, slik at leseren lettere skal forstå innholdet i resten av oppgaven. Innholdshåndteringssystem, presentasjonsmønstre, presentasjonsmønsterspesifikasjonen og DPG 2.0 blir forklart.

Kapittel 3: Problembeskrivelse

Dette kapittelet beskriver dagens situasjon, hva som er problemet med dagens situasjon, og hvorfor en løsning på problemet er interessant. Mer konkret handler det om å forklare hvorfor det er vanskelig å tilrettelegge for interaksjon, hvilke løsninger som er tilgjengelige, og hvilke tiltak som foreslås for å løse de andre målene i oppgaven.

Kapittel 4: Analyse av pluginarkitektur

I problembeskrivelsen kommer det fram at målene kan oppnås ved å forbedre pluginarkitekturen. Derfor beskriver dette kapittelet den eksisterende arkitekturen. Mangler og begrensninger vil påpekes og det vil komme forslag til forbedringer.

Kapittel 5: Endringer i pluginarkitektur

Kapittel 5 tar forslagene fra kapittel 4 i betrakning, og beskriver løsninger og endringer som er gjort på arkitektur og system.

Kapittel 6: Interaksjonsplugins

Her blir interaksjon forklart, og hvorfor det er hensynsmessig med interaksjon i et innholdshåndteringssystem i dag. For å vise styrken til den nye pluginarkitekturen er det utviklet to interaktive plugins, som blir beskrevet i kapittelet.

Kapittel 7: Ny presentasjonsmønsterspesifikasjon

I dette kapittelet forklares svakhetene til presentasjonsmønsterspesifikasjonen i DPG 2.0. Forbedringer som er gjort blir forklart og på slutten gis en før- og ettersammenligning av spesifikasjonen.

Kapittel 8: Søk

Kapittelet om søk gir en kort introduksjon til søk, hvorfor søk er viktig, og hvordan indeksering, tekstanalyse og søk foregår. I andre halvdel av kapittelet vil forskjellige aktuelle søketeknologier beskrives, og tilslutt vil en søketeknologi ble valgt ut fra gitte kriterier.

Kapittel 9: Integrering av søk i DPG

I dette kapittelet forklares det hvordan søking integreres med DPG, og hvilke vurderinger det er tatt hensyn til for å kunne oppnå søk.

Kapittel 10: Konklusjon, evaluering og videre arbeid

I det siste kapittelet vil oppgaven evalueres i forhold til målene for oppgaven. Det vil også være en kort vurdering av teknologiene som ble benyttet, og forfatterens oppfatning om arbeidsprosessen. Det vil også være en del om videre arbeid, et den som vil være viktig for fremtidige masterstudenter som ønsker å jobbe videre med JAFU-prosjektet.

Kapittel 2

Bakgrunn

2.1 Innholdshåndteringssystem

Et *innholdshåndteringssystem* (eng. *Content Management System*) er et system som håndterer innhold. For å forstår hva dette innebærer må ordet *innhold* defineres. Innhold kan sies å være en enhet av digital informasjon som kan ha flere former. Eksempler på innholdstyper er ren tekst, bilder, grafikk, video og mer [8]. Et innholdshåndteringssystem blir da et system som håndterer disse digitale enhetene.

Et innholdshåndteringssystem er et system som har løsninger for å effektivt håndtere innholdet. Dette betyr at den har løsninger for å skape, endre, fjerne og publisere innhold. Internettbaserte innholdshåndteringssystem er en vanlig type innholdshåndteringssystem der dokumenter kan framvist i en nettleser. I de fleste innholdshåndteringssystem er det viktig at systemet skal kunne brukes av personer uten teknisk bakgrunn.

To populære eksempler på slike innholdshåndteringssystem er Wordpress [79] og Joomla! [48].

2.2 Presentasjonsmønstre

Presentasjonsmønstre er en idé som ble påtenkt i 2003 av Khalid Azim Mughal [55]. JAFU-prosjektet er tett knyttet til fjernundervisningen, og hvert år ble kurssider laget ved at forelesere kopierte et kurs og endret innholdet. Alt kursinnholdet bestod av HTML, så innholdsendringen ble en tungvint prosess. Tanken om presentasjonsmønstre handler om at struktur og innhold er to seperate enheter, og at innholdet skal kunne endres selv om struktur er lik, eller strukturen endres slik at innholdet beholdes.

For å konkretisere denne tanken kan det tas utgangspunkt i en nettside for en fotballklubb. Først lages det en presentasjon med en gitt struktur og innhold. Presentasjonsmønsteret til presentasjonen kan ha en struktur som beskriver en spillerstall, en ligatabell og en nyhetsside. Innholdet i presentasjonen kan da innholde informasjon om for eksempel *Sportklubben Brann*. Ved å beholde strukturen, men endre innhold og design kan for eksempel *Tromsø Idrettslag* benytte seg av den samme strukturen, og da trenger kun innhold og designet å endres.

Det er også mulig å benytte innholdet i flere sammenhenger, slik at en kan omforme strukturen på en side og gjenbruke innholdet.

2.3 Presentasjonsmønsterspesifikasjon

Presentasjonsmønsterspesifikasjonen definerer lovlig syntaks som presentasjonsmønstre kan ha. Dette betyr at den definerer hvilke elementer et presentasjonsmønster består av, hvordan disse elementene er knyttet sammen, og hvilke bestanddeler hvert element kan bestå av. Derfor kan det sies at uttrykkskraften til presentasjonsmønstre avhenger av presentasjonsmønsterspesifikasjonen. Målet for spesifikasjonen er at den skal være komplett og lettfattelig. For hvert presentasjonsmønster kan det opprettes flere instanser, og hver instans kalles en presentasjon.

Presentasjonsmønsterspesifikasjonen har fire hovedbestanddeler.

- *utsnitt* (eng. *view*)
- *side* (eng. *page*)
- *entitet* (eng. *entity*)
- *entitetsinstans* (eng. *entity-instance*)

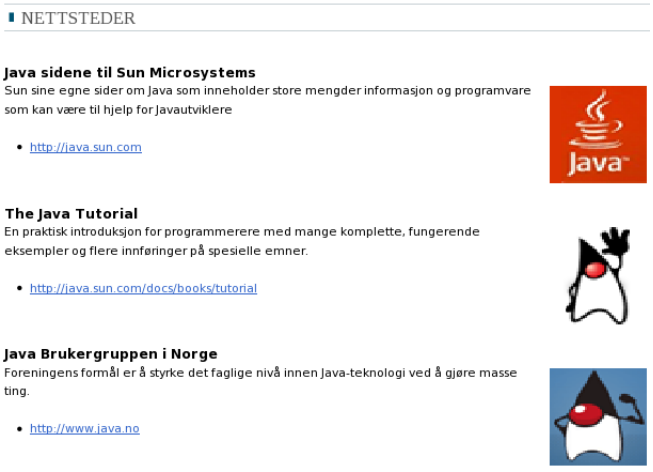
En entitet består av en liste med felter. Et felt kan enten være en innebygget datatype som *string*, *xhtml* eller *date*, en *plugin* eller en liste av entiteter. Datatypene defineres ved å sette attributtet *type* til datatypen som ønskes. Et felt er det minste elementet i presentasjonsmønsterspesifikasjonen, og det er i disse feltene innholdet settes. Figur 2.1 viser hvordan programvare kan representeres i en presentasjonsmønsterspesifikasjon som benytter Extensible Markup Language (XML). Programvare har i eksempelet en tittel, en beskrivelse, et bilde og en lenke.

Listing 2.1: En entitet som beskriver programvare

```
<entity id="programvareEntitet">
  <field type="string">tittel</field>
  <field type="xhtml">beskrivelse</field>
  <field type="image">bilde</field>
  <field type="string">lenke</field>
</entity>
```

En entitetsinstanse er instans av en entitet, og inneholder en referanse til entiteten den instansierer. Det kan lages flere instanser av samme entitet, slik at en kan ha forskjellig innhold til samme entitetstruktur. Et eksempel på dette er dersom du har en spillerstallentitet for en ballklubb. Da kan det instansieres en instans for A-laget og en instans for B-laget.

Et utsnitt består av en referanse til en entitetsinstans og en sti til en transformasjonsfil. Dette gjør at det er mulig å framstille en entitet på forskjellige måter, ved å knytte den opp til forskjellige transformasjonsfiler. Utsnittets oppgave er å definere hvordan entiteten skal vises. I fotballag-eksempelet kan et utsnitt skrive ut lagnavn, beskrivelse og spillere sortert etter fornavn, mens et annet utsnitt bare skriver ut spillerne sortert på etternavn. Figur 2.1 viser et eksempel på et utsnitt der entiteten består av en liste med nettsteder.



Figur 2.1: Utsnitt der entiteten er en liste med nettsteder

En *side* består av en samling med utsnitt. En side er den største bestanddelen i presentasjonsmønsterspesifikasjonen. I deklarasjonen av en side kan det legges til informasjon om hvert utsnitt, for eksempel kan det velges et standardutsnitt for en side eller om et utsnitt alltid skal være synlig på den siden. I presentasjonen vil alle sider ha linker i en hovedmeny. *Side* fungerer på altså som et toppnivå menyvalg på en nettside, og hvert utsnitt er et undervalg fra det valgte menyvalget. I et fotballpresentasjonsmønster kan en side være *om klubben* og et utsnitt kan være *spillerstall* eller *stadioninformasjon*.

2.4 DPG 2.0

2.4.1 Generelt om DPG 2.0

DPG 2.0 er et innholdshåndteringssystem som benytter seg av presentasjonsmønstre. Den er utviklet ved Universitetet i Bergen av studenter som har hatt dette som oppgave i et fag eller som masteroppgaver. DPG 2.0 benytter presentasjonsmønsterspesifikasjonen beskrevet i seksjon 2.3. DPG 2.0 bruker det åpne applikasjonsrammeverket Spring [67] og benytter teknikken Dependency Injection [33] for å tilrettelegge for modularitet og testing. I tillegg benyttes Spring-modulen MVC for å få en Model-View-Controller [78] arkitektur.

For å realisere presentasjonsmønstre må presentasjonsmønsterspesifikasjonen implementeres. Dette er gjort i DPG 2.0 ved at presentasjonsmønstre skrives i formatet XML i en fil kalt `pattern.xml`. Her defineres alle elementene fra seksjon 2.3.

DPG 2.0 benytter spesifikasjonen Java Content Repository 170 (JCR-170) [58] for lagring og henting av innhold. Systemet benytter referanseimplementasjonen Jackrabbit [19] for å realisere dette. Spesifikasjonen benytter i stor grad eksisterende XML-standarder, XML brukes for representasjon av data og XPath benyttes som spørrespråk.

For transformering av utsnitt benyttes teknologien `Extensible Stylesheet Language Transformations` (`XSLT`) [76] versjon 1.0. XSLT gjør det mulig å omforme XML til annen XML. Det er ofte brukt for å gjøre XML om til Extensible HyperText Markup Language (XHTML) [75] eller HyperText Markup Language (HTML) [74], og det er også det XSLT brukes til i DPG 2.0.

For å rendrere *sider* benytter DPG 2.0 Velocity [31]-maler. En Velocity-mal beskriver en HTML-side, der en del parametere må fylles av brukeren av malen. I DPG 2.0 er en slik parameter som oftest et view. Velocitys oppgave i DPG 2.0 er å finne ut hvilke utsnitt som er aktiv og deretter rendrere disse.

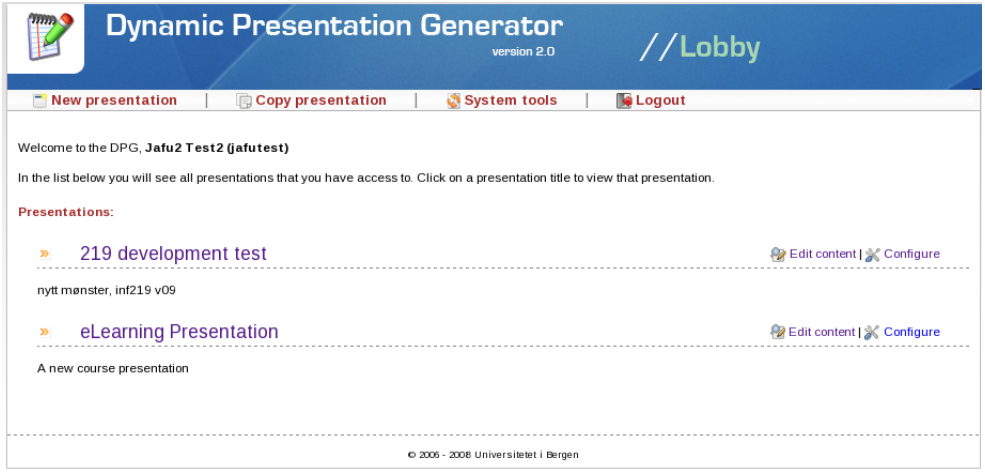
I DPG 2.0 finnes det også tre roller, `reader`, `publisher` og `admin`. Ved å ha forskjellige roller kan brukere få tilgang til forskjellige deler av DPG 2.0.

2.4.2 Delsystem i DPG 2.0

DPG 2.0 er inndelt i flere pakker som håndterer funksjonaliteten til forskjellige deler. Det mest av funksjonaliteten som kreves for å håndtere presentasjonsmønsterspesifikasjonen finnes i pakken *core*. Denne pakken håndterer også lagring av data, sikkerhet og en del diverse.

Pakken `Lobby` behandler innloggingssiden og inngangsportalen som brukeren møter etter innlogging. I denne portalen får brukeren listet opp alle presentasjoner som brukeren har

tilgang til. Hvis en bruker har en rollen *publisher* er det lenker til sider der innholdet i presentasjonen kan endres. Hvis brukeren er av rollen *admin* får vedkommende opp lenker til opprettelse av nye presentasjoner, og administrering av eksisterende presentasjoner. Figur 2.2 viser hvordan lobbyen ser ut dersom brukeren har tilgang til presentasjonene *219 development test* og *eLearning Presentation* og har rollen *admin*.



Figur 2.2: Lobby

DPG 2.0 er også tilknyttet brukerhåndteringssystemet Webucator 3.0 [50]. Ved innlogging vil Webucatorsystemet returnere tilgjengelige presentasjoner for brukeren i tillegg til hvilken rolle vedkommende har. Dette gjøres ved å kalle en vevtjeneste tilbudt av Webucator.

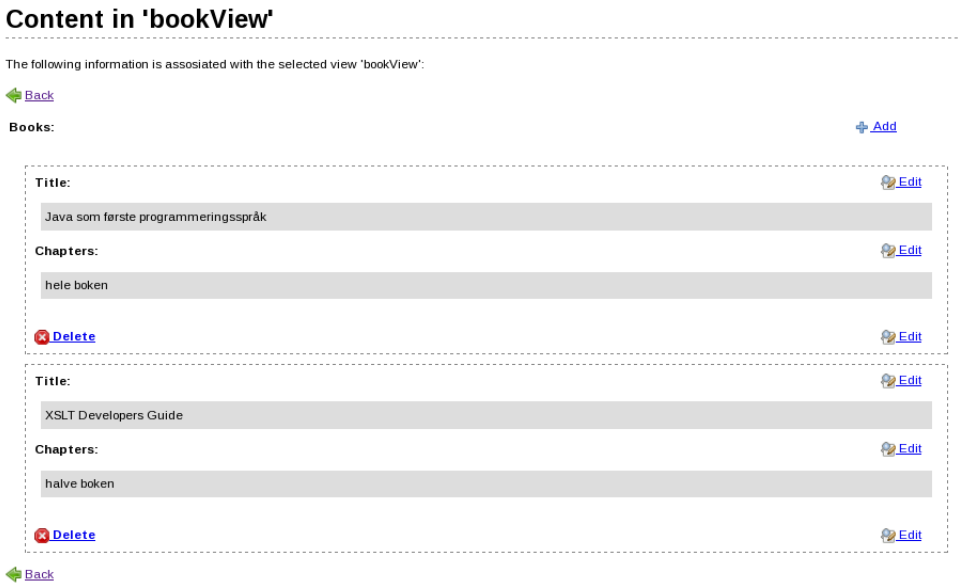
DPG 2.0 har også en pluginpakke for pluginhåndtering. I presentasjonsmønsterspesifikasjonen kan et felt i en entitet defineres som plugin. En plugin gir større fleksibilitet i utformingen av feltet. Plugin ble i all hovedsak benyttet i DPG 2.0 for å vise multimedia som video og lyd, eller flashbaserte spill og lignende. I kapittel 4: Analyse av pluginarkitektur vil denne pluginarkitekturen ses på i detalj, ettersom den spiller en viktig rolle i å nå målet om interaksjon i DPG 2.0.

De tre resterende pakker håndterer de tre delsystemene i DPG 2.0. Delsystemene er tett knyttet til de tre rollene `admin`, `publisher` og `reader`.

1. `Presentation Manager` (PM) benyttes av brukere med rollen `admin`. I dette delsystemet kan presentasjoner opprettes og tilpasses.
2. Delsystemet `Presentation Content Editor` (PCE) benyttes av rollen `publisher`. I PCE kan brukere legge til, redigere eller slette innhold, og ressurser håndteres.
3. Brukere med den tredje rollen, `viewer`, bruker delsystemet `Presentation Viewer` (PV). Dette delsystemet rendrerer endelige dokument som kan leses av sluttbrukere

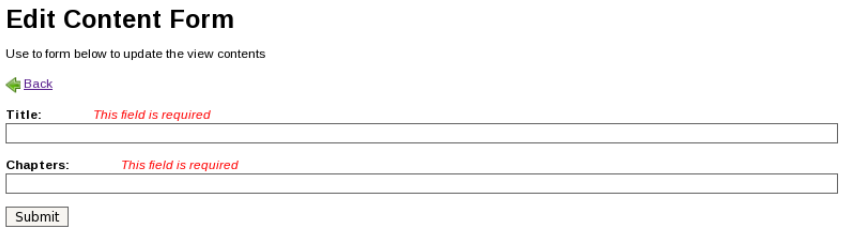
med rollen `viewer`. De kan lese informasjon lagt ut av `publishers` og laste ned tilgjengelige ressurser.

Figuren 2.3 viser skjermbildet i delsystemet PCE dersom en vil endre innholdet i utsnittet *bookView*.



Figur 2.3: PCE - Endring av ett view

Dersom en trykker `add`-knappen på toppen av skjermbildet, kommer man til siden som er viser på figur 2.4. Dette illustrerer hvordan innholdsendring foregår i DPG 2.0. Eksempelet viser også at en bruker ikke trenger å vite noe om vevutvikling for å publisere innhold i systemet.



Figur 2.4: PCE - Legge til en bok

Anbefalt lesning for detaljer om DPG 2.0 er Bjørn Christian Sebaks oppgave *Dynamic Presentasjon Generator 2.0 - Utvikling av ny dynamisk presentasjonsgenerator og presentasjonsmønsterspesifikasjon* [63]. Persistensløsningen er beskrevet at Karianne Berg i sin

oppgave *Persistensproblematikk i Dynamisk Presentasjonsgenerator* [4]. Bjørn Ove Ingvaldsen har skrevet om plugins og multimedia i sin oppgave *Multimedia i Dynamisk Presentasjonsgenerator 2.0*.

Kapittel 3

Problembeskrivelse

Med DPG 2.0 ble det dannet et svært godt fundament for videre utvikling. Den utviklede koden og de valgte løsningene var god, men på grunn av tidrammer ble det i DPG 2.0 ikke tid til å legge tilrette for interaksjon og søk etter innhold.

3.1 Hvordan tilrettelegge for interaksjon i DPG 2.0?

Interaksjon i denne oppgaven omhandler brukernes informasjonsutveksling til datasystemet og andre brukere.

I DPG 2.0 har ikke brukere med rollen `viewer` mulighet til å forme innhold i systemet. Brukere med rollen `viewer` kan lese innholdet på siden og laste ned ressurser. Målet bak denne oppgaven er å la `viewers` benytte seg av funksjoner som søk, og å tillate at de skal kunne bidra med eget innhold. Et eksempel på sistnevnte er muligheten til å legge igjen kommentarer, for eksempel på en nyhetsartikkel dersom presentasjonsmønsteret beskriver en avis.

Et spørsmål blir da hvordan denne funksjonaliteten kan integreres med DPG 2.0 på en måte som gjør interaksjon til en naturlig del av systemet. DPG 2.0 er tett knyttet opp mot presentasjonsmønstre, og det er derfor ønskelig å kunne se på muligheter der interaksjon blir en del av dette. Dette betyr at interaksjon enten må bli et nytt element i presentasjonsmønsterspesifikasjonen, eller at interaksjon integreres med et eksisterende element i spesifikasjonen.

En mulighet er å ha egne utsnitt dedikert til å håndtere interaksjon. Med dette forslaget må det tilrettelegges en ny måte å håndtere utsnitt på, som ikke tar hensyn til entiteter. Dette er en ugunstig løsning, ettersom det da blir et spesialtilfelle av utsnitt. En bedre løsning vil unngå spesialtilfeller og heller benytte entiteter og felt.

En annen løsning er å definere en ny innebygget datatype for interaksjon. Innholdet i det nye feltet som er håndtert av denne datatypen vil da kunne tilpasses ut fra hvilket type interaksjonsmulighet som er ønsket. Dette vil gjøre at presentasjonsmønsterspesifikasjonen vil få enda en datatype, noe som øker kompleksiteten, som igjen øker terskelen for å forstå spesifikasjonen.

En tredje løsning er å benytte seg av den eksisterende pluginarkitekturen for å håndtere interaksjon, ved å lage såkalte interaksjonsplugins. Dette forslaget er veldig likt det forrige forslaget, men vil benytte seg av den eksisterende pluginarkitekturen, istedenfor å lage en ny datatype. Dette vil gjøre at interaksjon skjer på feltnivå og vil ikke kreve endringer i presentasjonsmønsterspesifikasjonen. Forslaget krever en del tilpasninger av pluginarkitekturen, ettersom pluginarkitekturen i DPG 2.0 i all hovedsaklig ble brukt til å muliggjøre multimedia som video og lyd. En ny arkitektur vil kreve gode løsninger for lagring og uthenting av data, i tillegg til at plugins må ha mulighet til å håndtere inndata sendt fra bruker.

Den siste løsningen krever ingen endringer i presentasjonsmønsterspesifikasjonen og benytter seg av eksisterende løsninger, og ble derfor den valgte veien å gå. I kapittel 4 - Analyse av pluginarkitektur vil manglene i arkitekturen drøftes og løsninger beskrives.

3.2 Presentasjonsmønsterspesifikasjon

Presentasjonsmønsterspesifikasjonen lider i dag av overflødige elementer, og spesifikasjonen behandler flere datatyper enn nødvendig. Ved å skape en bedre presentasjonsmønsterspesifikasjonen vil det være enklere å utvikle plugins, som behøves for å klare målet om interaksjon i DPG. Presentasjonsmønsterspesifikasjonen må analyseres for å kunne gjøre den mer presis og dermed bedre. Dette gjennomgås i kapittelet 7: Ny presentasjonsmønsterspesifikasjon.

3.3 Søk

Søk er en viktig funksjonalitet i de fleste innholdshåndteringssystem. Det er svært viktig at systemet gjør det lett for brukere å finne innhold på en god og effektiv måte. I DPG 2.0 er det ingen muligheter for søk, noe som anses som et krav i de fleste innholdshåndteringssystem. Ettersom DPG 2.0 har en tett tilknytning til presentasjonsmønstre er det viktig å finne en løsning som integreres godt mot dette.

I søk er indeksering av innhold en viktig prosess. Indeksering handler om at informasjonen i systemet systematiseres og klargjøres for søking. Ettersom informasjonen i entiteter kan finnes i flere forskjellige utsnitt er det viktig å finne en god måte å indeksere innholdet

på, slik at det er mulig å gjenfinne informasjonen. Det er også viktig å ta hensyn til brukervennligheten, og det er viktig at løsningen ligner på eksisterende løsninger som er kjent for de fleste brukere.

Det finnes flere søketeknologier, og det må gjøres en vurdering av hvilke søketeknologier som passer best til DPG.

Kapittel 4

Analyse av pluginarkitektur

Dette kapittelet er skrevet i fellesskap av Peder Lång Skeidsvoll [65] og Tobias Rusås Olsen. Kapittelet inngår i begges masteroppgaver. Årsaken til at dette er et felleskapittel er at begge parter var interessert i å videreutvikle pluginarkitekturen for å kunne gjennomføre målene i sine masteroppgaver som henholdsvis omhandlet bruk av interaktive plugins og AJAX i DPG. Konklusjoner i dette kapittelet er gjort på grunnlag av samtale mellom de to forfatterene. Det er også kommet innspill fra veiledere under ukentlige møter.

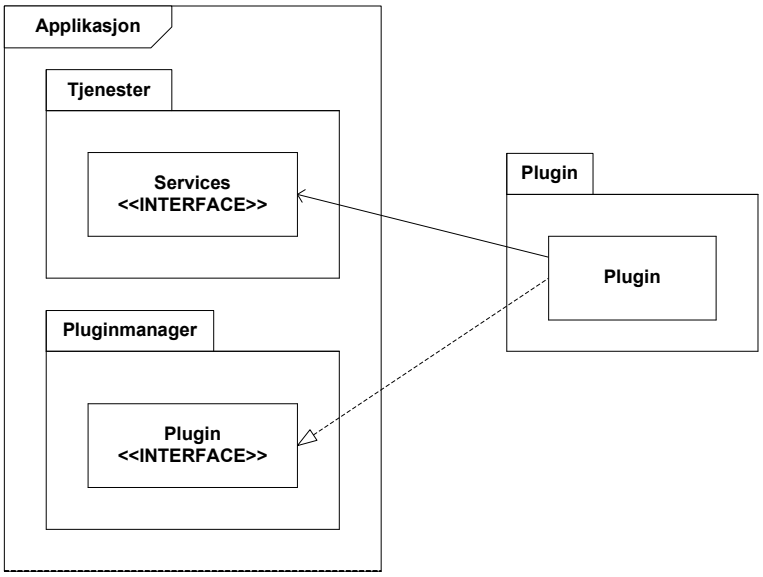
4.1 Generelt om pluginarkitektur

Plugins, også kalt *addins*, *add-ons*, *snapins* eller *extensions*, er et stykke programvare som interagerer med en *vertapplikasjon* (eng. *host application*) for å tilby en spesifikk funksjonalitet *på etterspørsel* (eng. *on demand*). Årsaker til at man kan ønske å benytte en pluginarkitektur kan være å:

- Gjøre det mulig for tredjepartsutviklere å utvide applikasjonen.
- Ha støtte muligheter som ikke er forutsett.
- Redusere størrelsen på vertapplikasjonen.
- Skille kildekoden fra applikasjonen av lisensårsaker.

Applikasjonen har ofte en egen pluginmodul som håndterer kommunikasjon mellom plugin og applikasjon.

For at en plugin skal kunne anvendes, må den registreres hos programmet som skal bruke den, den såkalte vertapplikasjon. I tillegg til dette trengs det en protokoll som gjør det mulig at plugin og vertapplikasjonen kan utveksle data. Figur 4.1 viser hvordan de forskjellige



Figur 4.1: Eksempel på hvordan vertapplikasjon og plugins kommuniserer.

komponentene kommuniserer. Vertapplikasjonen skal kunne opptre uavhengig av plugins, og bør helst ikke være avhengig av plugins for å kunne fungere. Dette betyr at applikasjonen ikke bør slutte å fungere dersom en plugin mangler. Ideelt sett skal det være mulig å legge til og endre plugins dynamisk, uten å måtte gjøre endringer i selve vertapplikasjonen.

Et eksempel på bruk av plugins finnes i nettlesere. Nettlesere bruker plugins for å vise for eksempel video og andre presentasjonsformater. Eksempler på dette er Adobe Flash [2], Apple QuickTime [43], Microsoft Silverlight [9].

4.2 Pluginarkitekturen i DPG 2.0

Pluginarkitekturen i DPG 2.0 ble utarbeidet av Bjørn Ove Ingvaldsen [46] i hans masteroppgaveprosjekt ved Institutt for Informatikk, UiB i 2008. Ingvaldsens oppgave var å gi DPG 2.0 mulighet til å behandle multimedia som video og bilder. Til dette formålet konkluderte han med at den mest fornuftige løsningen var å implementere en pluginarkitektur.

Pluginarkitekturen bygger på Martin Fowlers pluginmønster [34]. Dette mønsteret dikterer at det lages *kontrakter* (eng. *interfaces*) som plugins må implementere. Figur 4.1 viser et diagram over dette mønsteret. I DPG 2.0 finnes det to slike kontrakter:

- `SingleFieldInputPlugin`: Denne kontrakten tillater brukeren å legge inn et felt som inndata til plugin. Dette vises i listing 4.3. Metoden som må implementeres

heter `generateTag()` og returnerer et JDOM-element. Eksempler på bruk av denne kontrakten er plugins som håndterer et bilde eller en video.
- `EntityListInputPlugin`: Denne kontrakten tillater brukeren å legge inn en liste med entiteter som inndata. Metoden som må implementeres heter `generateTagWithEntityListInput()`. Den returnerer, som `SingleFieldInputPlugins` metode `generateTag()`, et JDOM-element. Eksempel på bruk er et bildegalleri der brukeren kan navigere mellom flere bilder.

En annen viktig del av pluginarkitekturen er hvordan plugins skal importeres til DPG. Dette er løst ved at utvikleren legger til et nytt element i en XML-filen `plugIn.xml`. Dette vises i listing 4.1. I denne XML-filen er alle plugins som brukes av DPG listet opp med sti til hvor man finner selve implementasjonen til plugins. samt litt annen informasjon. Ved oppstart går DPG gjennom hver plugin som er definert i filen, og importerer dem til systemet slik at de er klar til bruk.

4.3 Legge til plugins i en presentasjon i DPG 2.0

For å legge til en ny plugin må følgende steg gjennomgås:

1. Implementere en plugin.
2. Konfigurere plugin i filen `plugIn.xml`, i presentasjonsmønsteret og i pluginkonfigurasjonsfilen `pluginPattern.xml`.
3. Starte DPG på nytt slik at alle plugins lastes på nytt.

4.3.1 Implementasjon av en ny plugin

Implementasjon av plugins er en enkel prosess. Dette er stegene som må utføres:

- Opprette et nytt Java-prosjekt.
- Legge til JAR-filen `pluginInterfaces.jar` i *byggebanen* (eng. *build path*) til prosjektet. Denne JAR-filen finnes i DPG i `web/WEB-INF`-mappen.
- Opprette en ny klasse som implementerer enten kontrakten `SingleFieldInputPlugin` eller kontrakten `EntityListInputPlugin` - den første for et enkelt felt og den andre for en liste med entiteter. Pluginet må implementere `generateTag()`-metoden.
- I metoden `generateTag()` opprettes det et `Element`-objekt fra JDOM-pakken. Et slikt element representerer en XML-tag, og kan derfor ha tekst og kan tilknyttes attributter eller underelementer. Dette gir stor fleksibilitet i utformingen av innholdet som plugin returnerer.

Når denne prosessen er gjennomført kan pluginet brukes i DPG.

4.3.2 Konfigurering av plugins i DPG 2.0

4.3.2.1 Gjøre plugins klar for bruk i DPG 2.0

For å bruke plugins i en presentasjon må systemet ha tilgang til pluginet. Dette gjøres ved at pluginutvikleren eksporterer klassen med pluginet som en JAR-fil, og plasserer filen i DPG sin `lib`-mappe. Når dette er gjort må utvikleren redigere XML-filen `plugIn.xml`. I denne filen defineres alle plugins som skal benyttes i applikasjonen. For hver plugin defineres *stien* (eng. *path*) til pluginet, navnet på pluginet som brukes i DPG 2.0, hvilken kontrakt som er implementert (`SingleFieldInputPlugin` eller `EntityListInputPlugin`), og hva slags type inndata pluginet krever. Denne inndatatypen kan være av *string*, *date*, *entity-list* eller *file*.

Listing 4.1 viser et eksempel på hvordan en plugin er definert. I eksempelet er det definert at navnet på pluginet er `JavaToHTMLPlugin`, den forventer inndata av typen `file`, og den skal kun ha én inndata ettersom den implementerer kontrakten `SingleFieldInputPlugin`. I feltet `class-name` skrives stien til hvor pluginimplementasjonen finnes.

Listing 4.1: Konfigurasjonsfil for Java-til-HTML-plugin.

```
<plugin>
  <class-name>javaToHtml/JavaToHtmlPlugin.class</class-name>
  <interface>SingleFieldInputPlugin</interface>
  <plugin-name>JavaToHTMLPlugin</plugin-name>
  <formfield-type>file</formfield-type>
</plugin>
```

4.3.2.2 Legge til plugin i en presentasjon

Det neste steget er å legge til pluginet i en presentasjon. Først går man inn i presentasjonsmønsteret og finner filen `pluginPattern.xml`. Listing 4.2 er et eksempel på hvordan en plugin defineres i denne filen.

Listing 4.2: Konfigurasjon av plugin i pluginPattern.xml.

```
<!-- Her definerer det at pluginet MinPlugin skal kalles minPlugin i -->
<!-- pattern.xml                                                      -->
 <plugin-config id="minPlugin" plugin-name="MinPlugin">
  <!-- Den neste linjen definerer et parameter -->
  <param name="mittParameter">verdi</param>
</plugin-config>
```

I denne filen kan det legges til inndata, kalt *parameterne*, til pluginet. Dersom det arbeides med en bildeplugin kan eksempler på slike parametre være *høyde* og *bredde*. Disse parametrene skiller seg fra inndata til plugins ved at disse parameterene ikke kan endres i delsystemet PCE. Fordelen med å ha parameterkonfigurering her er at en kan ha unike verdier for hver presentasjon. Dette betyr at en kan bruke samme bildeplugin i flere presentasjoner, men likevel ha unik høyde og bredde per bilde per presentasjon.

Deretter må utvikleren inn i filen `pattern.xml` for å knytte plugin opp mot et felt i en entitet. Dette gjøres ved å legge til et nytt felt i en entitet, definere at feltet skal håndteres av en plugin, og deretter si hvilken plugin som skal håndtere feltet. I listing 4.3 vises det et eksempel på en entitet med en felt som håndteres av en plugin og et felt som håndteres av en streng.

Listing 4.3: Definering av et felt som plugin

```
<entity id="testEntity">
  <fields>
    <field type="string">testInnhold</field>
    <field type="plugin" pluginConfig="minPlugin">minPlugin</field>
  </fields>
</entity>
```

Verdien til attributtet `pluginConfig` må være lik navnet som er gitt på en plugin i filen `pluginPattern.xml`.

Det siste steget er å legge til et XSLT-uttrykk i transformasjonsfilen som tilhører utsnittet. I dette tilfellet kan det være:

```
<xsl:copy-of select="minPlugin/*" />
```

Til slutt må man inn i selve presentasjonen og legge til et nytt element som stemmer overens med det som er definert i XSLT-dokumentet. Elementet må ha en attributt av type `type` som er satt til verdien `plugin` for at man skal kunne legge til inndata til feltet i PCE. Dette feltet burde kunne automatisk genereres hvis det mangler, men i DPG 2.0 må det legges til manuelt. Dette vises under.

```
<minPlugin type="plugin">innhold</minPlugin>
```

4.3.3 Initialisering av plugins

Innlastingen av plugins i DPG 2.0 gjøres ved at systemet går gjennom hver plugin i filen `plugIn.xml`. Ut fra hva som står definert i denne filen, kobler den hvert plugin-navn opp

mot rikig plugin-klasse. Klassen må ligge i systemets *byggebane*. Etter dette er gjort kobler DPG seg sammen med pluginet ved hjelp av *refleksjon* (eng. *reflection*) [56].

Denne operasjonen kjøres automatisk ved oppstart av DPG.

4.4 Svakheter og løsninger i nåværende pluginarkitektur

Etter å ha gått gjennom pluginarkitekturen, ble det avdekket en rekke mangler og svakheter slik den ble ferdigstilt i DPG 2.0. Nå vil svakhetene analyseres og forslag til forbedring bli beskrevet. De faktiske implementasjonsendringene er diskutert i kapittel 5.

4.4.1 Bruk av plugins i presentasjonsmønstre

I DPG 2.0 er det to måter å definere datatypen til felter i entiteter:

- Definere at feltet er av en innebygget datatype som `string`, `xhtml` eller `file`.
- Definere at feltet skal bruke `plugin` som datatype.

At det er to forskjellige måter å definere typen til et felt på blir vurdert som en svakhet, siden det kan løses ved kun å bruke én måte. Det er ønskelig at de innebygde datatypene også skal kunne håndteres av en plugin. Da kan definisjonen av datatypen fjernes og en trenger bare å definere hvilken plugin som skal håndtere feltet. Dette vil føre til at man får en generell mekanisme for håndtering av felt, som konseptuelt er en bedre løsning. Dette vil gjøre systemet enklere å forstå og lettere å vedlikeholde. For å implementere dette trengs det plugins som håndterer de innebygde datatypene som `string` og `file`.

4.4.2 Forenkling av plugininstallasjon

I delkapittel 4.3.2 ble prosessen med å lage og integrere en plugin i DPG forklart. Dette er en unødvendig komplisert prosess med flere fallgruver. Deriblant ble det oppdaget at filen `plugIn.xml` stort sett innholdt informasjon som kunne automatiseres ved hjelp av diverse teknikker, som refleksjon eller ved å endre på pluginarkitekturen.

Det ble derfor konkludert med at XML-filen `plugIn.xml` er overflødig. Dens arbeidsoppgaver kan automatiseres ved hjelp av refleksjon og ved at plugins selv sørger for å holde rede på hva slags inndata de krever. Det er plugin selv som vet best hva slags type inndata og hvilke parametere den trenger for å fungere.

En fordel med å beholde filen `plugIn.xml` er at man da vet hvor man skal se for å finne ut hvilke plugins som er tilgjengelig i DPG. Likevel er ikke dette en god nok grunn for å komplisere prosessen med å inkludere nye plugins. Dersom en lurer på hvilke plugins som er tilgjengelig for bruk, kan man se på stien der JAR-filene som innholder implementasjonen av forskjellige plugins er lokalisert.

4.4.3 Ressurstilgang

Plugins har ikke tilgang til ressurser fra andre entiteter. Dette ville være en nyttig mulighet dersom plugins skal brukes til å behandle informasjon fra presentasjonen den befinner seg i.

Et eksempel på når slik tilgang vil være nyttig, er en plugin som henter ut informasjon fra en entitetsliste og presenterer den på en ny måte. Noe tilsvarende er gjort på nettstedet *fxdteam.com* [36], som benytter seg av det åpne API-laget til musikknettstedet *last.fm* [49]. De henter informasjon om hvilke sanger en bruker har hørt på for å genere en *merkelappsky* (eng. *tagcloud*) [47] som på en lettfattelig måte gir oversikt over musikksjangrene vedkommende lytter på. De sjangrene som er lyttet på flest ganger er vist i større skriftstørrelse enn de som er lyttet på færre ganger. I figur 4.2 kan man se at sjangeren *Minimal* har størst skriftstørrelse, noe som betyr at den er den mest populære sjangeren hos denne brukeren. De små navnene under sjangeren er eksempler på populære artister eller grupper som opererer i sjangeren.

Etter å ha sett nærmere på om ressurstilgang virkelig ville være nyttig, ble det konkludert med at det var best å løse slike situasjoner uten å benytte plugins. Det å presentere informasjonen fra entiteter i forskjellige utsnitt på forskjellige måter er en del av nøkkelfunksjonaliteten til DPG, og det er utbredt støtte for forskjellige måter å sortere eller filtrere informasjonen i XSLT-transformasjonen.

En fremtidig løsning kunne vært å tilby et API-lag som programmerere kan benytte for å hente ut informasjon fra systemet, men nytteverdien for denne løsningen er liten. Slike API-lag krever at formen av dataene er faste. DPG har en dynamisk struktur, og derfor er det vanskelig å hente ut data slik det ofte gjøres i slike API-lag.

4.4.4 Filsystemplassering av plugins

I dagens system legges plugins i mappen `lib` under DPGs rotmappe. Denne mappen brukes også til alle andre bibliotek, som for eksempel loggsystemet log4j [30] og XML-biblioteket JDOM. Dersom man skal gjøre endringer i `lib`-mappen er det ikke lett å få oversikt når bibliotek og pluginfiler er i samme mappe. En bedre løsning er å plassere plugins i en egen undermappe under `lib`-mappen.

Figur 4.2: Eksempel på merkelappskyplugin til last.fm.

4.4.5 Hendelseshåndtering

Plugins kan i skrivende stund ikke reagere på hendelser. Det hadde vært nyttig å ha denne funksjonaliteten dersom en plugin skal kunne oppdatere innholdet sitt basert på for eksempel innholdsendring i systemet, eller at et visst klokkeslett inntreffer. Dette er en såpass sentral funksjon at det er ønskelig med en generell mekanisme som håndterer dette.

4.4.6 Skrive- og lesetilgang

Plugins har ikke mulighet til å lese eller skrive til filer. Dette må være mulig for brukere av presentasjonen dersom man har en plugin som håndterer lagring av kommentarer. For å lese av kommentarer trengs det en plugin med lesetilgang, slik at man kan hente ut lagrede kommentarer. Et annet eksempel der lagring vil være fornuftig er dersom man har et pluginspill og ønsker å lagre de beste resultatene.

Den valgte løsningen benytter den eksisterende klassen `ResourceDao` for lagring av data. Denne klassen bruker Java Content Repository (JCR). Flere av parameterne som kreves i klassen `ResourceDao` kan settes automatisk, og det er derfor fornuftig å benytte et *fasademønster* (eng. *facade-pattern*) [37] for å få en bedre kontrakt å jobbe mot.

4.4.7 Inndataløse plugins

Et av feltene som må defineres i filen `plugIn.xml` for hver plugin er feltet `field`. Dette feltet beskriver hva slags data som forventes av rollen *publisher* når innholdet i presentasjonen blir redigert i delsystemet PCE. Dette feltet kan være `string`, `date` eller `file`. I flere scenarioer vil det derimot oppstå situasjoner der inndata ikke er ønskelig.

Et eksempel på en slik situasjon er sporing av brukeraktivitet ved hjelp av verktøyet Google Analytics [40]. Dette verktøyet krever at sidene inkluderer et fire linjer langt Javascript som håndterer koblingen til Google. Det eneste som må forandres i denne kodesnutten er at man må definere en *bruker-ID*. Denne bruker-IDen vil være lik for alle sidene i en presentasjon. Den beste løsningen vil i denne situasjonen være å definere bruker-ID som en parameter i konfigurasjonsfilen `pluginPattern.xml`, innholdet i denne filen kan ikke forandres av en publisher. Bruker-ID er det eneste konfigurerbare med en slik plugin, og derfor trenger ikke plugin et eget inndatafelt.

4.4.8 Redgjøring av inndata og parametre

En plugin kan hente inndata fra to forskjellige kilder. Den første kilden er gjennom å forandre presentasjonen, det vil si at en bruker som er med i brukergruppen *publisher* gjør forandringer i skjemaene til delsystemet PCE. Den andre kilden er parametre som settes i filen `pluginPattern.xml` (vist i 5. linje i listing 4.2).

Omverdenen har likevel ikke noen mulighet for å vite hva pluginen forventer av hverken parametre eller inndata i presentasjonen. Dette gjør at for eksempel et fremtidig redigeringsverktøy som skal hjelpe utvikleren med å produsere korrekte presentasjonsmønstre vil få problemer. Dersom et redigeringsverktøy eller en utvikler vil vite hvilken type inndata eller parametere en plugin trenger, må den inn i implementasjonen av pluginet og finne ut hvor inndata fra parameteret `parameterMap` eller `idOfResource` er brukt.

For å gjøre det enklere å finne ut hvilke parametere og inndata en plugin trenger, bør det derfor opprettes metoder som returnerer dette.

4.4.9 Tillate inndata til plugins

Plugins i DPG 2.0 tillater kun inndata fra delsystemet PCE og parametere i filen `pluginPattern.xml`. Det er ønskelig å utvikle støtte for at også brukere med brukerrollen *reader* skal kunne sende data til systemet.

4.4.10 Pluginkontrakter

Pluginkontraktene er definert på en svært uoversiktlig måte. Dette skyldes først og fremst at kontraktene har for mange parametere, `SingleFieldInputPlugin` og `EntityListInputPlugin` har henholdsvis elleve og tretten parametere som vist i listing 4.4 og 4.5. Med så mange parametere er det svært vanskelig å få kontroll over parameterrekkefølgen. Det hjelper heller ikke at parameterrekkefølgen på kontraktene er inkonsekvent, de inneholder flere like parametere, men de dukker opp i forskjellige rekkefølge i de to kontraktene. I følge boken *Clean Code* bør man unngå å ha flere enn to parametere i en metode [51].

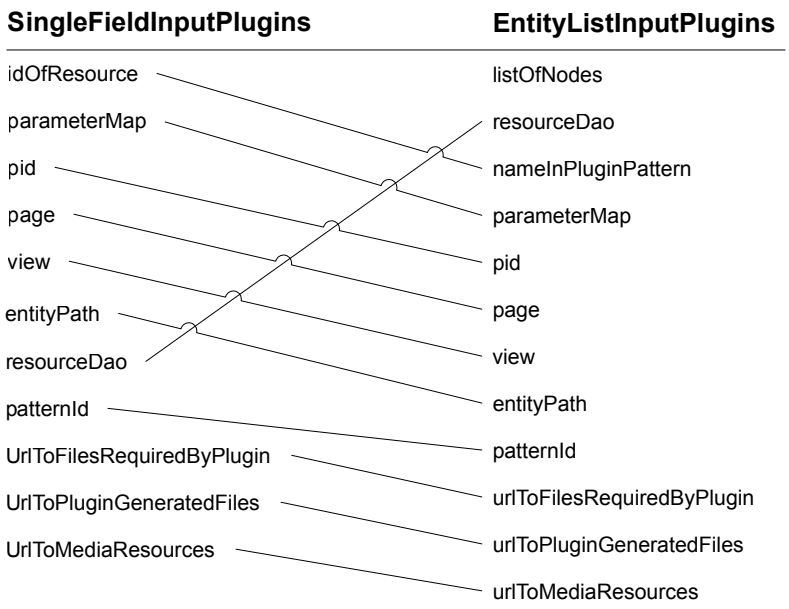
Listing 4.4: SingleFieldInputPlugins sin metode for generering av pluginelement

```
public Element generateTag(
  String idOfResource,
  Map<String,String>parameterMap,
  String pid,
  String page,
  String view,
  String entityPath,
  ResourceDao resourceDao,
  String patternId,
  String UrlToFilesRequiredByPlugin,
  String UrlToPluginGeneratedFiles,
  String UrlToMediaResources);
```

Listing 4.5: EntityListInputPlugins metode for generering av pluginelement

```
public Element generateTagWithEntityListInput(
  List<?> listOfNodes,
  ResourceDao resourceDao,
  String nameInPluginPattern,
  Map<String,String> parameterMap,
  String pid,
  String page,
  String view,
  String entityPath,
  String patternId,
  String urlToFilesRequiredByPlugin,
  String urlToPluginGeneratedFiles,
  String urlToMediaResources);
```

Et eksempel på den nevnte inkonsekventheten er parametret `resourceDao`, som forekommer som nummer syv i `SingleFieldInputPlugin` og som nummer to i `EntityListInputPlugin`. For en oversikt over inkonsistensen, se figur 4.3. Strekene forbinder de ekvivalente parametrene i de to pluginkontraktene `SingleFieldInputPlugins` og `EntityListInputPlugins`. Det er viktig å påpeke



Figur 4.3: Forskjeller mellom de to kontraktene.

at feltene `idOfResource` og `nameInPluginPattern` inneholder samme informasjon, og at de tre siste parameterene er skrevet med forskjellig bokstavtype. I følge Javas kodekonvensjoner [53] skal variabler begynne med liten forbokstav, og de tre siste parameterne i listing 4.4 bør endres til liten forbokstav.

En bedre og mer oversiktlig løsning for å håndtere disse parameterene er å benytte seg av en pluginbønne som har kontroll på de forskjellige parameterene. Dette vil bryte bakoverkompabilitet, men pluginkontraktene bør uansett endres på grunn av inkonsekvens i parameterrekkefølgen.

Kapittel 5

Endringer i pluginarkitektur

Dette kapittelet er skrevet i fellesskap av Peder Lång Skeidsvoll og Tobias Rusås Olsen. Kapittelet inngår i begges masteroppgaver. Årsaken til at dette er et felleskapittel er at begge parter var interessert i å videreutvikle pluginarkitekturen for å kunne gjennomføre målene i sine masteroppgaver, som henholdsvis omhandler bruk av interaktive plugins og AJAX i DPG. Konklusjoner i dette kapittelet er gjort på grunnlag av samtale mellom de to forfatterene. Det er også kommet innspill fra veiledere under ukentlige møter.

5.1 Utbedringer

I kapittel 4 ble pluginarkitekturen til DPG 2.0 gjennomgått, og en rekke svakheter og mangler ble avdekket. I dette kapittelet vil utbedringene som er utført på systemet bli gjennomgått.

5.1.1 Bruk av plugins i presentasjonsmønstre

Det er blitt bestemt å fjerne innebygde typer, og la alle felt være prosessert av plugins. Les mer om dette i kapittel 7: *Ny presentasjonsmønsterspesifikasjon.*

5.1.2 Endring av navn på filen `pluginPattern.xml`

I den orginale versjonen av pluginarkitekturen fantes det en konfigurasjonsfil med navn `pluginPattern.xml`. Denne filen ble brukt til å konfigurere plugins. Med konfigurering menes det å gi hver enkelt plugin et navn som brukes i `pattern.xml`, og parametere som inndata til pluginet. Filnavnet `pluginPattern.xml` gir ingen informasjon om dette.

Derfor er det besluttet å gi filen det nye navnet `pluginConfig.xml`, som reflekterer filens innhold på en bedre måte.

5.1.3 Forenkling av plugininstallasjon

Ved å fjerne konfigurasjonsfilen `plugIn.xml` er plugininstallasjonen gjort betraktelig enklere. Denne filen ble brukt til å laste inn plugins til DPG 2.0, slik at de kunne benyttes i presentasjonsmønstre.

For å kunne fjerne denne filen måtte det finnes erstatninger for dens bruksområder. Dette er arbeidsoppgavene til konfigurasjonsfilen `plugIn.xml`:

1. Definere sti til hvor plugins befinner seg i filsystemet.
2. Finne ut hvilken type inndata pluginet forventer, for eksempel `string`, `xhtml` eller `file`.
3. Gi pluginet et navn til bruk i pluginkonfigurasjonsfilen `pluginPattern.xml`.
4. Definere hvilken kontrakt pluginet implementerter, `SingleFieldInputPlugin` eller `EntityListInputPlugin`.

For å unngå å måtte definere hvor hver enkelt plugin er plassert, er det vurdert at den beste løsningen er å spesifisere en sti til én mappe som inneholder plugins. Stien til mappen med plugins blir da definert i en *innstillingsfil* (eng. *property file*). Det ble bestemt å legge stien i filen `repository.properties`, siden dette er filen der alle andre stier som omhandler DPGs *oppbevaringssted* (eng. *repository*) er definert. Dette er et eksempel der pluginstien settes til mappen `lib/plugins`:

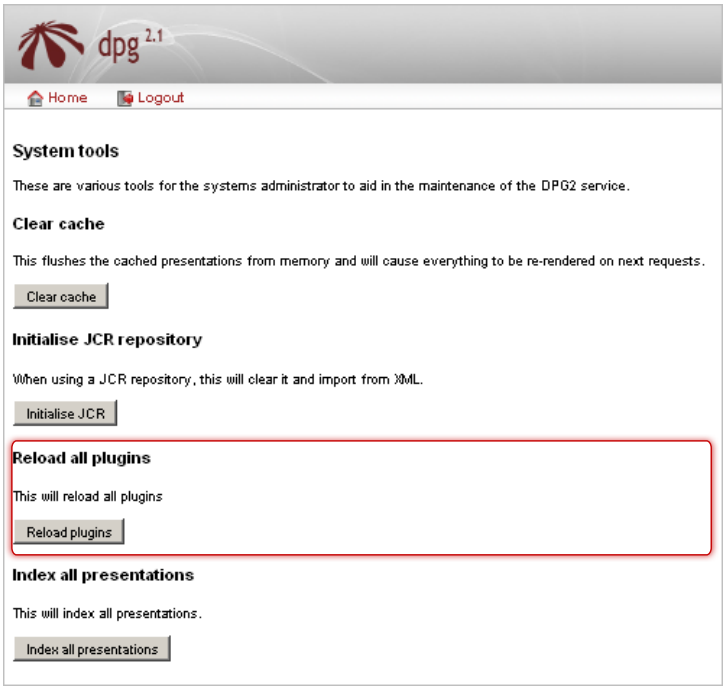
```
repository.path.plugins = lib/plugins
```

Når det er laget en plugin, pakkes den inn i en JAR-fil og legges i denne mappen.

DPG 2.1 vil ved oppstart, eller ved å trykke knappen merket *Reload Plugins* i systemets *Verktøy*-side (siden vises i figur 5.1), automatisk sjekke alle JAR-filer etter klasser som implementerer grensesnittet `SingleFieldInputPlugin` og laste disse inn i systemet ved hjelp av refleksjon. På denne måten blir punkt 1 og 4 løst.

For å løse punkt to, ble det lagt til en ny metode i kontrakten `SingleFieldInputPlugin`. Metoden har følgende signatur:

```
public String getFieldType();
```



Figur 5.1: Verktøyvinduet til DPG 2.1. Feltet for å laste inn plugins er rammet inn.

Metoden returnerer hvilken type inndata pluginet forventer, for eksempel `file` eller `string`.

I forhold til punkt 3 er en mulig løsning at DPG kan bruke klassenavnet til pluginet som navn. Det vil si at hvis klassenavnet til en plugin er `MapPlugin`, vil navnet til pluginen i `pluginConfig.xml` bli det samme. Likevel bør man ha muligheten til å gi pluginet et annet navn dersom det skulle være behov for det. Et eksempel er hvis man ønsker en plugin med navn `String`. Uten alternativ løsning ville dette navnet ha havnet i konflikt med `String`-klassen i Java-biblioteket.

Dette kan løses ved å legge til en ekstra metode i kontrakten, for eksempel `getName()`. Dette ville vært trivielt å implementere, men ville forkludret kontrakten uten god grunn. DPG må dessuten ha sjekke returverdien til `getName()`-metoden for verdien `null` hvis man skal ha klassenavn som standardverdi.

En annen løsning er å ha en abstrakt klasse som et mellomledd mellom kontrakten og pluginet. Denne klassen kan da ha en implementasjon av metoden `getName()` som returnerer klassenavnet, men som kan overskrives i pluginimplementasjonen dersom man ønsker å ha noe annet enn standardverdien. Ulempen med denne løsningen er at plugins da ville utvidet en klasse framfor å implementere et grensesnitt, noe som ikke er anbefalt [37]. Likevel vil denne implementasjonen løse problemet på en tilfredsstillende måte.

Løsningen som er valgt er å bruke *annoteringer* (eng. *annotations*) [54]. Listing 5.1 viser

et eksempel på hvordan man bruker en slik annotering.

Listing 5.1: Navnsetting

```
@PluginName(name="String")
public class MyAnnotatedStringPlugin implements SingleFieldInputPlugin {
  ...
}
```

I denne løsningen sjekker DPG 2.1 om klassen er annotert med `@PluginName`. Dersom den er det, vil den få det annoterte navnet i systemet. Hvis den ikke har noen annotasjon, vil pluginet bli navngitt etter navnet på klassen.

5.1.4 Ressurstilgang

Som nevnt i delkapittel 4.4.3 ble det enighet i at det ikke er en god idé å gi plugins tilgang til ressurser fra andre utsnitt, og dette er derfor ikke implementert.

5.1.5 Hendelseshåndtering

Hendelseshåndtering vil uten tvil være en veldig nyttig funksjon. Det er likevel konkludert med at det ikke blir prioritert å bruke tid på å implementere denne løsningen, da dette vil være en relativt stor og tidkrevende oppgave som vil gå på bekostning av oppfyllelsen av oppgavens hovedmål.

Senere masterstudenter oppfordres til å utvikle en løsning for dette.

5.1.6 Skrive- og lesetilgang

Skrive- og lesetilgang er en essensiel funksjonalitet. Dette er fordi avanserte plugins må ha tilgang til å lagre og hente ut data. Det er utviklet en løsning som bruker DPGs eksisterende arkitektur for å lagre ressurser. Denne arkitekturen har en kontrakt med navn `ResourceDao` som inngangsport.

Listing 5.2: Eksempel på metode fra `ResourceDao`

```
public void importPresentationResource(
        String presentationId,     // unik id for en presentasjon
        String resourceName,       // Hvor innholdet skal lagres og unikt
            navn
                                   //                              for
                                       innholdet
        String parentFolderPath,   // mappe der innhold skal lagres
        String mimetype,           // definisjon av innholdstype
```

```
        ResourceType type,          // type ressurs
        InputStream resourceStream // innholdet som skal lagres
);
```

`ResourceDao` tilbyr en rekke metoder for lagre eller hente ut data fra persistenslaget. I listing 5.2 vises et eksempel på en typisk metode fra denne kontrakten. Metoden viser hvilke parametre som kreves for å importere en ressurs.

Det vil nå bli gjennomgått hvilke parametre som er nødvendige, og hvilke som det ikke er behov for.

Når en plugin vil lagre en ressurs vil den alltid vite hvilken presentasjon den befinner seg i. Det er derfor ikke nødvendig å eksplisitt oppgi hvilken presentasjon den vil lagre ressursen i. Det er heller ikke ønskelig at en plugin skal kunne lagre data i en annen presentasjon. Parameteret `presentationId` er derfor ikke nødvendig.

Parameteret `parentFolderPath` er ikke tatt i bruk i dagens system og er derfor ikke nødvendig å oppgi.

Parameteret `ResourceType` er en *oppramstype* (eng. *enumeration*) der en plugin alltid vil ha verdien `ResourceType.PLUGIN_RESOURCE`, det er derfor ikke nødvendig å oppgi dette parameteret eksplisitt.

På grunn av disse observasjonene, ble det bestemt å lage en ny kontrakt for lagring av pluginressurser. Denne kontrakten fikk navnet `PluginDao` og bruker et *fasademønster* (eng. *facade pattern*) [37]. Et fasademønster brukes til å tilby et enkelt grensesnitt til et komplekst undersystem. Metoden for å lagre en ressurs til persistenslaget, som ble vist i listing 5.2, er i `PluginDao` blitt forenklet til å bli som i listing 5.3. Metoden `saveResource()` trenger kun tre parametre, en klar forenkling i forhold til metoden `importPresentationResource()` som krever seks.

Listing 5.3: Eksempel på metode fra `PluginDao`

```
public String saveResource(
  String filePath,
  String mimeType,
  InputStream resource
);
```

Klassen `ResourceDao` lagrer data som en nøkkelverditabell ved at det blir defineret en nøkkel som passer til en verdi. Utvikleren må bruke nøkkelen for å hente verdien. I dette tilfellet er parameteret `filePath` nøkkelen og parameteret `resource` verdien. I tillegg knyttes det til en `mimeType` som gjenspeiler hvilken type data som lagres. *MIME* (*Multipurpose Internet Mail Extensions*) [6] er en tekststreng som forteller mottakeren av ressursen hva slags type ressurs det er. Dette for at mottakeren skal kunne vite hva den skal gjøre med den. For eksempel vet en nettleser at en MIME-type som er av typen

`application/xhtml+xml` skal behandles som HTML, mens en MIME-type av typen `image/gif` skal behandles som et bilde.

Denne løsningen gir stor fleksibilitet i forhold til hva man velger å lagre. Det som lagres er av klassen `InputStream` og man kan derfor lagre stort sett hva man vil, enten det er en bildefil, en XML-fil eller et lydspor. Under vises stien som benyttes for å kunne rette en forespørsel til serveren som returnerer en ressurs.

```
getPatternResource.html?patternId=patternId&type=
                                    PLUGIN_RESOURCE&fileId=filePath
```

Selv om dette er en bedre løsning en den som var der fra før, er den ikke optimal. Et av de største problemene er at plugins kan skrive over hverandres ressurser. Dette er fordi det ikke finnes en egen lagringsplass for hver enkelt plugin, eller noen form for eierskap på ressursene. Dette bør håndteres i en fremtidig versjon av DPG.

5.1.7 Inndataløse plugins

Plugins som ikke trenger inndata fra brukerrollen *publisher*, kan få slippe det. Dette gjøres ved å definere at pluginets metode `getFieldType()` returnerer strengverdien *none*. Dersom en *publisher* vil endre innholdet på en side der en slik plugin finnes, vil ikke feltet dukke opp over listen med felt med inndata.

5.1.8 Redegjøring av inndata og parametre

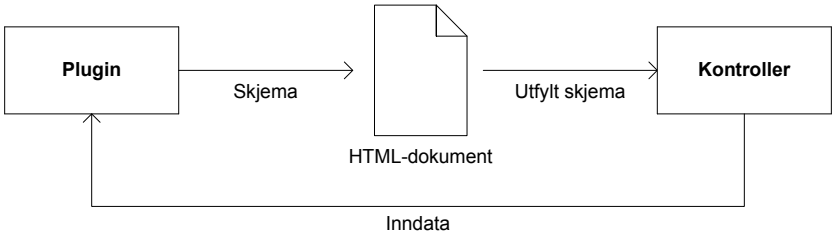
For å gjøre det lettere for fremtidige utviklingsverktøy for presentasjonsmønstre å forstå hvilke parametre en plugin støtter, er alle plugins utrustet med metoden `getParameters()`. Slik ser signaturen til metoden ut:

```
public List<String> getParameters();
```

Metoden returnerer en liste med navnene til alle parametrene som plugin-programmereren trenger å ta hensyn til.

5.1.9 Tillate at plugins håndterer inndata

Det er utviklet støtte for plugins som kan få inndata fra brukere med brukerrollen *reader*. For å gjøre dette mulig er det utviklet en kontrollerklasse med navn

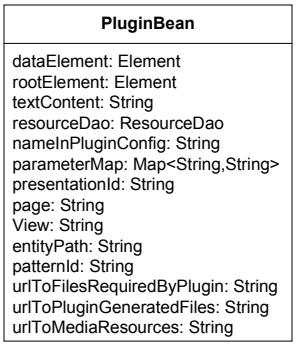


Figur 5.2: Interaksjon fra bruker.

`PluginFormController`. Kontrollerens oppgave er å håndtere inndata som er sendt fra brukeren. Inndataen blir sendt gjennom et HTML-skjema generert av pluginet og fylt ut av brukeren. `PluginFormController` har et forespørselsobjekt som inneholder inndata. Inndata blir omgjort til en nøkkelverditabell i kontrolleren. Nøkkelverditabellen sendes så tilbake til pluginet i metoden `processInputs()`. Dette er en ny metode som alle plugins må implementere. Plugins kan så behandle denne nøkkelverditabellen etter eget ønske. Figur 5.2 viser hvordan denne prosessen foregår.

5.1.10 Pluginskontrakter

Det er valgt å lage en *bønne* (eng. *bean*) med navn `PluginBean`. Denne bønnen inneholder alle parametrene som tidligere ble sendt som metodeparametre (se figur 4.3). I tillegg til dette er det valgt å fjerne hele `EntityListInputPlugins`-kontrakten fordi den har så mye overlappende funksjonalitet slik at den ekstra funksjonaliteten like gjerne kan legges til i kontrakten `SingleFieldInputPlugins`. Metodene til `PluginBean` vises i figur 5.3.

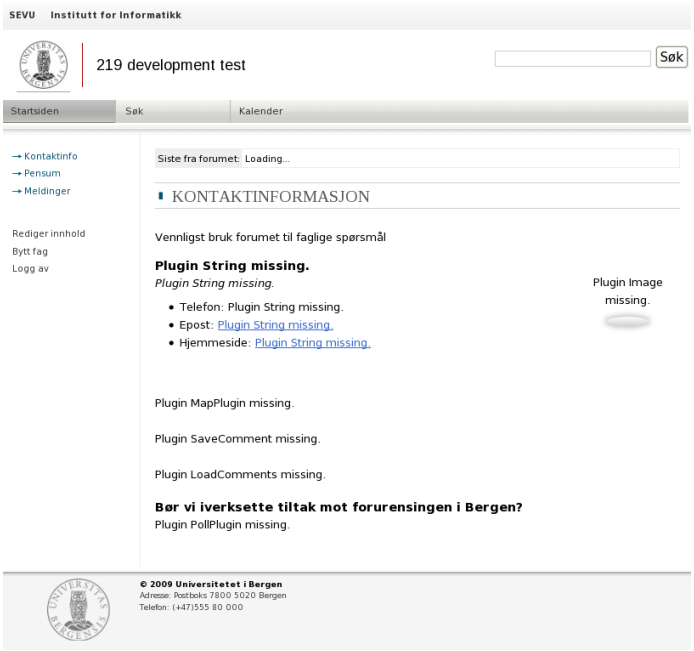


Figur 5.3: Den nye PluginBean klassen.

5.1.11 Feilhåndtering av plugins

Det er et krav til pluginapplikasjoner at feilende plugins ikke skal få applikasjonen til å feile. Det er derfor implementert en endring som gir beskjed på nettsiden hvis em plugin mangler. Man vil også få beskjed om hvilken plugin det er som mangler, slik at man skal kunne håndtere problemet uten å lese i systemets logger.

Feilhåndteringen er løst ved å undersøke om plugin-navnet definert i filen `pluginConfig.xml` er lastet inn i systemet. Dersom pluginet ikke finnes, returneres det istedenfor et JDOM-element med innhold som forklarer at pluginet mangler. Figur 5.4 viser hvordan siden ser ut dersom alle plugins mangler.



Figur 5.4: Skjermbilde av side der alle plugins mangler.

5.2 Konklusjon

Det har nå blitt gjennomgått en rekke forandringer som er utført i DPGs pluginarkitektur. Forandringene har ført til at det nå er lettere å:

- Installere plugins.
- Bruke plugins i presentasjonsmønstre.
- Utvikle kraftige plugins.

- Videreutvikle DPGs pluginarkitektur.

I kapittel 7 undersøkes det hvordan presentasjonsmønsterspesifikasjonen kan forbedres for å støtte den nye plugin-arkitekturen.

Kapittel 6

Interaksjonsplugins

6.1 Interaksjonsevolusjon

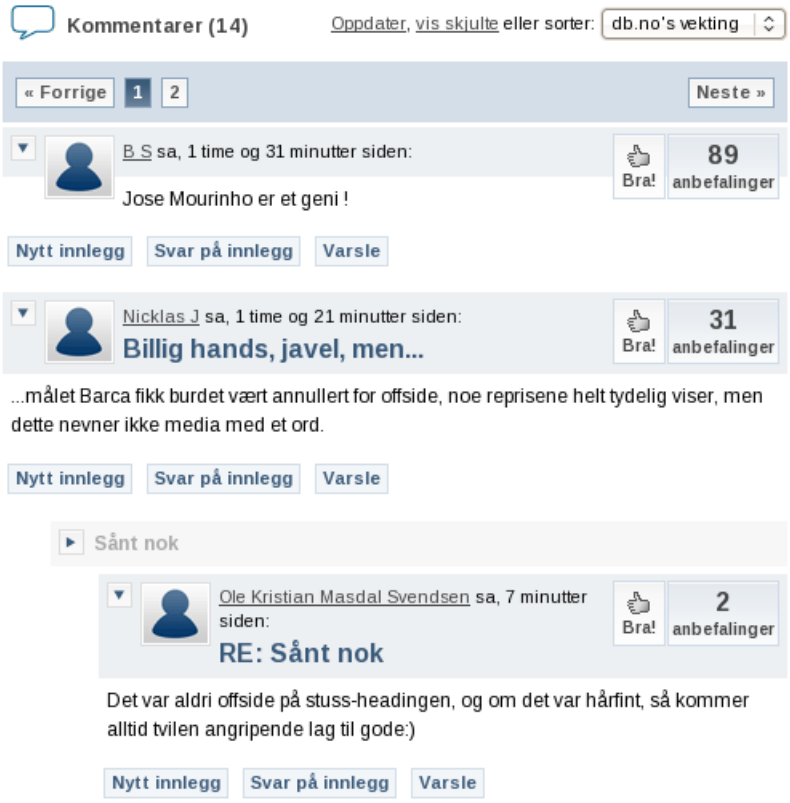
Da Internett begynte å få fotfeste i Norge på midten av 1990-tallet, florerte det med statiske nettsider. Dette var sider som ikke ble bearbeidet av serveren før den forespurte siden ble sendt til brukerens nettleser.

Etter hvert som tiden gikk utviklet teknologiene seg slik at de fleste utviklere kunne skape dynamiske nettsider, der innholdet ble preprosessert av vevtjeneren før den ble servert til brukeren. De fleste dynamiske vevsider bruker ulike valg gjort av brukeren for å kunne sortere og bearbeide informasjonen som returnes til brukeren. Eksempler på slike valg er filtrering av informasjon, menyvalg og sidenavigering.

I dag er det ikke bare interaksjon mellom system og bruker, men også kommunikasjon mellom brukere. Denne tanken er ikke ny, og allerede i 1980 ble Usenet tatt i bruk for første gang. Usenet er et verdensomspennende distribuert diskusjonssystem [73] som fremdeles er i bruk. Etter verdensveven fikk fotfeste ble det utviklet flere teknologier for kommunikasjon mellom brukere, som for eksempel e-post-lister (eng. *electronic mailing list*), nyhetsgrupper, nettforum, nettverksprotokollbaserte tjenester som Internet Relay Chat (IRC) [70] og direktemeldingsprogram som ICQ [42] og MSN Messenger [52]. Etter hvert ble det også mer og mer vanlig at brukerkommunikasjon kunne opptre direkte i nettleseren, enten ved at nettlesere hadde integrerte løsninger for de nevnte teknologier, eller ved å bruke HTML. Eksempler på dette er blogger, nettforum, og lignende. Nettet er i dag blitt et sosialt sted der det kan utveksles erfaringer, kunnskap, og der temaer kan diskuteres. [39]. I en blogg drøftes det ofte temaer som er av interesse for flere, og kommentarene til bloggposten er ofte vel så interessante som bloggposten i seg selv.

I dag har flere av de store norske aktørene på verdensveven lagt inn muligheten for interaksjon. Eksempler er *vg.no* [38] som har et eget diskusjonsforum der brukere kan diskutere

fritt, og *dagbladet.no* [11], som har lagt inn mulighet for å kommentere artikler. Ved å tillate interaksjon kan brukere komme med innspill. Figur 6.1 viser Dagbladets kommentarfeltløsning der brukerne diskuterer et emne. Dagbladet har også utvidet funksjonaliteten til kommentarfeltet ved at brukere kan stemme opp gode kommentarer, her illustrert ved en knapp med tommel-opp-ikon. Dette gjør at kommentarer med gode anmeldelser rangeres høyere, slik at det er lettere å finne de mest interessante kommentarene. Dagbladets løsning tillater også nøsting av kommentarer, slik at brukere kan kommentere en kommentar, såkalte trådete kommentarer.

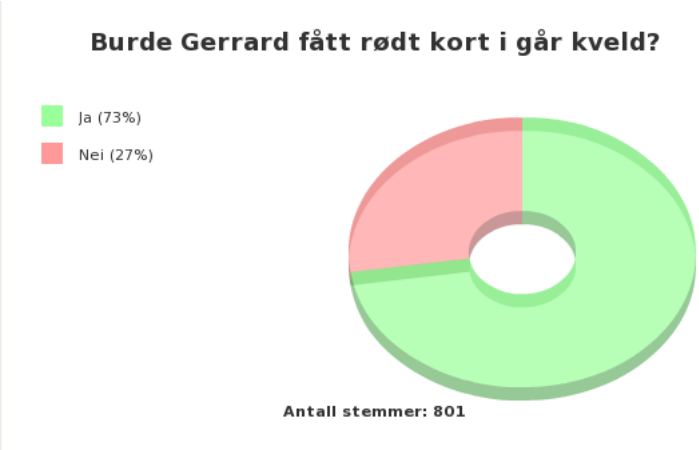


Figur 6.1: Dagbladets kommentarfelt

En annen vanlig interaksjonsmulighet er stemmegiving. Brukeren blir presentert med en problemstilling der du kan svare ved å velge en av flere valg. Dersom du sender inn din stemme blir valgresultatene vist fram. Brukeren blir da presentert med en oversikt som viser resultatene fra stemmegivingen. Hvert valg er da enten tilknyttet en prosentvis andel av stemmene, eller antall stemmer som er gitt på dette alternativet. Ofte er resultatene framvist i en graf slik at det er lett å se styrkeforholdet til de forskjellige valgalternativene. Det er vanlig å benytte et stolpediagram eller et kakediagram for dette. Figur 6.2 og 6.3 viser hvordan Dagbladets et eksempel på deres løsning for avstemninger. Ofte er avstemningene tilknyttet en artikkel slik at leserne skal få et innblikk i lesernes oppfatning av emnet.

I forhold til fjernundervisningen på UiB kan slike avstemninger være nyttig, ettersom det kan stilles spørsmål som om obligatoriske oppgaver er for vanskelig, eller om det er en passe arbeidsmengde i kurset. Slike avstemninger bør være anonyme, slik at ingen skal føle seg utilpass med å avgi svar.

Figur 6.2: Eksempel på avstemning hos dagbladet



Figur 6.3: Avstemningsresultater hos dagbladet

Termen Web 2.0, som dukket opp i 2004, blir brukt om vevapplikasjoner som fasiliterer interaktiv informasjonsdeling, interoperabilitet, brukerorintert design og samarbeid på verdensveven [77]. Eksempler på nettsider som følger denne karakteristikken er *facebook.com* [17], *twitter.com* [71] og *flickr.com* [18]. Dette er eksempler på nettsteder der brukerne står for innholdet på siden, og informasjon deles på tvers av brukere.

Twitter, Facebook og Flickr er blitt tre enormt populære nettsteder, mye på grunn av det brukergenererte innholdet. Med dette i tankene var det interessant å se om slik interaksjon kunne tilbys av DPG.

6.2 Interaksjon i DPG 2.1

For brukere med rollen *reader* i DPG 2.0, kan man oppgi hvilken *side* og hvilket *utsnitt* en vil ha servert, og dermed få rendret og servert denne siden. Uansett hvilke operasjoner en bruker velger å gjøre vil innholdet i systemet være uforandret. Ved å gi brukeren muligheten til å interagere med systemet kan brukeren selv påvirke systemets oppførsel. Dette gjøres ved å tillate brukere å supplere med inndata, slik at innholdet på siden endres utfra informasjon brukere har lagt inn.

For å illustrere interaksjonsmulighetene til DPG med den nye pluginarkitekturen er følgende plugins laget:

- Plugins for håndtering av kommentarfelt
- Avstemningsplugin

6.2.1 Kommentarfelt

For å realisere muligheten for kommentarfelt ble det laget to nye plugins for å håndtere dette, en plugin lagrer nye kommentarer, og en annen plugin henter ut kommentarer som er lagret. Disse plugins heter henholdsvis `SaveComment` og `LoadComments`.

6.2.1.1 Lagring av kommentarer

Kommentarene lagres ved hjelp av kontrakten `pluginDao`, som er beskrevet i kapittel 5. Et av de viktigste parameterene til metoden `saveResource()` er parameteren `filePath`. Her oppgis det en sti som forteller hvor kommentaren skal lagres. Denne stien må være unik, hvis ikke vil den overskrive det eksisterende innholdet.

I listing 6.1 vises de forskjellige elementene en `filePath` trenger for å kunne generere stien der innhold lagres.

Listing 6.1: Komponenter som trengs for å lagre en kommentar

```
<presentationsId>,
<pageId>,
<viewId>,
<commentId>
```

For å skille mellom de forskjellige elementene er liggestrek benyttet. Et konkret eksempel på en ferdig sti kan da være:

```
eLearning_resources_books_3
```

Dette betyr at kommentaren er tilknyttet presentasjonen `eLearning`, og kommentaren er lagt til siden `resources` på utsnittet `books`, og akkurat denne kommantaren er kommentar nummer fire, ettersom kommentarene begynner med verdien null.

Med en slik arkitektur er det en del begrensninger, blant annet kan man ikke ha mer enn ett kommentarfelt per utsnitt. Likevel anses det som svært usannsynlig at en vil trenge flere kommentarfelt per utsnitt.

For å finne verdien til det fjerde parameteren `commentId` kaller systemet metoden `doesResourceExist()` fra kontrakten `pluginDao` med verdiene fra listing 6.1. Verdien `commentId` begynner på verdien 0, og inkrementeres helt til metoden returnerer `false`. Dette betyr at det er funnet en ledig plass å lagre den nye kommentaren på.

Listing 6.2 viser koden som genererer strengen som lagres. Kommentaren blir datostemplet og brukernavn legges til slik at en vet hvem som har skrevet den.

Listing 6.2: Tekststreng som representerer en kommentar

```
private String generateCommentString(Map<String, Object> inputs) {
  Calendar calendar = Calendar.getInstance();
  String returnString = inputs.get("user").toString();
  returnString += " skrev den ";
  returnString += getNiceDate(calendar.getTime());
  returnString += ": ";
  returnString += inputs.get("comment").toString();
  return returnString;
}
```

Når prosessen er gjennomført vil kommentaren være lagret på en ledig plass, klar til å hentes ut av pluginet som henter lagrede kommentarer.

6.2.1.2 Lasting av lagrede kommentarer

For å hente ut alle kommentarer tilknyttet et utsnitt benyttes pluginet `LoadComments`. Dette pluginet får vite hvilken presentasjon, side og utsnitt den skal hente kommentarer fra. Det opprettes en tom resultatliste der kommentarene skal lagres. Deretter kalles metoden `doesResourceExist()` på en tekststreng generert med parameterne fra listing 6.1, parameteren `commentId` begynner med verdi 0. For å hente ut den lagrete kommentaren benyttes metoden `getPresentationResourceById()`, den returnerte kommentaren lagres i resultatlisten. Deretter inkrementeres `commentId` og metoden kalles på nytt. Denne prosessen gjentas helt til metoden returnerer `false`. Når dette inntreffer er alle lagrete kommentarer tilknyttet dette utsnittet hentet ut og lagret i resultatlisten.

Pluginet itererer så over resultatlisten og fester innholdet i kommentaren til rotelementet som pluginet er tilknyttet ved hjelp av metoden `getRootElement().addContent()` i klassen `PluginBean`. `XSLT`-transformasjonen transformerer innholdet til `XHTML` som kan

framvises på den endelig rendrerte siden. Figur 6.4 viser et utsnitt med et kommentarfelt. Det er to kommentarer lagt til i dette utsnittet, de nyeste kommentarene vises øverst. Bruksområder for kommentarfeltet kan være å legge inn utfyllende informasjon om sidens innhold eller å rapportere om feil og mangler på siden.

Figur 6.4: Kommentarfelt og to kommentarer

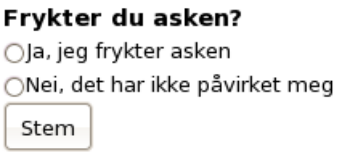
Svakhetene til denne løsningen er at det ikke er lett å kontrollere kommentarene etter at kommentarstrengen er generert i pluginet. For å forbedre dette kan det lages et ekstra lag mellom pluginet og `PluginDao`, som virker på samme måte som en *objekt-relasjonell avbildning* (eng. *object-relational mapping*) [3]. Dette gjør at en får tilgang til de forskjellige feltene en kommentar innholder, som igjen gjør at en kan utvide funksjonaliteten slik at man for eksempel kan sortere kommentarer på datoer eller andre felt. Andre eksempler på bruk er det å legge til slett-knapper på egne kommentarer.

6.2.2 Plugin for stemmegivning

Den enkleste versjonen av stemmegivninger består av at brukere kan stemme på ett alternativ. Mer avanserte løsninger er utformet som en flervalgsoppgave, der det kan krysses av for flere av svaralternativene. Fordi plugins er laget for å illustrere mulighetene til den nye

pluginarkitekturen, og på grunn av tidsrammer, benytter den implementerte stemmegivningsplugin `PollPlugin` det første, enklere alternativet.

Stemmegivningspluginet genererer først en tekststreng med et spørsmål, deretter legger den til et `HTML`-skjema med valgfritt antall svaralternativer. Til hvert svaralternativ er det tilknyttet en `HTML`-radioknapp som er tilknyttet svaralternativteksten. Det er også tilknyttet et inndatafelt av typen *submit*, som lager en knapp for å sende inn det valgte svaralternativ. Figur 6.5 viser et eksempel på hvordan dette ser ut i nettleseren.



Figur 6.5: Askestemmegivning

Brukeren velger et alternativ og sender inn skjemaet. Systemet prosesserer svaret, og presenterer antall svar for hvert alternativ. Det blir også generert et bilde av en graf som viser antall svar per valgalternativ i et kakediagram.

Pluginet defineres et unikt navn for selve stemmegivningen. Dette blir en unik id for hver stemmegivning og kan kalles for stemmegivningens *grunnstamme*. Slik kan en grunnstamme se ut:

```
poll_ash
```

For å generere alternativer legges alternativet til på slutten av grunnstammen, hvis svaralternativet er *yes* vil resultatet bli slik:

```
poll_ash_yes
```

I denne stien lagres et heltallsverdi, som representerer hvor mange stemmer dette svaralternativet har fått. For å forhindre at en bruker stemmer flere ganger på samme stemmegivning lagres også brukernavnet etter grunnstammen. Verdien her settes til svaralternativet brukeren valgte. Hvis brukeren heter *unclebob* vil stien bli slik:

```
poll_ash_unclebob
```

Hvis det skal lagres et nytt resultat sjekkes det om ressursen med denne filstien finnes. Hvis ressursen eksisterer, hentes verdien lagret under stien `poll_ash_`*alternativ* ut, verdien

gjøres om til et heltall, inkrementeres og lagres tilbake på samme sted. Hvis den ikke eksisterer vil verdien settes til 1.

Når brukeren sender inn skjemaet blir valget prosessert og brukeren sendes tilbake til siden der stemmegivningen befant seg. Da vil resultatene vises istedenfor stemmegivningen. Alle svaralternativene er lagret i en liste, og resultatene hentes ut ved at pluginet har kontroll over hvilke svaralternativer som er knyttet til en stemmegivning, og kan derfor hente ute antall svar per alternativ ved å hente ressurser fra riktig sti. Tilslutt ender man opp med en nøkkelverditabell der nøkkelen er alternativet og verdien er antall treff. JDOM-elementer knyttes til rotelementet på lik måte som i delkapittel 6.2.1.2. Et eksempel på en slik nøkkelverditabell er vist i tabell 6.1. Første kolonnen viser svaralternativstien, og andre kolonne er antall stemmer på dette alternativet.

Tabell 6.1: Nøkkelverditabell med stemmegivningsalternativer og stemmer

Svaralternativ	Antall stemmer
poll_ash_ja	2
poll_ash_nei	4
poll_ash_kanskje	3

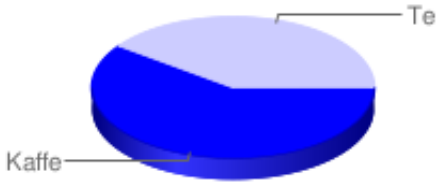
Stemmegivningsresultatene brukes også til å generere og lagre en bildesti med en lenke som peker til en adresse som Googles tilbyr. Google Chart Tools [41] tilbyr visualisering av data i form av diverse grafer, enten ved hjelp av et tilpasset JavaScript, eller ved å benytte et API kalt Google Chart API. Det er sistnevnte som er brukt i `PollPlugin`.

Google Chart API returnerer en `PNG`-bildefil som respons til en `URL GET` eller `POST`-forespørsel. Med i forespørselen legges dataen som skal visualiseres, og diverse annen informasjon, som kolonnenavn, bildestørrelse og hvilken type graf som skal returneres. Listing 6.3 viser et eksempel på en URL som genererer en graf. Linje 2 forteller hvilken type graf som ønskes, *p3* sier at det ønskes et kakediagram i 3D. Linje 3 er størrelsen på grafen i antall piksler, linje 4 er en beskrivelse av dataene. T-en betyr at den bruker et enkelt tekstformat med flytverdier fra 0-100. Det finnes også tilgjengelige format. Linje 5 gir en etikett til de forskjellige bestanddelene i grafen og linje 6 sier hvilken farge grafen skal ha som utgangspunkt, hvis grunnfargen er blå vil alle kakestykkene ha en blålig farge.

Listing 6.3: URL til Google Chart API

```
1 http://chart.apis.google.com/chart?
2   cht=p3&
3   chs=250x100&
4   chd=t:60,40&
5   chl=Kaffe|Te&
6   chco=0000FF
```

Resultatet av forespørselen fra listing 6.3 er vist på figur 6.6



Figur 6.6: Generert graf fra innholdet i kallet fra 6.3

Figur 6.7 viser et eksempel på resultat fra stemmegivning på figur 6.5. Først blir brukeren presentert med hvilket svaralternativ som ble valgt, deretter er de forskjellige alternativene listet opp med hvor mange som ble valgt av de forskjellige alternativene. Tilslutt vises Google Chart-grafen med sin visuelle fremstilling av resultatene.

Figur 6.7: Askeresultater

For at det skal være lettere å tilpasse pluginet, kan pluginet konfigureres i konfigurasjonsfilen `pluginConfig.xml`. Ved å legge konfigurasjonen her trengs det ikke kunnskap om koden til pluginet for å tilpasse pluginet. Pluginet `PollPlugin` trenger fire parametere for å fungere:

1. `pollName` - Unikt navn for stemmegivningen
2. `pollHeader` - Overskrift til stemmegivningen
3. `pollChoices` - Kort navn på et alternativ, for eksempel *"ja"*
4. `pollChoicesVerbose` - Utbroderende forklaring av alternativet, for eksempel *"nei, regnet påvirker ikke humøret mitt"*

For å sørge for at pluginet er riktig konfigurert gir pluginet feilmeldinger dersom et parameter ikke er satt eller er feil konfigurert. Feilmeldingene blir knyttet til rotnoden og vil derfor være synlige der pluginet vises i den endelige rendrerte siden. Alle fire parameterene er påkrevd, og spesifikke feilmeldinger om hvilke plugins som mangler blir gitt dersom dette inntreffer. For at det skal være mulig å oppgi flere alternativer i parameteren `pollChoices` og parameteren `pollChoicesVerbose` skiller en forskjellige alternativer fra hverandre ved å bruke tegnet `|`. Antall alternativer til parameteren `pollChoices` og parameteren `pollChoicesVerbose` må være lik, hvis ikke vil det gis en feilmelding.

Et eksempel på hvordan en slik plugins er konfigurert er vist i listing 6.4. Pluginet konfigurerer stemmegivningen fra figur 6.5 og 6.7.

Listing 6.4: Konfigurasjon av PollPlugin

```
<plugin-config id="poll" plugin-name="PollPlugin">
  <param name="pollName">ash</param>
  <param name="pollHeader">Frykter du asken?</param>
  <param name="pollChoices">Ja|Nei</param>
  <param name="pollChoicesVerbose">
    Ja, jeg frykter asken|
    Nei, det har ikke pavirket meg
  </param>
</plugin-config>
```

6.2.3 Oppsummering

Løsningene som er presentert her er et *proof-of-concept*. Det er bevist at det mulig å lage interaktive plugins i DPG 2.1. Begge plugins kan videreutvikles og forbedres. For forslag om om videre arbeid, se delkapittel 10.3

Kapittel 7

Ny presentasjonsmønsterspesifikasjon

Dette kapittelet er skrevet i fellesskap av Peder Lång Skeidsvoll og Tobias Rusås Olsen. Kapittelet inngår i begges masteroppgaver. Konklusjoner i dette kapittelet er gjort på grunnlag av samtale mellom de to forfatterene. Det er også kommet innspill fra veiledere under ukentlige møter.

Dette kapittelet beskriver endringer og forbedringer gjort i presentasjonsmønsterspesifikasjonen, som definerer elementene i et presentasjonsmønster.

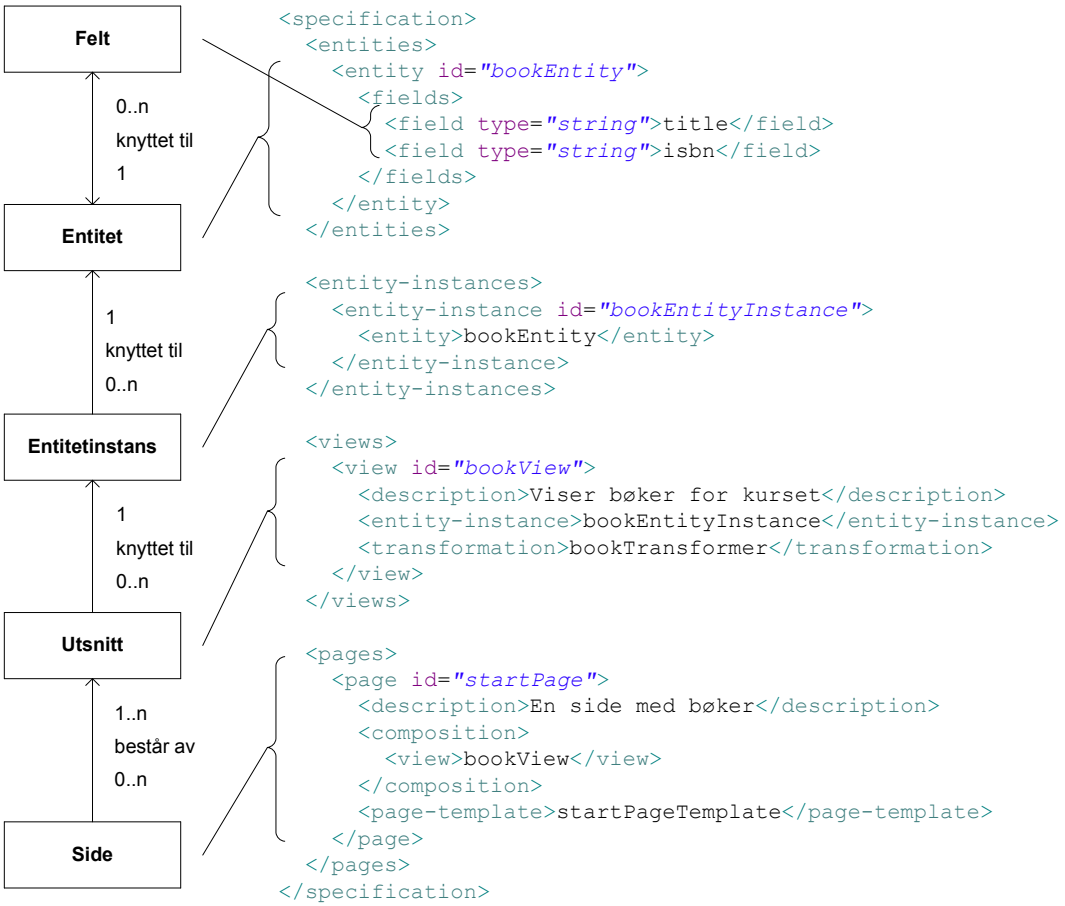
7.1 Bakgrunn

Før arbeidet med DPG 2.0 startet, gikk masterstudentene Rossini og Liberati [72] gjennom presentasjonsmønsterspesifikasjonen til DPG 1.0 og foreslo endringer av denne. Dette arbeidet ble så igjen revidert, i all hovedsak av masterstudenten Sebak [63], noe som resulterte i presentasjonsmønsterspesifikasjonen som benyttes i DPG 2.0.

Figur 7.1 viser hvordan de forskjellige komponentene i presentasjonsmønsterspesifikasjonen til DPG 2.0 og sammenhengen mellom dem. For eksempel viser den at en *side* (eng. *page*) består av en eller flere *utsnitt* (eng. *view*), og et et *utsnitt* er å finne på null eller flere *sider*. Det er ikke laget noen tekniske begrensninger på at et utsnitt må være tilknyttet en side, men et utsnitt uten en slik kobling vil ikke ha noen nytteverdi.

7.2 Bruk av plugins i presentasjonsmønstre

DPG 2.0 har to forskjellige måter å definere datatypen til et felt: Innebygde typer eller plugins. Det finnes i DPG 2.0 fem forskjellige innebygde typer: `string`, `date`, `entity-list`,



Figur 7.1: Presentasjonsmønsterspesifikasjon til DPG 2.0.

`file` og `date`. Listing 7.1 viser hvordan en innebygd type defineres i DPG 2.0. I dette eksempelet finnes det et felt som heter `boktittel` som er av typen `string`.

Listing 7.1: Eksempel på et felt av typen `string`.

```
<field type="string">
  boktittel
</field>
```

Dersom et felt skal håndteres av en plugin i stedet for en innebygd type, må feltet defineres som i listing 7.2. I dette eksempelet er typen til feltet `boktittel` definert som `plugin` gjennom `type`-attributtet.

Dette forteller DPG 2.0 at systemet nå skal bruke en plugin, og gjør samtidig at DPG 2.0 forventer et attributt med navn `pluginConfig`. Dette attributtet forteller DPG at feltet skal bruke pluginet `bokPlugin`.

Listing 7.2: Eksempel på et felt som bruker en plugin.

```
<field type="plugin" pluginConfig="bokPlugin">
  boktittel
</field>
```

Dette er en uheldig løsning fordi plugins ikke bør være et spesialtilfelle. En bedre løsning er å la plugins være den eneste måten å definere typen til et felt. Ved å implementere denne løsningen vil DPG få en generell mekanisme for håndtering av felttyper. Fordelen med denne løsningen er:

- Bruken av plugins fremmes slik at det blir mer aktuelt for presentasjonsmønsterutviklere å benytte seg av plugins.
- Filen `pattern.xml` vil bli enklere å lese og skrive.
- Kodebasen til DPG kortes ned siden den ikke trenger å behandle spesialtilfeller.

I DPG 2.1 er de innebygde typene forsvunnet, og alt er overlatt til plugins. I den nye spesifikasjon er det semantiske innholdet til attributtet `type` endret. Attributtet `type` angir nå hvilken plugin feltet skal håndteres av. Slik ser det ut hvis det lages et felt med den nye spesifikasjonen, der navnet på feltet er `feltnavn`, og `pluginNavn` er navnet på pluginet som håndterer det:

```
<field type="pluginNavn">feltnavn</field>
```

Denne endringen gjør det enklere å bruke plugins i et presentasjonsmønster ved at det ikke lenger er et spesialtilfelle, men den vanlige fremgangsmåten for å definere typen til et felt. For å gjøre overgangen lettere er det utviklet plugins som erstatter de tidligere innebygde typene. Disse er:

- `date`
- `file`
- `string`
- `xhtml`

Den eneste innebygde typen som ikke er implementert er lister, og det vil i det neste delkapittelet diskuteres løsninger for dette.

7.3 Lister

I DPG 2.0 er lister definert ved å sette attributten `type` til `entity-list` i et felt. Listing 7.3 viser et eksempel på en slik listedeklarasjon. Attributten `ref-id` gir DPG beskjed om hvilken entitet det skal lages liste av.

Listing 7.3: Et felt defineres som en entitetsliste.

```
<entity id="softwareEntity">
  <fields>
    <field type="entity-list" ref-id="url">URLs</field>
  </fields>
</entity>
```

I avsnitt 7.2 ble de innebygde typene til DPG fjernet, og det trengs derfor en erstatning for lister. Lister er en spesiell felttype fordi den inneholder andre entiteter. Målet er at også lister skal omfavnes av den generelle mekanismen for felttyper.

Det er utarbeidet tre forslag for hvordan lister skal håndteres i DPG 2.1, og disse vil nå bli presentert.

7.3.1 Lister behandles som i DPG 2.0

Dette forslaget innebærer å la lister være uforandret. Fordelen er at mønstre som allerede er utviklet kan brukes uten modifikasjon, noe som gir en viss bakoverkompabilitet. Løsningen i DPG 2.0 er i tillegg en velprøvd metode som har vist seg å fungere i praksis. Ulempen til forslaget er at målet om uniform håndtering av felt ikke tilfredsstilles, og forslaget er derfor uaktuelt dersom det finnes andre løsninger som tilfredsstiller dette målet.

7.3.2 Initialisering av lister i entititetsinstansen

Det andre forslaget er å definere entitetslister på entitetsinstansnivå. Dette betyr at lister definerers på nivået over entitet (se figur 7.1).

Listing 7.4 viser et eksempel på hvordan en slik løsning kan se ut. Her defineres en todimensjonal liste av entiteten `softwareEntity`.

Listing 7.4: Lister definiert i entitetsinstanser.

```
<entity-instances>
  <!-- Her defineres en todimensjonal liste -->
  <entity-instance id="urlEntityInstance" type="[][]">
    <entity>softwareEntity</entity>
  </entity-instance>
</entity-instances>

<entity id="softwareEntity">
  <fields>
    <field type="string">URLs</field>
  </fields>
</entity>
```

Listen er definert ved hjelp av []-paranteser. Dette er samme syntaks som blir brukt til å definere *tabeller* (eng. *arrays*) i programmeringsspråkene Java og C. I dette tilfellet er listen todimensjonal.

Svakheten til denne løsningen er at det er umulig å definere lister innenfor andre lister. Dette er fordi entiteter ikke kan inneholde entitetsinstanser.

For eksempel er det i Kursmønsteret viktig å kunne ha flere oppgaver på en uke. Dette må løses ved at det er en liste med uke-entiteter, der hver uke-entitet inneholder en liste med oppgave-entiteter. Figur 7.2 viser et diagram over eksempelet. Diagrammet har en liste med uker, der hver uke inneholder en liste med oppgaver. Dette kan ikke gjennomføres med dette forslaget. En uke-entitet måtte i så tilfelle hatt en referanse til en entitetsinstans, noe som ikke er mulig fordi lister defineres, i dette forslaget, på entitetsinstansnivå.

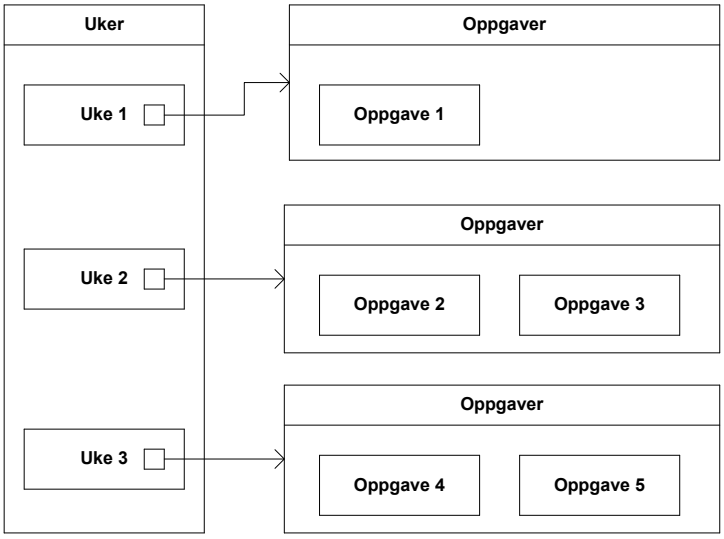
7.3.3 Lister som plugins

Det siste forslaget som er utarbeidet er å la lister bli behandlet av plugins. Det vil da være få forskjeller i presentasjonsmønsteret fra hvordan det ble skrevet i DPG 2.0. Forslaget vil kreve en del endringer i koden. Listing 7.5 viser hvordan en entitet som inneholder en liste i ett av feltene sine vil se ut hvis dette forslaget tas i bruk. Her defineres en liste av entiteten `url`.

Listing 7.5: Lister definert med plugins.

```
<entity id="softwareEntity">

  <fields>
    <field type="stringPlugin">title</field>
    <!-- Her blir listen med url-er definert -->
```



Figur 7.2: Liste med uker som inneholder lister med oppgaver.

```
    <field type="list" entity="url">urls</field>
  </fields>

</entity>

<entity id="url">
  <fields>
  <field type="string">title</field>
  <field type="string">address</field>
  </fields>
</entity>
```

Denne fremgangsmåten har flere fordeler. Blant annet er det mulig å utvikle flere listeplugins som for eksempel kan generere RSS-filer basert på innholdet i listen.

7.3.3.1 Konklusjon

Det er ikke nødvendig å ta hensyn til bakoverkompatiblitet. Det er avdekket at presentasjonsmønsterspesifikasjonen i DPG 2.0 har flere alvorlige svakheter, og derfor bør disse tas hånd om før det utvikles flere presentasjonsmønstre. Denne vurderingen gjør at det forslaget om å beholde listehåndteringen fra DPG 2.0 ikke er aktuelt.

Forslaget som omhandler instansieringen av lister på instansnivå, ser ved første øyekast ut som et godt forslag. Ved nærmere undersøkelse viser det seg at det vil redusere funksjonaliteten til systemet. Dette er fordi lister i dette forslaget ikke kan inneholde andre lister, som påpekt i 7.3.2. Derfor er heller ikke dette et tilfredsstillende alternativ.

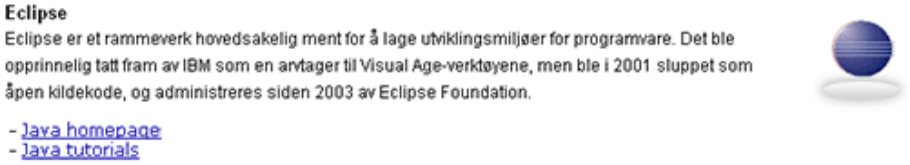
Det siste forslaget, der liste ble behandlet av plugins, utvider funksjonaliteten til lister ved å gjøre dem mer fleksible. Det vil i tillegg føre til at målet om en felles mekanisme for behandling av typer blir overholdt. På grunnlag av dette er det bestemt å utvikle en pluginløsning for bruk av lister.

7.4 Fjerning av `fields`-elementet

I presentasjonsmønsterspesifikasjonen til DPG 2.0 er elementet `fields` påkrevd. Dette elementet innholder `field`-elementene til en entitet. Listing 7.6 viser et eksempel på hvordan `fields` elementet blir brukt i DPG 2.0. Siden det ikke er mulig å legge inn andre elementer enn `field` i en entitet, er det besluttet å fjerne dette elementet. Elementet hadde ingen funksjon, og gjorde filen `pattern.xml` mer komplisert. Endringen gjør det lettere å lese og skrive presentasjonsmønstre.

7.5 Resultat

Figur 7.3 et et skjermbilde som viser hvordan en entitet kan fremstå når den er ferdig rendret. Navnet på entiteten er `Programvare`. Entiteten innholder en tittel, en beskrivelse, et bilde, og en liste med lenker. Lenkene består av en adresse og et lenkenavn.



Figur 7.3: Et utsnitt som inneholder en entitet

Listing 7.6 viser hvordan disse entitetene blir definert i filen `pattern.xml` i DPG 2.0. Hvert felt må definere en type, og dersom det er valgt type `plugin` må det i tillegg defineres en attributt som forteller hvilken type plugin dette feltet skal knyttes til.

Listing 7.6: Definisjon av entiteter i DPG 2.0.

```
<entities>
  <entity id="softwareEntity">
    <fields>
      <field type="string">title</field>
      <field type="xhtml">description</field>
      <field type="plugin" pluginConfig="imagePlugin">image</field>
      <field type="entity-list" ref-id="url">URLs</field>
    </fields>
  </entity>
```

```
    <entity id="url">
      <fields>
        <field type="string">address</field>
        <field type="string">name</field>
      </fields>
    </entity>
  </entities>
```

Listing 7.7 viser hvordan de samme entitetene som i listing 7.6 defineres med den nye presentasjonsmønsterspesifikasjonen til DPG 2.1. Alle feltene blir definert som plugins, inkludert lister. Koden er mer kompakt, og er både lettere å skrive og lese. Ved å indikere forskjellen mellom presentasjonsmønsterspesifikasjonene i DPG 2.0 og DPG 2.1 har typen til feltene i sistnevnte spesifikasjon fått postfikset `Plugin` selv om alle felter uansett er håndtert av plugins.

Listing 7.7: Definisjon av entiteter i DPG 2.1.

```
<entities>
 <entity id="softwareEntity">
   <field type="stringPlugin">title</field>
   <field type="xhtmlPlugin">description</field>
   <field type="imagePlugin">image</field>
   <field entity="url" type="listePlugin">urls</field>
 </entity>

 <entity id="url">
   <field type="stringPlugin">address</field>
   <field type="stringPlugin">name</field>
 </entity>
</entities>
```

Kapittel 8

Søk

8.1 Kort om søk

Søk er en betegnelse som involverer maskinprosesser, menneskelige prosesser, menneskelige tanker og til og med menneskelige følelser [5]. Søk handler om å finne den informasjonen som man ser etter.

Mengden av informasjon på verdensveven er i dag stadig økende, og uten gode søketeknologier er det lett å drukne i den enorme mengden av informasjon. Derfor er søk en kritisk prosess for nettsider med store innholsdmengder. I løpet av de siste årene er søk blitt en så kjent prosess at de fleste sider med en viss mengde informasjon tilbyr søkefunksjonalitet. Nettsidene kan enten benytte interne løsninger for søk, eller løsninger der eksterne aktører håndterer søkingen for dem. Et eksempel på en slik aktør er Google [44], som tilbyr søk på en enkelt nettside ved å benytte nøkkelordet *site* etterfulgt av adressen til nettsiden det skal søkes i.

DPG 2.0 har ikke slike søkemuligheter, dette er en stor mangel dersom presentasjoner inneholder mye informasjon. Ferdige løsninger som den Google tilbyr vil dessverre ikke fungere i DPG. Dette er fordi DPG krever et brukere er innlogget før innhold aksesseres. Det er heller ikke ønskelig å søke i flere presentasjoner på en gang, fordi innholdet i hver presentasjon er unik.

Søkeprosessen kan deles inn i tre hovedområder:

1. Indeksering av dokumenter
2. Tekstanalyse av innsendte dokumenter
3. Søking etter relevante dokumenter

8.1.1 Indeksering av dokumenter

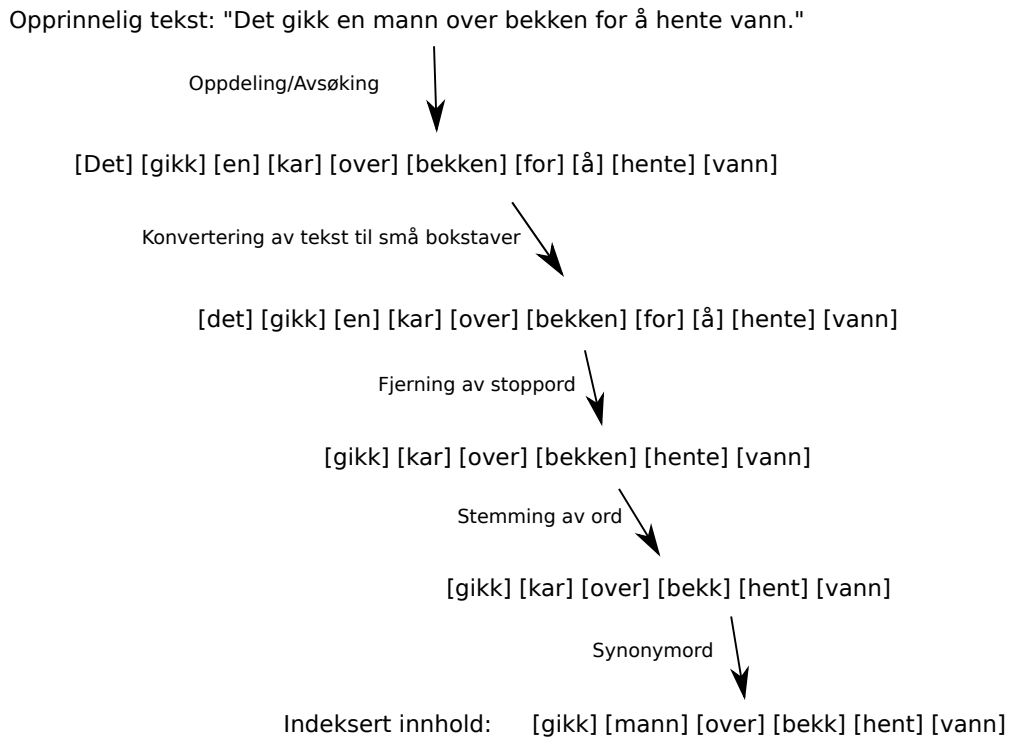
I søkesammenheng betyr indeksering det å samle, tolke og lagre data for å kunne fort og effektivt finne igjen den lagrede informasjonen. Indekseringen handler om å hente ut tekstlig informasjon fra forskjellige typer data der tekstlig informasjon kan hentes ut. Målet med å lage en indeks er å hjelpe søkemotoren til å finne dokumenter som er relevant til en søketekst. Dersom en ikke hadde indekser, måtte det søkes gjennom hvert eneste dokument for å finne informasjon, noe som ville vært en svært tungvint og ineffektiv løsning. Ved hjelp av gode indekser kan informasjon lagret blant flere millioner dokumenter gjenfinnes på få sekunder.

8.1.2 Tekstanalyse

Når dokumentene er sendt inn til indeksering gjøres det en tekstanalyse på innholdet. Det gjøres en rekke forbedringer og optimaliseringer for at indeksene skal være så presis som mulig i forhold til det semantiske innholdet i dem. Det finnes en rekke steg som er vanlig å følge når dokumentene optimaliseres. Målet er å ekstrahere kjerneinnholdet i dokumentet. Her er et par av de mest vanlige teknikkene som brukes for å oppnå dette:

- Oppdeling av setning til enkeltstående ord
- Konvertere all tekst til liten skrift.
- Fjerne ord som ikke gir betydning for innholdet i teksten, såkalte *stopp-ord* . Eksempler på dette er *og*, *dette*, *i*, *ved* og lignende. I noen tilfeller er det likevel viktig å ta vare på stoppord, fordi det kan utelate viktige søkefraser. Eksempler på søkefraser der en ikke kan utelukke stoppord er bandnavn som *The Who* eller *Take That*. Dette må tas hensyn til av søkemotoren.
- *Stemming* av ord, det vil si at ordene konverteres slik kun stammen av ordene beholdes. Et eksempel er at *leser* blir til *les* og *hoppet* blir til *hopp*. Dette vil gjøre at søk på *leste* vil være lik ordet *leser*, selvom ordene er skrevet forskjellig. Her er det viktig å bemerke at forskjellige språk har forskjellig stemming, og tekstanalysereren må ta hensyn til dette.
- *Synonymord* er en annen teknikk. Dette gjør at ord med lik mening, men forskjellig skrivemåte tolkes likt. Et eksempel på dette er *"boks"* som har samme betydning som ordet *"eske"*.

Figur 8.1 viser et eksempel på hvordan de forskjellige filtrene virker på en setning. Teksten ved siden av pilene sier hvilken teknikk som er benyttet i det steget.



Figur 8.1: Eksempel på hvordan filtrering fungerer

8.1.3 Søking

Et søk foregår ved at det rettes en forespørsel til søkemotoren som da gjør et søk i indeksen, med mål om å hente ut relevante dokumenter fra indeksen. For å få best mulig resultat må en bestemme seg for hvilke søketeknikker en vil benytte seg av. I tillegg er det viktig å tenke på brukervennlighet, det må tas hensyn til hvordan brukere er vant til å benytte søk.

8.1.3.1 Hvor bra er et søkeresultat?

For å forstå hvor bra et søkeresultat må de to begrepene *presisjon* (eng: *precision*) og *tilbakekall* (eng: *recall*) forklares.

Presisjon er andelen av relevante dokumenter i forhold til det totale antall dokumenter funnet. [5]

Presisjon regnes ut ved ved å bruke følgende formel:

$$presisjon = \frac{relevantReturnert}{totalReturnert} \tag{8.1}$$

relevantReturnert er antall relevante dokumenter returnert, og *totalReturnert* er det totale antall dokumenter returnert.

Hvis du får et resultatsett på 10 dokumenter, og 5 av dokumentene er relevante, er presisjonen på 0,5.

Tilbakekall er antallet relevante dokumenter i svarsettet i forhold til den totale mengden relevante dokumenter du søker i [5]. For å finnes tilbakekallverdien benyttes følgende formel:

$$tilbakekall = \frac{funnetRelevante}{totalRelevante} \tag{8.2}$$

funnetRelevante er antall relevante dokumenter returnert og *totalRelevante* er det totale antallet returnerte dokumenter i indeksen.

Dersom du har lav tilbakekall betyr det at det returneres få relevante dokumenter i forhold til antallet relevante dokumenter i dokumentsettet. Hvis det hentes ut 6 relevante dokumenter, og det er totalt 18 relevante dokumenter i dokumentsettet betyr det at tilbakekallet er på 0.33.

Både presisjon og tilbakekall gir et tall med verdi mellom 0 og 1, der 1 er den beste verdien.

Tilbakekall og presisjon er ofte to motstridende termer, en høy presisjon vil ofte føre til en lav tilbakekall og omvendt [13]. Høy presisjon kan oppnås ved å bruke `AND`-operatoren mellom søkeord, da må eksakte ord og fraser matche. Høy tilbakekall kan oppnås ved å bruke `OR`-operatoren, og benytte seg av *tilnærmingssøk* (eng: *fuzzy-søk*), ordstemming og synonymordlister. Presisjon og tilbakekall påvirkes altså av analyseringen og optimaliseringen av indeksene.

8.1.3.2 Presentasjon av søkeresultater

De fleste internettbrukere er blitt vant med å søke etter informasjon på internett. Selskapet Google har gjort så stor suksess med sine søkeløsninger at navnet deres er blitt et verb synonymt med ordet *å søke*. En av fordelene til Google i forhold til flere av sine konkurrenter var ikke bare at de var meget gode på å finne relevante dokumenter, men også fordi de

hadde et lettfattelig design og en enkel tilnærming [69]. Figur 8.2 viser hvordan Google presenterer sitt søk. Det er en enkel løsning der brukeren kun trenger å forholde seg til en enkel søkeboks.

Figur 8.2: Enkelt søk hos Google.com

De fleste søkemotorer har også støtte for avansert søk. Figur 8.3 viser hvordan Google sin løsning ser ut. Her kan det føres opp hvilke ord som ikke skal inkluderes, hvilke språk det ønskes resultater fra, søk i et spesifikt domene, antall søketreff det ønskes per side og mye mer.

Når søket er gjennomført blir resultatdokumentene vist. Søkeresultatene returnerer ofte i en listeform der de mest relevante treffene er presentert først. Hvert søketreff gir en link til treffdokumentet og kort informasjon om innholdet rundt søkeordet, slik at brukeren har større grunnlag til å vurdere om siden er relevant i forhold til det man søker etter.

8.2 Valg av teknologier

For å bestemme seg for en teknologi er det viktig å undersøke DPG og dens arkitektur. DPG håndterer en eller flere presentasjoner der hver presentasjon har unikt innhold. Hver bruker har tilgang til et subsett av alle presentasjoner. Innholdet i hver presentasjon kan være unik selvom strukturen kan være lik. På grunnlag av dette føles det unaturlig å kunne søke i andre presentasjoner enn den man befinner seg i.

Figur 8.3: Avansert søk hos Google.com

Det ble gjennomført en mulighetsstudie for å undersøke hvilke søketeknologier som var aktuell. Tidlig i prosessen med valg av teknologi ble Apache Lucene [20] nevnt som en sentral aktør når det gjelder søking med Java. Lucene er et informasjonsgjenfinningsbibliotek som er skrevet i Java, og det var mulig å oppdrive flere prosjekter som baserte seg på nettopp Lucene.

Ettersom DPG benytter Apache Jackrabbit [19] for lagring av innhold i forhold til JCR-standarden [58], og Jackrabbit har innebygget støtte for Lucene, var dette noe som måtte undersøkes nærmere. Andre alternativer som var aktuelle, var søkeserveren Apache Solr [24] og rammeverket Compass [60], og web-crawleren Apache Nutch [21], alle baserer seg på Lucene. Andre alternative som Sphinx [45], Xapian [61] og andre søkebibliotek/rammeverk ble utelukket fordi de baserte seg på C++, mens ønsket vårt var å fortsette å benytte oss av løsninger med Java, ettersom DPG er utviklet i Java.

8.2.1 Lucene og Jackrabbit

Den første innfallsvinkelen gikk ut på å bruke den innebygde støtten for Lucene i Jackrabbit. Dette foregår ved at man konfigurerer klassen `SearchIndex` i Jackrabbit sin konfigurasjonsfil *repository.xml*. Jackrabbits implementasjon bygger på standardimplementasjo-

nen til Lucene [28]. Man knytter denne klassen opp mot diverse *teksthentere* (eng: *extractors*), for at den skal klare å hente ut innhold fra forskjellige filtyper.

Dessverre finnes det flere ulemper med denne løsningen, som gjør at den ikke er aktuell. Informasjonen i presentasjoner ligger ikke utelukkende lagret i JCR-oppbevaringsstedet, også i både plugins og velocitymaler før alt tilslutt settes sammen til lesbar tekst. Denne løsningen vil også ha en sterk tilknytning til Jackrabbit-implementasjonen, noe som er et stort problem dersom en ønsker å bytte ut måten man lagrer innhold på. I tillegg er graden av dokumentasjon rundt indekseringen og Jackrabbit svært lav, og blir derfor vanskelig å både utvikle og tilpasse.

8.2.2 Lucene

En annen løsning var å bare bruke Lucene, uten å benytte det høyere abstraksjonslaget tilbudt av andre søketeknologier. Dette overlater mye av arbeidet til utvikleren, ettersom det kun er den mest sentrale funksjonaliteten som er tilgjengelig.

Lucene tilbyr to hovedoperasjoner, indeksering av dokumenter og søking i indeksen, se figur 8.4.

Indeksering av dokumenter skjer ved at man har et datasett med tekstlig innhold som skal indekseres. Først defineres forskjellige inndatafelt basert på formen til innholdet i datasettet, deretter opprettes det et tomt dokument og man fyller inn de forskjellige inndatafeltene med informasjon hentet fra datasettet. For hvert inndatafelt må en definere om inndatafeltet skal lagres og indekseres.

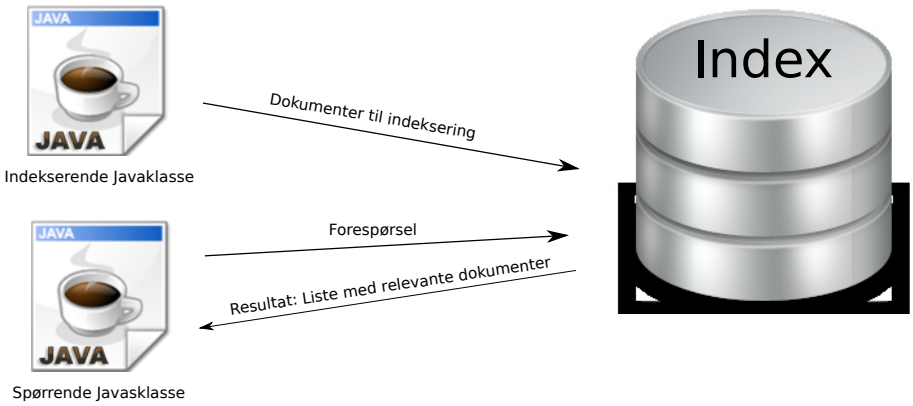
Figur 8.1 viser opprettelsen av et nytt dokument der det blir lagt til et nytt inndatafelt som skal både lagres og indekseres.

Listing 8.1: Eksempel på inndatafelt

```
Document doc = new Document();
doc.add(new Field("id", urlToResource, Field.Store.YES, Field.Index.
    ANALYZED));
```

Denne prosessen gjentas for hvert datasett, og tilslutt har man en liste med dokumenter. Denne listen sendes inn til et objekt av typen `IndexWriter`, som har en tilkobling til et analyseringsobjekt og en referanse til selve indeksen. Ut fra verdiene i analyseringsobjektet vil feltene i dokumentene kjøres gjennom diverse filtre. Deretter sendes dokumentene til indeksen.

For å søke i indeksen opprettes det et objekt av en søkerklasse og det sendes et forespørselsobjekt til søkerobjektet. Resultatet av metodekallet som utfører i søket er en liste med de mest relevante dokumentene.



Figur 8.4: Indeksering og uthenting av informasjon i Lucene.

Lucene er en mulig løsning, fordi den tilbyr alt som trengs for å kunne søke og gjenfinne informasjon i presentasjoner og vil derfor være et aktuelt alternativ.

For mer informasjon om Lucene, anbefales boken Lucene in Action [59] og artikkelen Using Apache Lucene to search text [66].

8.2.3 Web-crawling ved hjelp av Apache Nutch

Web-crawling fungerer ved at det angis en startside i form av en nettside, web-crawleren indekserer innhold på siden, besøker alle lenker og gjentar prosessen. Dette utfører altså på lik måte som et bredde-først-søk. En kan angi en viss søkedybde i web-crawlerprogrammet slik at programmet gir seg i det den har nådd den gitte dybden eller at den har besøkt alle lenker. Det går også an å definere at programmet skal holde seg ett eller flere domener.

Apache Nutch [21] er en slik web-crawler. Den er enkel å sette opp, og ved å gi den en enkel kommando kan den indeksere hele nettsider. Den baserer seg på Lucene. For at en slik web-crawler skal fungere i DPG må den ha tilgang til en presentasjon, da kan web-crawleren automatisk gå gjennom de tilgjengelig sidene og indeksere det den kommer over.

Dersom det viser seg at det er mulig å bruke Apache Nutch sammen med DPG uten at løsningen blir tung og dårlig, vil dette være en svært aktuell og enkel løsning.

8.2.4 Rammeverket Compass

Compass [60] er et rammeverk som også baserer seg på Lucene. Den tilbyr et lettfattelig API-lag for *objektrelasjonelle avbildninger* (eng: *object-relational mapping*) for å indeksere innhold. Den har også innebygget støtte for å automatisk speile endringer gjort i den objektrelasjonelle avbildningen til søkemotoren, slik at indeksene alltid er oppdaterte. Rammeverket integrerer svært godt med Spring.

På grunn av den gode koblingen til Spring, og at den tilbyr mer enn bare Lucene, er Compass også et alternativ som må vurderes.

8.2.5 Søkeserveren Solr

Solr [24] er en søkeserver med åpen kildekode og baserer seg også på Lucene. Den tilbyr en rekke vevtjenester, og du kan lett indeksere dokumenter til serveren ved å sende XML over HTTP. For å få tak i søkeresultater gjøres det en forespørsel til serveren og resultatet av spørringen blir returnert i programmeringsspråkuavhengig format. Solr kjører i en hvilken som helst Java servletkontainer. Solrserveren har gode og fleksible konfigurasjonsmuligheter. En fordel med Solr i forhold til DPG, er at ettersom det er en ekstern server, så vil den kunne indeksere innhold asynkront av DPG. Solr har støtte for distribuert søk, caching og mye mer.

Hvis en benytter Java er det lett å indeksere og å hente ut svar fra forespørsler ved hjelp av SolrJ [29]. SolrJ er en klient som tilbyr et grensesnitt i Java for å indeksere, oppdatere og gjøre forespørsler til en Solr-server.

For å indeksere data opprettes det dokumenter som i form er helt identisk med de dokumentene som Lucene trenger for å indeksere. Disse dokumentene lagres i en helt vanlig samling (Java sin samlingskontrakt `Collection`). Deretter opprettes det en tilkobling til serveren og dokumentlisten sendes inn til denne. Dokumentene blir omgjort til et lesbart XML-dokument som serveren vet hvordan den skal håndtere. Dokumentene bearbeides automatisk på serveren og er straks klar til bruk.

Serveren er konfigurert med et skjema der det defineres hvilke type inndata som forventes og hvilke datatyper hvert inndatafelt er. I samme konfigurasjonsfil knyttes hvert inndatafelt opp mot en liste med filtre for å optimalisere teksten for søk.

For å søke i de bearbeidede, indekserte dokumentene gjøres det en forespørsel til solrserveren som en HTTP GET forespørsel til en bestemt URL med diverse parametere, og da returneres det et XML-dokument med resultatene fra søket.

Solr har stor fleksibilitet i søkingen ved at den har støtte for *utheving* (eng: *highlighting*) av søkeord, sortering av søketreff og mer. Resultatene kan returneres både i XML-format og JavaScript Object Notation (JSON), slik at de aller fleste språk har støtte for å håndtere svarsettet. Det er også støtte for å begrense antall treff, eller å spørre etter søkeresultat nummer 10-20.

Solr er et veldig aktuelt valg på grunn av den modulære strukturen (alle teknologier som forstår XML/JSON kan hente informasjon fra indeksene), og den ekstra funksjonaliteten, utover det Lucene tilbyr, som kan være interessant.

8.2.6 Endelig valg av teknologi

Ettersom alle forslagene har Lucene i bunn trengs det kun å se på det som tilbys utover grunnfunksjonaliteten til Lucene, for så å undersøke om det er et poeng å benytte noe annet enn bare Lucene.

Lucene gir rask, skalerbar indeksering. Den har kraftige, presise og effektive søkealgoritmer, resultatene kan fåes rangert, en rekke forskjellige typer av spørringer (tilnærmingssøk, frasesøk, og mer), søk på inndatafelt, sortering på felt og mye mer som trengs for å få en effektivt søkemotor.

Tabell 8.1 viser forskjellige kriterier som er lagt til grunn for vurderingen. Alle alternativene har åpen kildekode, baserer seg på Lucene, er skrevet i Java og har god dokumentasjon. De resterende kriteriene blir drøftet under den endelige vurderingen av hver teknologi under tabellen.

Tabell 8.1: Diverse kriterier for valg av søketeknologi

	Lucene	Solr	Nutch	Compass
Åpen Kildekode	☑	☑	☑	☑
Basert på Lucene	☑	☑	☑	☑
Skrevet i Java	☑	☑	☑	☑
God dokumentasjon tilgjengelig	☑	☑	☑	☑
Integrerte løsninger for Spring	☐	☐	☐	☑
Fleksible søkeresultat	☐	☑	☐	☑
Lett å integrere i DPG	☑	☑	☐	☑
Språkuavhengige svar	☐	☑	☐	☐

Å benytte seg av Nutch ser ved første øyekast ut som en god løsning, fordi det lettest indekserbare innholdet i DPG er de endelige rendrerte sidene. Problemene oppstår når Nutch skal kobles til DPG. For at Nutch skal klare å indeksere innhold, må programmet ha tilgang til passordbeskyttede sider. Nutch har støtte for å kunne autentisere seg før den begynner web-crawlingen, men kun dersom nettsidene bruker *NTLM*, *Basic* eller *Digest*

for autentifisering [22]. DPG benytter en selvlaget måte å autorisere brukere på, ved bruk av Webucator sin vevtjeneste. Autentisering til DPG kan skje ved hjelp av såkalt Post-autentisering der brukeren sender inn brukernavn og passord til en bestemt side før den begynner crawingen. Denne løsningen er dessverre ikke implementert i Nutch enda [23].

Selvom hvis dette var implementert ville det likevel vært en kronglete løsning, fordi det er ønskelig at en kun skal indeksere en presentasjon om gangen, og derfor må en gi Nutch tilgang til maks en presentasjon om gangen. Dette gjør at det må opprettes en ny bruker i Webucator for hver presentasjon som er tilgjengelig. På grunnlag av dette ble Nutch ikke den valgte løsningen.

Compass er et alternativ som tilbyr en rekke ekstra funksjonalitet. Compass gjør det lettere å kommunisere med Lucene som opererer på et lavere abstraksjonsnivå. Problemet med Compass er at dens fokusområde er på å håndtere indeksering av data lagret som objekter, XML, JSON eller lignende. Nå er riktignok det mye data håndtert med XML i DPG, men informasjonen ligger lagret både i Velocity-maler, i plugins, og i Jackrabbit sitt repository. Det er først når den endelige siden blir rendrert at en får forståelig informasjon, og derfor mister man mye av det Compass har å tilby.

De siste alternativet er Solr. Så lenge en klarer å generere dokumenter med tekstlig innhold er Solr veldig behjelpelig. En klar fordel med Solr er at serveren er helt seperat fra resten av systemet og derfor går det fint an å bytte ut måten en indekserer dokumenter på uten at det påvirker hvordan svarresultatene hentes ut. Kommunikasjonen mellom server og system skjer språkuavhengig. Solr tilbyr mer enn bare Lucene. På grunnlag av dette ble det bestemt at Solr var det beste valget.

Kapittel 9

Integrering av søk i DPG

For å gjøre det mulig å søke i DPG ved hjelp av Solr, må det installeres og konfigreres en Solr-søkeserver som skal ha innstillinger som passer i forhold til presentasjonene som DPG håndterer. Den viktige neste prosessen er å bestemme hvordan innholdet i presentasjonene skal indekseres, deretter må man bestemme hvordan søketjenesten skal integreres i DPG. Tilslutt må det gjøres valg i hvordan søkeresultatene skal presenteres for sluttbrukeren.

9.1 Installasjon av Solr

Det første steget er å konfigurere og sette opp Solr-serveren. Solr kan konfigureres til å fungere i de fleste servletkontainere, og på flere operativsystem. Siden både testserveren og den lokale installasjonen av DPG kjører på Apache Tomcat [27], ble det et naturlig valg å bruke samme platform for Solr. Det valgte operativsystem ble Fedora 12 av samme årsaker.

9.1.1 Solr på Tomcat

Et av de store valgene når en installerer Solr er å sette opp hvor mange indekser en vil ha, ettersom Solr har støtte for flere kjerner. Dette betyr at du kan ha indeksene til flere applikasjoner kjørende på samme Solr-instans, slik at man slipper å ha en Solr-instans for hver applikasjon. For øyeblikket er det ikke nødvendig med mer enn en kjerne ettersom DPG er eneste applikasjon som vil benytte seg av Solr-søkeserveren, og derfor ble Solr installert med kun en kjerne.

For å konfigurere Solr er det i all hovedsak to konfigurasjonsfiler i XML-format en bør undersøke nærmere. I filen `schema.xml` defineres det hvilke inndatafelt som skal indekseres,

hvilken datatype hvert inndatafelt innholder og hvilke filtre hvert felt skal kjøres gjennom før indeksfilen lagres.

Den andre filen er `solrconfig.xml` der diverse serverrelaterte innstillinger bestemmes.

9.1.2 Indeksering og deklarering av inndatafelt for indekserte dokumenter

Å definere hvilke forskjellige inndatafelt hvert dokument består av er en viktig beslutning fordi det avgjør hvor fleksible søk en kan gjennomføre. Før indekseringen starter bestemmes det hvilke verdier de forskjellige inndatafeltene har. Dersom en er sikker på at enkelte inndatafelter kun innholder tallverdier er det mulig å gjøre søk der det søkes etter dokumenter med verdier innenfor et gitt intervall, eller med verdier over en gitt sum.

Som nevnt tidligere er det `schema.xml` der disse inndatafeltene defineres.

Hvert felt knyttes opp mot en felttype. Denne felttypen består hovedsaklig av en liste med filtre som kjøres enten når feltet indekseres eller når selve spørringen mot serveren kjøres [59]. For å få en nærmere forståelse av valgene som er gjort på de forskjellige feltene, må selve indekseringsprosessen i DPG undersøkes.

Indeksering av alle presentasjoner foregår ved at man først benytter klassen `PresentationSpecificationDao` for å hente ut navnet på alle presentasjoner som er tilgjengelig i DPG. Dette navnet sendes man inn til klassen `PresentationIndexer`, som sender indeksdokumentene fra en presentasjon inn til Solr-serveren.

`PresentationIndexer` henter ut alle *sider* som er knyttet til en presentasjon, fra *sider* henter den ut alle utsnitt. På denne måten kan vi besøke alle utsnitt på hver side og kalle metoden `renderPage()` fra `PresentationViewerService` for å få hentet ut den ferdig rendrerte vevsiden som en tekststreng. Tekststrengen sendes deretter gjennom en SAX-parser fra verktøyet `Tika` [26]. Den genererer en *hendelse* (eng: *event*) hver gang den treffer på en HTML-tag, slik at det er lett å hente ut selve innholdet i tagen, som er det som er interessant å indeksere. Denne prosessen er vist på kodesnutt 9.1.

Listing 9.1: Utdrag fra hvordan indeksering av innhold foregår

```
package no.uib.ii.dpg2.core.search.impl;

public class PresentationIndexerBySolrJ implements PresentationIndexer {

  /**
  * Traverse pages and views and add documents to the Solr-server.
  * @param presentation
  * @throws SolrServerException
  * @throws IOException
```

```
    */
    private void traverseWebpagesAndAddDocumentsToSolr(String presentation)
      throws SolrServerException, IOException {
        Collection<SolrInputDocument> docs =
          new ArrayList<SolrInputDocument>();

      presentationSpecification =
        presentationViewerService.getPresentation(presentation);

      List<PageState> pageStates =
        presentationSpecification.getPageStates();

      for (PageState pageState : pageStates) {
        Page page = pageState.getPage();
        List<ViewState> viewStates = pageState.getViewStates();
        for (ViewState viewState : viewStates) {
          View view = viewState.getView();
          docs.add(
            addNewDocumentFromPageContents(
              presentation,
              page.getId(),
              view,
              pageState.getLabel(),
              viewState.getLabel()
            )
          );
        }
      }

      server.add(docs);
      server.commit();
    }
}
```

En fremtidig løsning vil forhåpentligvis også ta hensyn til tittelen på siden den parser, slik at også dette kan lagres som et felt. Det er vanlig at når de forskjellige søkeresultatene skal presenteres, så er hovedteksten til hvert søketreff tittelen på siden en kommer til. Dette er dessverre ikke den beste løsningen i DPG fordi det krever at denne tittelen settes manuelt for hver side. Tittelen på hver side er i DPG 2.0 navnet på presentasjonen den befinner seg i.

SolrJ sender inn en samling med objekter av typen `SolrInputDocument` . Dette objektet er en nøkkelverditabell bestående av inndatafelt og verdier. Navnet på nøkkelen må matche det som er beskrevet i `schema.xml`. Dokumentet består av fem felter. Kodesnutt 9.2 viser formen på inndataene som Solr-serveren godtar.

Listing 9.2: De fem inndatafeltene som Solr-serveren er konfigurert med

```
<add>
<doc>
    <field name="presentationId">eLearning</field>
```

```
      <field name="view">Nettsteder</field>
      <field name="page">Ressurser</field>
      <field name="content">
        The Java Tutorial. En praktisk introduksjon for programmerere med
        mange komplette, fungerende eksempler og flere innforinger pa
        spesielle emner.
      </field>
    <field name="url">
        presentation.html?pid=eLearning&page=ressurser&view=webLinkView
    </field>
  </doc>
  </add>
```

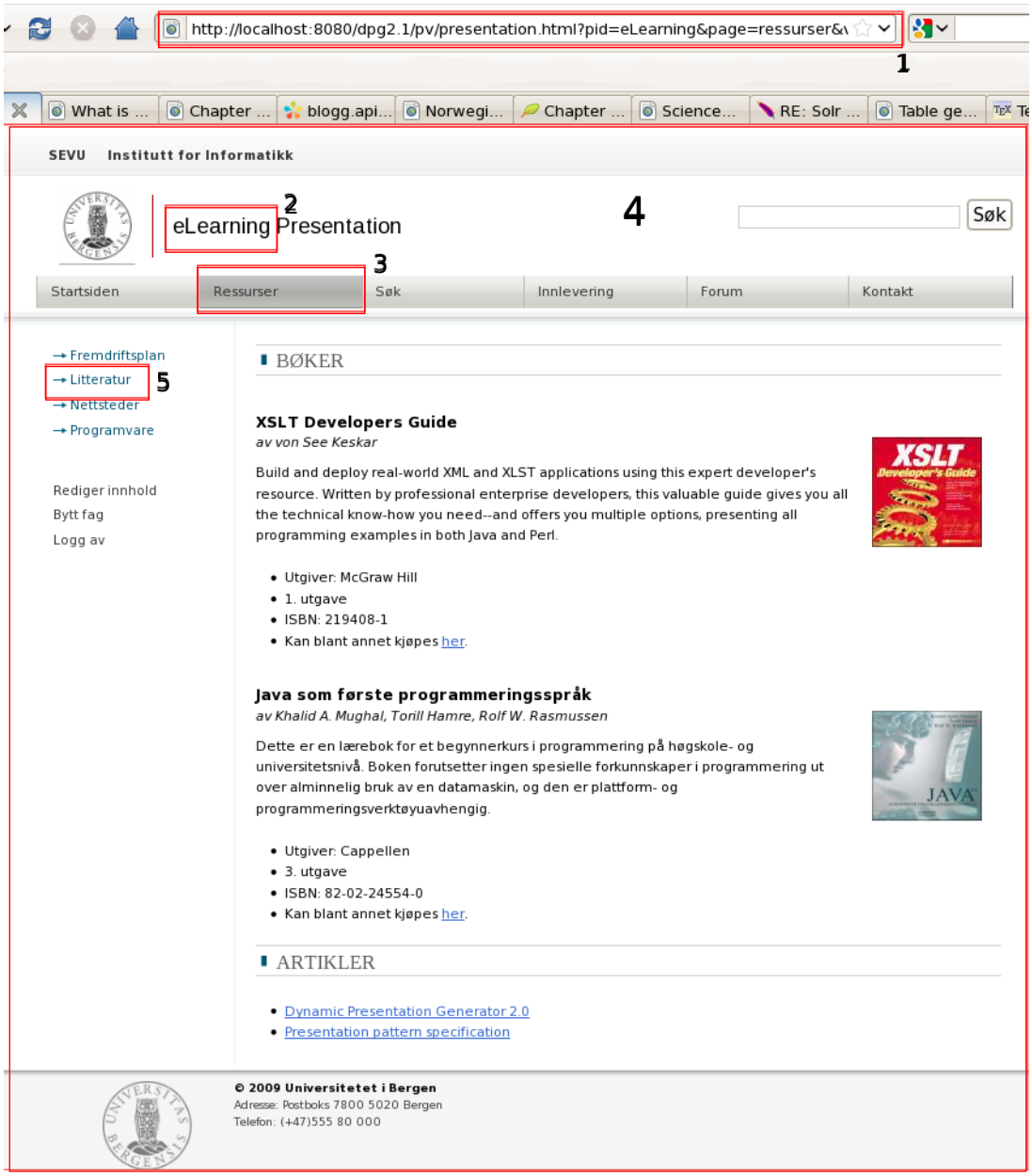
På skjermbildet 9.1 ser hvor de forskjellige feltene henter sin inndata fra.

1. `url`: lenke til siden som besøkes for å gjenfinne innholdet
2. `presentationId`: en unik identifikator for presentasjonen, på skjermbildet er det etiketten som er uthevet, selve identifikatoren er skjult for brukeren.
3. `page`: etikett på siden dokumentet er tilknyttet
4. `content`: selve innholdet i dokumentet
5. `view`: etikett på utsnittet dokumentet er tilknyttet

`content` er selve HTML-innholdet som blir generert når sider rendreres, `url` er lenken til siden der et evenuelt søketreff befinner seg. Den er generert av en kombinasjon av presentasjonsidentifikator, *side* og utsnitt. I tillegg lagrer den etikettenavnet på hver side og utsnitt, det vil si det navnet som står i menyene til hver side og hvert utsnitt. Den lagrer også presentasjonsidentifikatoren slik at en har mulighet til å gjøre søk der det svarsettdokumentene har denne identifikatoren.

Serveren vet da hvordan de forskjellige inndatafeltene skal behandles (i forhold til filtrering av informasjon og lignende), og både indekserer og lagrer dokumentene, slik at det er mulig å foreta et søk senere. Når hvert felt blir indeksert blir det utsatt for en rekke analyseringer og det vi kaller *avsøking* (eng. *tokenizing*). Avsøking handler om å dele innholdet inn i biter, og analyseringen går på å fjerne unødvendige biter og å tilpasse de resterende bitene slik at de er optimal i forhold til lagring. [59].

Feltet `content` inneholder det meste av innholdet på siden, og det er i dette feltet en oftest vil søke i. Innholdet knyttes opp mot felttypen `text`. Solr bruker Lucene sin klasse `WhitespaceTokenizer` for å avsøke tekststrengen med mellomrom som skilletegn, det vil si at tekststrengen deles opp i biter der mellomrom er brukt som skilletegn. Den har to forskjellige tekstanalyserere tilknyttet seg, en som aktiveres når dokumentene sendes inn til til indeksering hos serveren, og en som benyttes når forespørsler sendes til serveren. Ut fra hvordan spørringen er utformet kan den da benytte forskjellige tekstanalyserere. Solr benytter Lucene sine klasser for å filtrere innholdet. Inndatafeltet er knyttet opp mot følgende filtre:



Figur 9.1: Indekserte felter

- `LowerCaseFilter` - Gjør teksten om til små bokstaver
- `StopFilter` - Fjerner alle stoppord som er definert i en egen tekstfil. Innholdet i tekstfilen er hentet fra Jan Bruusgaard, som har konstruert en norsk stoppordliste [7].
- `SynonymFilter` - Definerer grupper med ord som er syntaktisk forskjellig men semantisk lik, som for eksempel "`nettet`" og "`veven`" i en egen tekstfil. Her er

det også mulig å legge til vanlige skrivefeil som slik at for eksempel "abonnement" blir oversatt til "abbonement".
- WordDelimiterFilter - Splitter opp ord til subord, fordi de på en eller annen måte henger sammen til vanlig. Et eksempel er at Hi-Fi blir til Hi Fi og O'Shea blir til O Shea.
- SnowballPorterFilter - tilbyr stemming av ord, med mer. [1]

Inndatafeltene page og view trenger ikke å indekseres fordi innholdet her finnes også i inndatafeltet content.

De to siste feltene, url og presentationId blir heller ikke indeksert, ettersom disse feltene innholder ikke-søkbare identifikasjonsverdier.

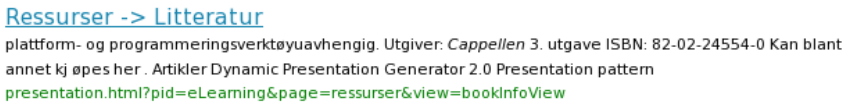
Det er også et par innstillinger i konfigurasjonsfilen Schema.xml der beslutninger er tatt. En definerer blant annet hvilket inndatafelt som en nøkkelen. Verdien må være unik, slik at dokumenter kan kunne skilles fra hverandre.

En viktig innstilling i konfigurasjonsfilen Schema.xml er om serveren skal benytte AND eller OR-operatoren når brukere fører inn flere søkeord. Hvis en ser tilbake på delkapittel 8.1.3.1, Hvor bra er et søkeresultat? ser man betydningen av dette. Det ble besluttet å benytte OR-operatoren. Dette var fordi det kun søkes i en presentasjon, noe som gjør at det blir mindre indekser. Da er det bedre å få et par upresise sider enn å ikke få søketreff på en relevant side. Hele konfigurasjonsfilen er tilgjengelig i tillegg A.

9.1.3 Serverinnstillinger

Nå som inndatafeltene er definerte, gjenstår det bare å ta hånd om de resterende serverinnstillingene som er definert i filen solrconfig.xml. De fleste valg i denne filen er ment for mer avansert bruk, og derfor er standardinnstillingene valgt på de aller fleste valg. Kun endrede innstillinger er omtalt.

Solr har støtte for fremheving av teksten som ble søkt på i svarsettet. Dette gjør at det er lettere å finne igjen den søkte teksten, som gjør det lettere å finne konteksten ordet står i. En kan definere hvor mye av teksten rundt søketreffordet som blir returnert. Standardverdien for dette er lav, den returnerte strengen er på 70 tegn, mens for eksempel Google returnerer rundt 150 tegn. Derfor ble variabelen som regulerer dette, hl.fragsize oppjustert til 150 tegn. Figur 9.2 viser et eksempel på hvordan en slik kontekst ser ut. Søkeordet *Cappelen* er i uthevet skrift.



Figur 9.2: Kontekst rundt søkeord

9.2 Når skal innholdet indekseres?

En utfordring når det kommer til søk, er å bestemme når innholdet skal indekseres. For større søkemotorer er det vanlig at indekseringen foregår både dag og natt, men da er det ofte hele verdensveven som skal indekseres. Den første løsningen var å ha en knapp som manuelt måtte trykkes på for å indeksere alle presentasjoner. Dette var en midlertidlig løsning, og etterhvert ble det funnet to nye måter å indeksere innholdet på.

9.2.1 Indeksering på definerte tidspunkt

Den ene går ut på å automatisk indeksere innholdet ved faste tidspunkter. Til dette ble jobbplanleggingstjenesten *Quartz* [68] brukt. Dette er en tjeneste som lar integrere sømløst med Spring.

Å legge til en ny planlagt jobb i Quartz er en ganske enkel implementasjon. Quartz tilbyr flere forskjellige måter å planlegge slike jobber på. For å få Quartz til å fungere må tre prosesser følges:

1. Definer hvilken klasse og hvilken metode som skal kjøres.
2. Lag en *utløser* (eng: *trigger*), denne bestemmer hvor ofte den tilknyttede klassen og metoden skal kjøres.
3. Opprett en planleggingsfabrikk som inneholder en liste over alle jobber som kjører på et prosjekt.

Disse prosessene håndteres av prosjektkonfigurasjonsfilene som Spring håndterer.

Siden indeksering er en relativt ressurskrevende prosess, bør man unngå å indeksere innholdet for ofte. Likevel er det viktig at indeksene er oppdaterte slik at de gjenspeiler det faktiske innholdet på de forskjellige presentasjonene.

Det ble bestemt å bruke to forskjellige utløsningsklasser. Den første er klassen `SimpleTrigger`, i den defineres det et tidspunkt for første gang metoden skal kjøres, og et repitisjonsintervall. Denne klassen brukes for å indeksere presentasjonene kort tid etter serveroppstart. I DPG ble verdien for første kall satt til tre minutter etter oppstart.

Den andre triggeren er en *cron*-trigger, som baserer seg på det samme systemet som cron-jobber i Linux. Ved å definere forskjellige verdier kan slike jobber kjøres en gang i måneden, eller en gang hver tredje dag på bestemte klokkeslett, og lignende. For DPG er det nok å kjøre indekseringsmetoden en gang i døgnet. For å unngå å forstyrre eventuelle brukere er den satt til å kjøres klokken 06:00.

Kodesnutten 9.3 viser applikasjonskontekstkonfigurasjonsfilen til DPG, og hvordan de forskjellige klassene er konfigurert. Bønnen `automaticIndexer` holder en referanse til klassen som er ansvarlig for å indeksere alt innhold, mens de to andre klassene sørger for å generere hendelser som gjør at metoden definert kjøres på bestemte tidspunkt.

Listing 9.3: Referansen til metoden som indekser og triggerklassene som kjører metoden på bestemte tidspunkt

```
<bean id="automaticIndexer"
  class="org.springframework.scheduling.quartz.
      MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="indexAllPresentations" />
  <property name="targetMethod" value="indexAllPresentations" />
</bean>

<bean id="triggerIndexing"
class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail" ref="automaticIndexer" />
  <property name="startDelay" value="180000" />
  <property name="repeatInterval" value="604800000" />
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.
    CronTriggerBean">
  <property name="jobDetail" ref="automaticIndexer" />
  <!-- run every morning at 6 AM -->
  <property name="cronExpression" value="0 0 6 * * ?" />
</bean>
```

9.2.2 Identifisering av hendelser som påvirker indeksinnholdet

For å holde indeksene så oppdatert som mulig, var det en idé å indeksere innholdet hver gang innholdet i presentasjonen ble endret. Denne hendelsen kan inntreffe ved to forskjellige hendelser i DPG:

1. når et HTML-skjema i delsystemet PCE blir sendt inn
2. når interaksjonsplugins behandles

Det var enkelt å sette opp DPG til å indeksere alt innholdet i en presentasjon på nytt når et HTML-skjema ble oppdatert. Likevel var det flere problemer med en så enkel løsning.

Hvis presentasjonen var stor tok indekseringen lang tid. Systemet var ikke responsiv mens indekseren pågikk. Derfor måtte denne prosessen forbedres.

Den første forbedringen var å indeksere innholdet asynkrynt, ved å legge indekseringen i en egen tråd. Dette er heldigvis en enkel oppgave, siden Spring har innebygget støtte for den slags [12]. Kontrakten `TaskExecutor` som tilbys av Spring er identisk til `java.util.concurrent.Executor`. Den inneholder en enkel metode: `execute(Runnable task)`.

For å implementere dette ble det laget en ny *oppgave* (eng: *task*) i presentasjonsindekseringsklassen `IndexAllPresentations`. Oppgaven er lagt til som en innerklasse i klassen 9.4.

Listing 9.4: Oppgave for asynkron indeksering av innhold

```
/**
 * Task for asynchronous indexing of views.
 * @author tol060
 *
 */
private class ViewIndexerTask implements Runnable {

  private String presentationId;
  private String entityInstance;

  public ViewIndexerTask(String presentationId, String entityInstance) {
    this.presentationId = presentationId;
    this.entityInstance = entityInstance;
  }

  @Override
  public void run() {
    indexSelectedViews(presentationId, entityInstance);
  }
}
```

Det injekteres en `TaskExecutor` fra Spring inn i klassen ved hjelp av Dependency Injection [35]. Når metoden `asyncIndexSelectedViews()` kalles kalles, kjøres `execute()`-metoden fra den injekterte klassen `TaskExecutor` med den nye oppgaven som argument. I execute-metoden går det et kall tilbake til ytterklassens metode `indexSelectedViews()`. Denne metoden er implementert på lik måte som `writeIndexes()` før endringene.

Ved å kjøre indekseringen asynkront ble programmet mer responsivt under indeksering, og med denne enkle løsningen var det stor forbedring siden man kunne fortsette å bruke systemet mens indekseringen pågikk. Likevel brukte indekseringen mye unødvendige ressurser fordi endringer på verdier i en entitet påvirker ikke alle utsnitt, og derfor kan flere sider rendreres unødvendig, noe som er belastende for serveren. Derfor ble det foreslått å lage en kobling mellom hvilken entitet som ble endret, og hvilke utsnitt som var påvirket av endringen.

Fra PCE-delen der endring av innhold foregår får man tilgang til variabelen *entityInstance*, denne består av en unik identifikator til en entitet i en presentasjon. For å finne utsnitt som er påvirket må en gjennom alle pages og alle utsnitt for å finne utsnitt som er tilknyttet denne entiteten. Kodesnutt 9.5 viser hvordan dette gjennomføres.

Listing 9.5: Finner utsnitt som er påvirket av endringer på en bestemt entitet

```
/**
 * Finds all views affected by a change in the entity instance.
 * @param presentationId
 * @param entityInstance
 * @return Map with each page and all views affected within that page
 */
public Map<PageState,List<ViewState>> findAffectedViews(String
    presentationId, String entityInstance) {
  presentationSpecification = presentationViewerService.getPresentation(
      presentationId);
  Map<PageState,List<ViewState>> result = new HashMap<PageState,List<
      ViewState>>();

  for (PageState pageState : presentationSpecification.getPageStates()) {
    List<ViewState> views = new ArrayList<ViewState>();
    for (ViewState viewState : pageState.getViewStates()) {
      View view = viewState.getView();
      logger.debug("Checking if entityinstance " + entityInstance + " is
          equal to  " + view.getEntityInstance().getId() + ".");
      if(view.getEntityInstance().getId().equals(entityInstance)) {
        views.add(viewState);
      }
    }
    logger.debug("Views: " + views);
    result.put(pageState, views);

  }
  return result;
}
```

Denne kodesnutten returnerer en nøkkelverditabell der hver nøkkel er en side, og den har en liste med utsnitt som er påvirket på hver side. Da er det lett å rendrere siden og oppdatere indeksen kun for de påvirkede dokumentene.

9.3 Implementasjon av søk i DPG

Når Solr-serveren er satt opp, og innhold er indeksert, kan det foretas enkle HTTP-GET-forespørsler til serveren for å hente ut søketreff. Listing 9.6 viser eksempel på resultat når det gjøres en forespørsel til serveren med søkeordet *jdk*:

`http://localhost:8080/solr/select/?q=jdk`.

Resultatet er manuelt forenklet for å illustrere poenget.

Listing 9.6: XML resultat fra søkeserver med søkeordet "JDK"

```
<response>
  <lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">1</int>
  <lst name="params">
    <str name="q">jdk</str>
  </lst>
</lst>

<result name="response" numFound="2" start="0">
  <doc>
    <str name="content"> ... programmeringsspraket Java. JDK - Java
    DevelopmentKit .. </str>
    <str name="page">Startsiden</str>
    <str name="presentationId">inf219</str>
    <str name="url"> presentation.html?pid=inf219&page=startPage&view=
        softwareView </str>
    <str name="view">Pensum</str>
  </doc>
  <doc>
    <str name="content">Java.  JDK - Java Development Kit  Inneholder JRE
        og i</str>
    <str name="page">Ressurser</str>
    <str name="presentationId">eLearning</str>
    <str name="url">presentation.html?pid=eLearning&page=ressurser&view=
        softwareView</str>
    <str name="view">Programvare</str>
  </doc>
</result>
</response>
```

Det er enkel og lesbar XML som blir returnert, likevel er det praktisk å benytte seg av SolrJ for å kommunisere med serveren istedenfor å tolke XML-en manuelt.

For å gjøre søking mulig ble det opprettet en serviceklasse med navn `PresentationSearcher`. Denne klassen har en metode med følgende signatur:

Listing 9.7: Metodesignatur for å søke i presentasjoner

```
public List<HitBean> searchPresentation(
  String presentationId,
  String searchString,
  int page
);
```

Implementering av metoden benytter seg av SolrJ for å hente ut søkeresultatene fra serveren. Først forberedes forespørselen der en rekke parametere settes, derettes sendes forespørselen

til serveren, og en får returnert en liste med `SolrDocument`-objekter, dette prosesseres i metoden `handleResult()`. Listing 9.8 viser hvordan dette er implementert. I søket sendes det med en en filtreringsopsjon som gjør at det kun returneres treff fra en bestemt presentasjon. For å få et mer håndterlig format på søkeresultatene er det opprettet en bønneklasse ved navn `HitBean` som inkapsulerer resultatene fra spørringen og en `HitBeans`-klasse som består av en liste med `HitBeans` og et felt som sier hvor mange treff søket fant totalt. `HitBean`-klassen inneholder de samme fem feltene som dokumentet som ble indeksert: `url`, `presentationId`, `viewLabel`, `pageLabel` og `content`. Hvordan denne avbildningen er implementert kan sees i listing 9.9.

Listing 9.8: Searching in a presentation

```
/**
 * Search in a given presentation after a certain keyWord (aka searchString
    )
 */
@Override
public HitBeans searchPresentation(String presentationId, String
    searchString, int page) {
  hitBeans = new HitBeans();
  hits = new ArrayList<HitBean>();

  try {
    SolrQuery query = setupQuery(presentationId, searchString, page);
    QueryResponse rsp = server.query(query);
    SolrDocumentList docs = rsp.getResults();

    hitBeans.setTotalNumberOfHits((int)docs.getNumFound());

    logger.debug("Found " + docs.size() + " hits in total.");

    for (SolrDocument document : docs) {
      handleResult(document, presentationId, rsp);
    }

  } catch (SolrServerException e) {
    e.printStackTrace();
    logger.debug("Could not make a connection to the solr-server. Are you
        sure
    it's up and running?");
  }

  hitBeans.setHitBeans(hits);
  return hitBeans;
}
```

Listing 9.9: Avbildning fra SolrDocument til HitBean

```
/**
 * Transforms one SolrDocument into a HitBean-object.
 *
```

```
     * @param document
     * @param presentationId
     * @param rsp
     */
    private void handleResult(SolrDocument document, String presentationId,
        QueryResponse rsp) {
      HitBean hitBean = new HitBean();
      hitBean.setPresentationId(presentationId);

      hitBean.setContext((String) document.getFieldValue("content"));
      hitBean.setView((String) document.getFieldValue("view"));
      hitBean.setPage((String) document.getFieldValue("page"));

      String url = (String) document.getFieldValue("url");
      hitBean.setUrl(url);

      if (rsp.getHighlighting().get(url) != null) {
        List<String> highlightSnippets = rsp.getHighlighting().get(url).get("
            content");
        hitBean.setContext(highlightSnippets.get(0));
      }

      hits.add(hitBean);
      logger.debug("Added a new search result bean!");
    }
```

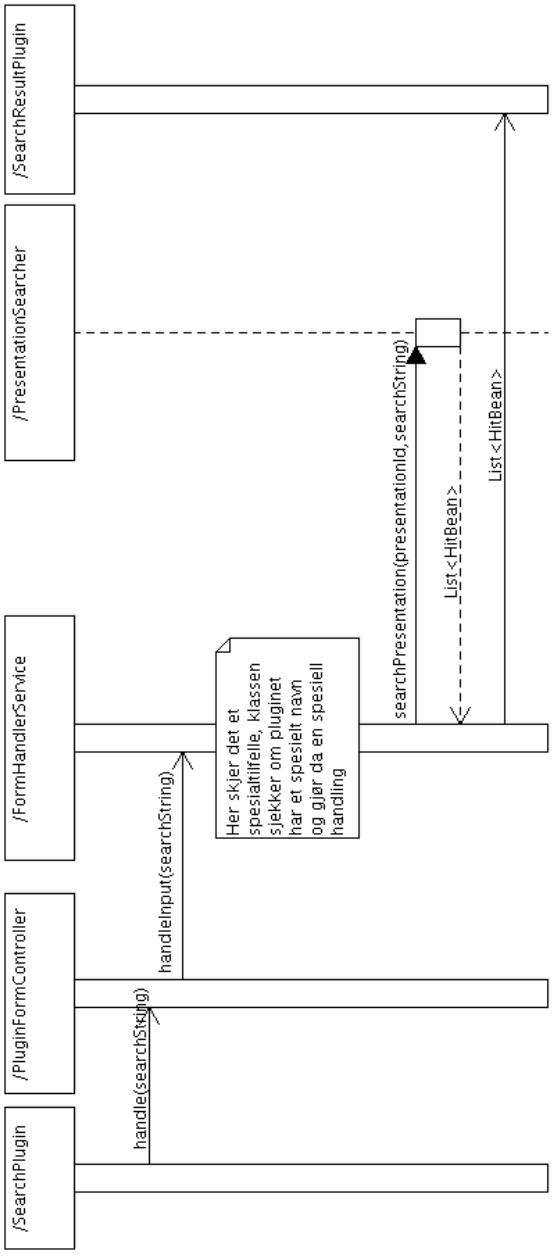
For å søke føres det inn søkeord i søkefeltet. En kan definere mer avanserte spørringer ved å bruke operatorer som hermetegn, plusstegn og minustegn. Hvis søkestrengen inkapsuleres i hermetegn spiller rekkefølgen en rolle, ved å bruke minus etterfulgt av nøkkelord kan søkeresultater med dette ordet utelukkes.

Den andre mulige hendelsen som kan endre innhold, er når interaksjonsplugins ble innsendt. Dette anses som en hendelse som kan skje for ofte til at det er hensynsmessig å indeksere innhold når dette inntreffer. Derfor blir innhold generert fra interaksjonsplugins kun oppdatert når Quartz oppdateres innholdet en gang i døgnet.

9.4 Integrering av søk i DPG

Det finnes flere alternativer for hvordan søk kan integreres i DPG. Den første ideen var å benytte seg av plugins for å håndtere søk, ettersom plugins har mulighet til å konstruere HTML-skjemaer, som kreves for å lage selve søkefeltet der søkeord føres inn. En slik plugin ble implementert. Skjemaet ble sendt inn til DPG og håndtert i klassen `PluginFormController`. I den første implementasjonen ble det undersøkt om pluginet som håndterte feltet hadde navnet `searchPlugin`. Hvis dette stemte, ble et søk gjennomført ved hjelp av `PresentationSearcher`-grensesnittet. En liste med søketreff som ble

omgjort til `HitBeans`. Dette `HitBeans`-objektet ble sendt med tilbake i nøkkelverditabellen `inputs` som ble sendt tilbake til søkepluginet. Søkepluginet brukte listen med søketreff for å generere en liste med JDOM-elementer som innholdt søkeresultatene. Når resultatsiden ble rendrert i DPG ville søkeresultatene vises. Et sekvensdiagram som viser denne løsningen er illustrert i figur 9.3.



Figur 9.3: Den første skisserte løsningen

Denne løsningen var ikke optimal. Søkingen ble gjort til et unntakstilfelle som ikke fulgte den standardiserte måten plugins ble behandlet på. Derfor ble det laget to alternative løsninger:

1. Forandre løsningen slik at den fungerte fungerte på lik linje med andre plugins
2. Ha søking som en del av kjernefunksjonaliteten i DPG

9.4.1 Søking med bare plugins

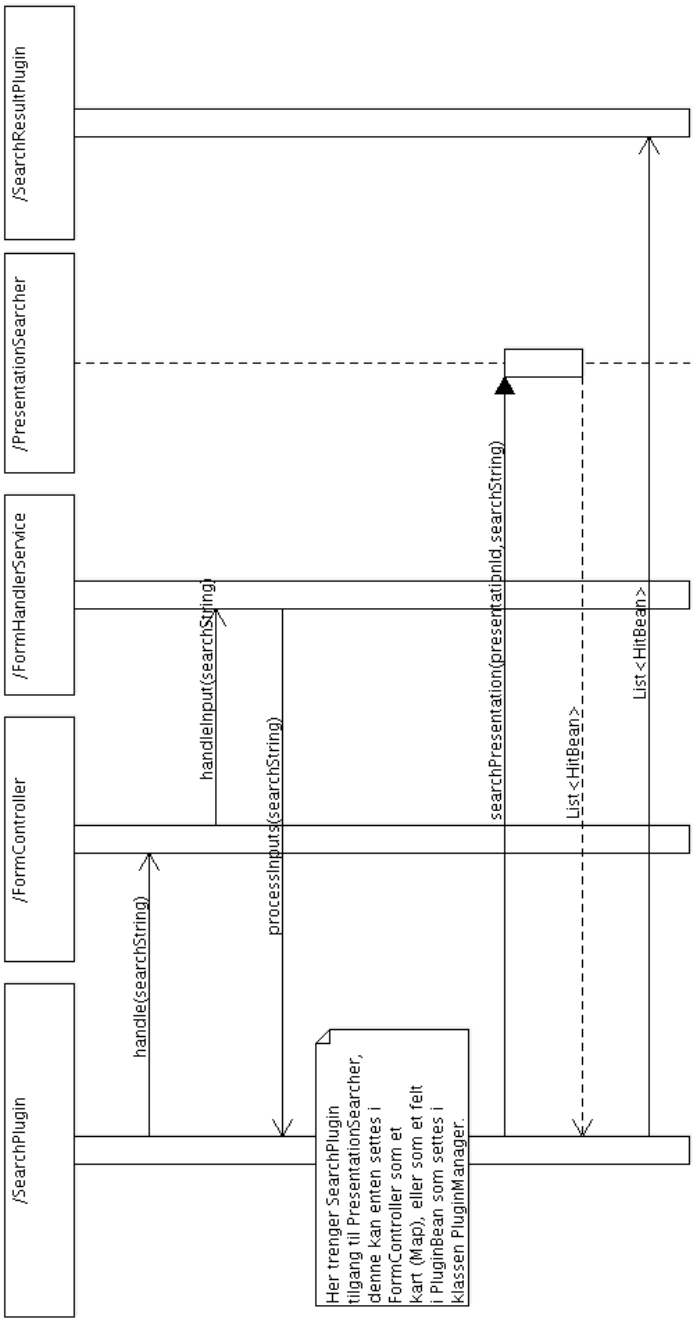
For at en slik løsning skal fungere, må søkepluginet på en eller annen måte ha tilgang på klassen `PresentationSearcher`. Dette kan kun gjøres ved at det legger til som et felt i klassen `PluginBean`, eller ved å inkludere `PresentationSearcher` i nøkkelverditabellen som blir videresendt til pluginet via metoden `processInputs()`. Sekvensdiagrammet på figur 9.4 skisserer hvordan de forskjellige komponentene ville interagert i en slik løsning.

Ettersom det i kapittel 4: Analyse av pluginarkitektur ble bestemt at plugins ikke skulle ha tilgang til slike ressurser, er det ikke mulig å injeksere en `PresentationSearcher` i selve pluginet. Løsningen med å sende det med i `PluginBean` er heller ikke ønskelig fordi det vil bli et spesialtilfelle ettersom det kun er søkepluginet som vil behøve tilgang til denne variabelen. Alternativet med å sende det med i nøkkelverditabellen til metoden `processInputs()` hos pluginet er også et spesialtilfelle, fordi det da må injekseres et objekt av klassen `PresentationSearcher` til alle plugins. Derfor er det heller ikke ønskelig behandle søking med plugins.

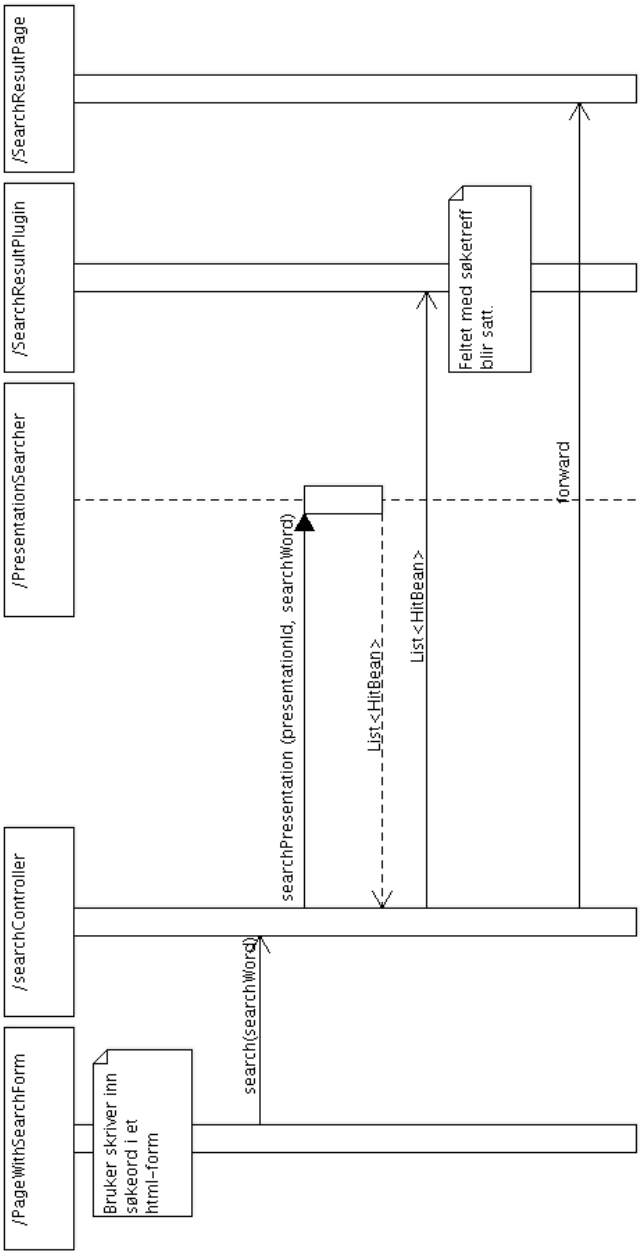
9.4.2 Søking som en del av kjernefunksjonaliteten i DPG

Med en løsning der søk er en del av kjernefunksjonaliteten i DPG er det mulig å legge inn søk som en del av malen til alle sider, ved å legge HTML-skjemaet direkte i hovedmalen `master_template.vm`. Søking foregår da ved at skjemaet som inneholder søkeordet blir sendt inn til en søkekontroller, som har et `PresentationSearcher`-objekt som brukes for finne søkeresultatene. Resultatene vil lagres som sessiondata, som enten kan leses ut i plugins, eller skrives ut direkte i søkeresultat-malen. Figur 9.5 skisserer denne løsnignen.

Et krav til denne løsningen er at en må ha en side og et utsnitt dedisert til søking. Det er mulig at en framtidig editor vil kunne automatisk legge til en side og et utsnitt med en spesiell identifikator, slik at dette ikke trenger å føres opp eksplisitt.



Figur 9.4: Søkeløsning som kun benytter plugins



Figur 9.5: Søkeløsning uten plugins

9.4.3 Tilrettelegging av sesjonsdata i velocitymaler

Det ble bestemt at søking skulle være en del av kjernefunksjonaliteten i DPG. Den enkleste løsningen var å lage både søkeskjemaet og søkeresultatene i malen, istedenfor å gå veien

via plugins. Dette viste seg å være problematisk, fordi DPG ikke har sesjonsdata i malene. Derfor måtte dette tas hånd om.

Løsningen ble å sende `HttpSession`-objektet som ble gjort tilgjengelig i kontrollerklassen `PresentationController` med til `renderPage()`-metoden som er ansvarlig for å rendrere sider. Sesjonsobjektet ble da gjort tilgjengelig i Velocity-malene, som gjorde det mulig å hente ut sesjonsdata derfra.

9.5 Presentasjon av søk

For at brukeren skal finne fram til informasjon ved hjelp av søk, er det viktig at søkeresultatsiden er lettfattelig og oversiktlig. Det bør ikke være for lite, og heller ikke for mye informasjon i hvert søketreff, og hvert søketreff bør være markert adskilt. I tillegg må det være mulig å få et visst innblikk i innholdet på siden der søkeresultatet finnes. Det er også smart å avgrense antall treff på en side, slik at ikke brukeren ikke overlesses med informasjon. Dette hjelper også på lastetider. Appendix B viser hele implementasjonen av denne malen. På grunnlag av dette ble søkeresultatet utformet som på figur 9.6.

Det kommer godt fram at det siste søkeordet var *kontakt* ettersom det står i søkefeltet. Det er også lagt til i den første linjen etter søkeboksen. Søkeordet er skrevet i uthevet tekst for å understreke dens betydning.

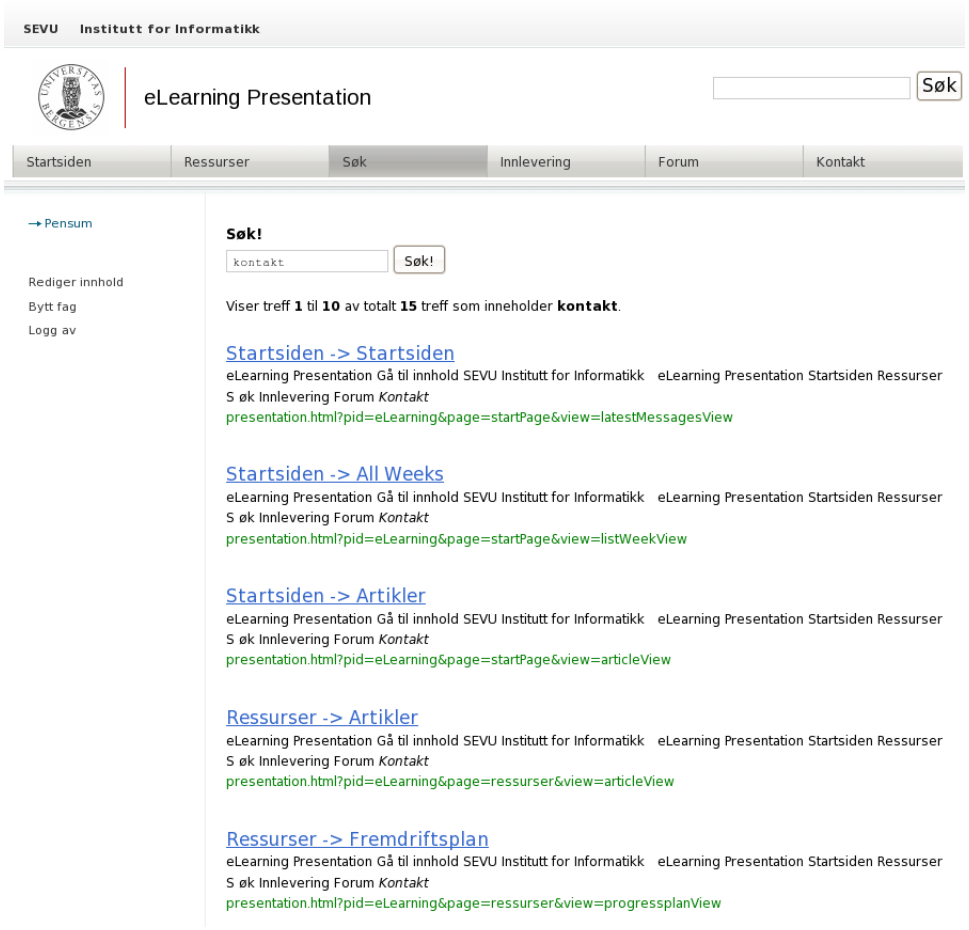
Det er også oppført hvor mange treff som ble funnet totalt, og hvilke treff som vises på skjermen nå. På bunnen av siden er det sidenavigeringsvalg dersom man har funnet flere enn 10 treff. Hvis man har færre treff enn 10, vil ikke sidenavigeren dukke opp.

Hver søkeresultat består av tre elementer, en overskrift, litt kontekstbasert informasjon og selve lenken til svarsettet.

Overskriften til søkeresultatet er skrevet i litt større skrift, og er sammensatt av to elementer: `PageLabel` og `ViewLabel`. Dette gjør at det er mulig å lese om hvor informasjonen finnes. I tillegg er overskriften en lenke til dokumentet der informasjonen er.

Den kontekstbaserte informasjonen er skrevet i vanlig skrift, og baserer seg på teksten som er skrevet rundt søkeordet som det var treff på. Selve søkeordet er skrevet i kursiv. Dette er fordi det skal være lettere å få øye ordet.

Den tredje delen er en selve lenkeaddressen en må følge for å komme til resultatsiden. Den er skrevet i grønn tekst, etter inspirasjon fra Google.



Figur 9.6: Søkeresultat

9.5.1 Autokomplettering ved hjelp av AJAX Solr

Når en bruker fører inn en bokstav i søkefeltet, vil DPG automatisk gi en liste med søkeforslag hentet fra det indekserte innholdet på Solr-serveren. Dette er mulig på grunn av et rammeverk som kombinerer Solr og asynkrone serverkall ved hjelp av AJAX. Navnet på dette rammeverket er *AJAX Solr* [15]. Rammeverket bruker AJAX som et grensesnitt mellom Solr og applikasjonen. AJAX Solr tilbyr en rekke *innretningerer* (eng. *widgets)*, for å blant annet lage merkelappskyer med de mest populære indekserte ordene, kalender for å se dokumenter tilknyttet visse datoer, og mer. Den valgte innretningen tilbyr søkeforslag når en bruker skriver inn søkeord i søkefeltet.

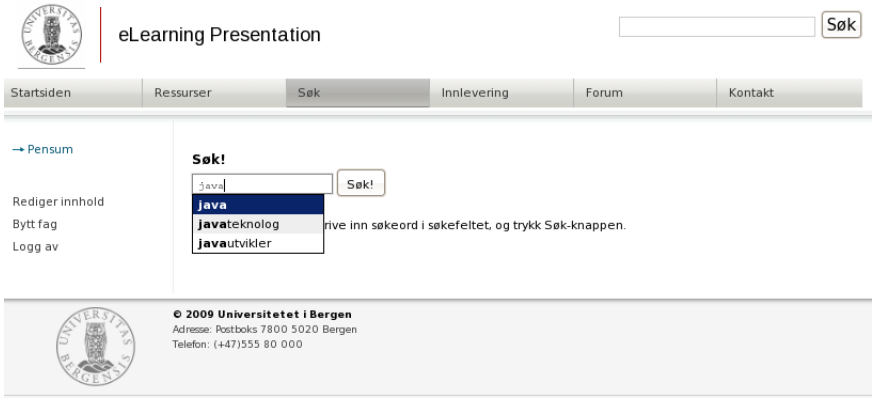
AJAX Solr benytter seg av jQuery [62], et rammeverk til JavaScript, som er et scriptspråk som opererer på klientsiden. Innretningen for autokomplettering av søkeord benytter seg av en autokompletteringsplugin som er tilgjengelig for jQuery.

Oppsettet av AJAX Solr er relativt enkelt, det første som trengs er en manager som holder styr på hvilke innretninger som skal være lastet, i tillegg til selve tilkoblingen til Solr-serveren. Den må også ha et sted å lagre forskjellige Solr-parametere, dette skjer ved hjelp av en klasse kalt `ParameterStore`. Manageren har en initialiseringsmetode som initialisere tilkoblingen til Solr-serveren, setter opp nødvendige parametere til `ParameterStore` og klargjør de forskjellige innretningene. Hver innretning har metoder som kalles av manageren før, under og etter spørringen mot Solr-serveren. For autokompletteringspluginet er det etter spørringen at arbeidet utføres.

Autokompletteringspluginet trenger et par parametere, blant annet hvilket element den skal virke på, og hvilke felter i Solr som den skal hente ut forslag fra.

Hvis AJAX Solr er feil satt opp vil ikke dette påvirke systemet ellers.

Figur 9.7 viser hvordan denne løsningen ser ut i praksis.



Figur 9.7: Eksempel på live-søkeresultat

Kapittel 10

Konklusjon, evaluering og videre arbeid

10.1 Evaluering av mål

10.1.1 Tilrettelegge arkitekturen i DPG for interaksjon

En del av oppgaven bestod i å undersøke arkitekturen til DPG 2.0 for å kunne finne ut hvordan interaksjon best kunne oppnås. Dette ble løst ved å tilpasse pluginarkitekturen med dette formålet. Med den nye pluginarkitekturen er det en enklere prosess å lage nye plugins ettersom konfigurasjonsfilen `plugIn.xml` er fjernet og erstattet med løsninger som ikke krever at presentasjonsmønsterutviklere må føre inn informasjon om plugins manuelt. Denne løsningen har ikke utelukkende fordeler, en mister blant annet et oversiktsdokument over alle tilgjengelige plugins til bruk i DPG, og en må derfor manuelt gå inn i `JAR`-filene for å undersøke hvilke plugins som finnes. Dette kan løses ved å introdusere et program som kan skrive ut navnet på- og informasjon om alle plugins som ligger lagret i pluginsmappen. Det integrerte utviklingsmiljøet eclipse har også en pluginarkitektur, og i dette programmet finnes det en oversikt over installerte plugins, med et grensesnitt til hvordan en installerer nye, eller fjerner installerte plugins. Det kan være interressant å utvikle noe tilsvarende for DPG.

Plugins kan nå lese- og skrive ressurser, noe som gjør dem mer uttrykkskraftig og gir flere muligheter. Løsningen som er beskrevet benytter det eksisterende persistenslaget. Selve implementasjonen er skjult bak en kontrakt, slik at det skal være mulig å endre databaseløsning uten at det påvirker måten plugins lagrer innhold på.

Pluginskontraktene er oppdatert, noe som har gjort at alle tidligere plugins ikke vil fungere. Dette var likevel nødvendig ettersom kontraktene både var uryddige og hadde for mange

parametre. Plugins som eksisterte i DPG 2.0 må lages på nytt, noe som ikke er ideelt. I tillegg er det tilbydt en ny metode som tillater at plugins kan håndtere inndata.

Pluginarkitekturen er blitt betrakelig forbedret, og målet om å tilrettelegge arkitekturen for interaksjon er oppnådd, til tross for at det fremdeles er et par mangler i pluginarkitekturen. Plugins kan fremdeles ikke reagere på hendelser i systemet. Et annet problem er at det er det kan være vanskelig å finne implementasjonen av plugins dersom utvikler ikke legger ved kildekoden i JAR-filen med class-filen til et plugin.

10.1.2 Legge til interaksjonsmuligheter i DPG

Det ble utviklet to nye plugins som viser interaksjonsmulighetene gjort tilgjengelig med den nye pluginarkitekturen. Forfatteren er fornøyd med at interaksjonsplugins ble laget ved å benytte eksisterende løsninger og målet er oppnådd. selvom de implementerte plugins har forbedringspotensiale, er det nyttige plugins som har bevist at det er mulig å skape interaksjon i DPG. Nå er veien lagt for å utvikle flere interaktive plugins. Å lage flere og mer avanserte plugins kan være en oppgave i nye prosjekt eller masteroppgaver.

10.1.3 Evaluere forskjellige søkealternativer

Det å finne en god søketeknologi viste seg å være en utfordring fordi de fleste aktuelle løsningene baserte seg på samme søketeknologi, Lucene. Det ble brukt en god del tid på å teste ut de forskjellige teknologiene slik at det kunne gjøres en skikkelig vurdering om de kunne brukes av DPG. Det var stor forskjell på de aktuelle løsningene til tross for at de alle var basert på Lucene. Det var likevel en kandidat som stod frem som et bunnsolid alternativ, Solr.

10.1.4 Implementere søk i DPG ut fra valgt søketeknologi

Solr er en separat applikasjon, dette gir både fordeler og ulemper. Det gjør installasjonsprosessen til DPG noe tyngre, ettersom det er to forskjellige applikasjoner som må konfigureres og settes opp. Fordelene er at man får en modulær løsning, slik at for eksempel indekseringsprosessen kan skiftes ut ved behov.

For selve implementasjonen ble det viktig å finne ut hvordan innhold skulle indekseres. Den endelige løsningen rendrerer alle mulige kombinasjoner av sider og utsnitt, tolker innholdet og indekserer dette. Dette gjør at en mister litt av fleksibiliten som tilbys av Solr, ettersom ingen felt kan defineres som for eksempel heltall, noe som ville gjort at en kunne utformet spørringer som tok hensyn til dette.

Indekseringen foregår nå både på bestemte tidspunkt, og ved endringer av innhold gjennom delsystemet PCE. Den sistenevnte indekseringsprosessen sørger for at indeksen til presentasjonen alltid vil innholde sist oppdaterte innhold lagt inn av en *publisher*. Indeksering på bestemte tidspunkter gjør at brukergenerert innhold også blir indeksert.

Det burde vært gjort en vurdering av søkeresultatene, men på grunn av at de eksisterende presentasjonene ikke inneholdt mye innhold, og utvikling av nye store presentasjonsmønster er en svært omfattende prosess, ble det vanskelig å få tid til å gjennomføre en slik evaluering. Av de gjennomførte søkene er inntrykket til forfatteren at søkemotoren fungerer veldig bra, men det burde vært gjort tester som dokumenterte blant annet presisjon og tilbakekall på forskjellige søkefraser. Dersom det vil utvikles flere presentasjonsmønstre mer større innholdsmengde, anbefales det å gjennomføre disse testene, slik at Solr-serverens innstillinger kan finjusteres ut fra resultatene.

10.1.5 Øke uttrykkskraften til presentasjonsmønsterspesifikasjonen

Den nye presentasjonsmønsterspesifikasjonen er nøye gjennomtenkt, og resultatet er en presis og helhetlig spesifikasjonen, Å lage nye presentasjonsmønstre er blitt en enklere prosess, som illustrert ved før- og etterlistingene i delkapittel 7.5. Spesifikasjonen er blitt mer effektiv med at man har fjernet unødvendige element, og den er mer helhetlig ettersom alle felt defineres på en helhetlig måte.

10.2 Vurdering av teknologier

10.2.1 Solr

Å sette opp selve Solr-serveren bød på en del problemer, men når brikkene falt på plass fungerte serveren som forventet. Dokumentasjonen var tilstrekkelig noe som førte til at problemer som oppstod ble enkle å løse. Solr-serveren er rask og stabil. Alt i alt har Solr vært en fryd å jobbe med, og kan anbefales til andre som ønsker å bygge søkefunksjonalitet inn i et innholdshåndteringssystem.

10.2.2 AJAX Solr

AJAX Solr er et rammeverk og har derfor en del begrensninger, men så lenge det finne en innretning for den ønskede funksjonaliteten er det bare å følge instruksene som er gitt på nettsiden til rammeverket for å få alt til å fungere.

10.3 Videre arbeid

10.3.1 Fjerne plugins som singleton

I pluginarkitekturen som ble utarbeidet i forbindelse med DPG 2.0 ble det konfigurert at en plugin er en singleton. Det gikk an å ha flere av samme plugin, og skille de fra hverandre ved å sette forskjellige parametere i konfigurasjonsfilen som nå heter `pluginConfig.xml`. Dette var en grei løsning for problemstillingene om multimedia, men ettersom pluginarkitekturen nå er utvidet, er det en bedre løsning at en istedenfor har ett objekt per definerte plugin i konfigurasjonsfilen `pluginConfig.xml`. Dette tillater at en kan tilknytte flere objekter av samme plugin, og likevel lagre data i instansvariablene. Hvis det i DPG 2.1 skal brukes flere stemmegivningsplugins, må det lages en plugin for hver avstemning, siden de ellers deler samme data.

For å løse dette må knytte sammen filen `pluginConfig` og klassen `PluginLoader` slik at det instansieres en ny instans for hver definerte plugin i filen `pluginConfig`.

10.3.2 Utvidelse med ny rolle - *anonymous reader*

I den opprinnelige oppgavebeskrivelsen som ble innsendt til instituttet var en del av oppgaven å kunne utvide systemet med en ny rolle, *anonymous reader*. Dette er en rolle som ikke krever innlogging. Dette betyr at en må unngå hele lobby-delen av DPG, og bli sendt direkte til en presentasjon. Dette er svært nyttig dersom presentasjonmønsteret beskriver en avis. Det er få som er villig til å logge inn på en side for å lese innholdet i en avis.

I forhold til JAFU-prosjektet hadde det vært nyttig å hatt en studiekatalog som viste hvilke kurs som var tilgjengelig på fjernundervisningen.

For å få dette til må det være mulig å gjøre at presentasjoner ikke krever autorisasjon.

10.3.3 Ferdigstilling og videreutvikling av interaksjonsplugins

De skisserte plugins i kapittel 6: Interaksjonsplugins har noe manglende funksjonalitet. Kommentarpluginet bør utvides slik at det er mulig å slette eller endre egne kommentarer, brukere med rollene `admin` eller `publisher` bør kunne slette eller endre alle eksisterende kommentarer. I tillegg bør kommentarer nøstes, slik at det kan kommenteres på en kommentar. Dette kan løses ved at pluginet genererer en lenke for sletting av kommentarer som brukeren selv har lagt inn. Dersom brukeren trykker på slett-lenken vil pluginet sjekke om brukeren har rettigheter til å slette kommentaren.

Stemmegivningspluginet kan forbedres ved å tillate *flervalgsalternativ* (eng *multiple choice*). Dette kan løses ved å ha dette med som en parameter til pluginet, og hvert svaralternativ

inkrementeres. Det bør også være mulighet til å ha prosentvise resultater til hvert svaralternativ, og det bør tilbys flere forskjellige diagramtyper, slik at det ikke er begrenset til å bare bruke kakediagram. Dette kan løses ved å ha dette med som en parameter til dette plugin. En annen svakhet er at en bruker kan lese av stemmen til en annen bruker ved å manipulere addressefeltet.

10.3.4 Interaksjon med andre sosiale medier

I dag tillater de fleste sosiale medier en kobling mellom dine nettsider og deres nettsider. Nettstedet Facebook introduserte nylig Åpen Graf-protokoll (eng. *Open Graph Protocol*) [16], i et forsøk på å ta et steg videre mot den semantisk veven (eng. *Semantic Web*) [64], en ide der innholdet på nettsider annoteres med hvilken type innhold den representerer. Ved å registrere en presentasjon i Facebooks åpen graf-protokoll er det mulig for brukere å skape en kobling til presentasjonen i facebook. Da er det mulig å sende ut en beskjed til alle brukere som har skapt en slik kobling når innholdet oppdateres. For å skape koblingen mellom presentasjon og facebook legges det enten inn en kodesnutt gjemt i en iframe-tag, eller ved å benytte et Javascript-bibliotek laget av Facebookutviklere som håndterer dette formålet. Det er enkelt å inkludere denne type JavaScript i DPG 2.1, mye takket være oppgaven til Peder Lång Skeidsvoll - Støtte for rike klienter i Dynamic Presentation Generator [65].

Ved å tillate en tettere kobling til sosiale medier er lettere å spre informasjonen som publiseres i DPG. Når et nytt element legges til i en entitetsliste, kan innholdet i det nye elementet automatisk publiseres til en Facebook, Twitter og andre sosiale medier. I Twitter benyttes det såkalte *hashtagger* (eng. *hashtags*) for å kategorisere innholdet. Dersom publisert innhold markeres med en slik hashtag er det mulig å spore innhold som er relatert til det publiserte innholdet. Dette gjør at twittermeldinger som innholder denne hashtaggen kan automatisk publiseres på siden der innholdet ble publisert i DPG.

10.4 Konklusjon

10.4.1 Personlige erfaringer

Arbeidet med å utvikle DPG 2.1 har gjort at forfatteren har blitt bedre kjent med en rekke teknologier som forfatteren kan bringe med seg videre i arbeidslivet. Det å delta i et større prosjekt i en gruppe med flere over lengre tid, har også vært en god og nyttig erfaring. På selve utviklingsfronten var det tidsvis en del treghet under utviklingen, og de gangene der test-drevet utvikling ikke ble brukt ble forfatteren ofte straffet, noe som er en erfaring å bemerke seg.

I kapittel 1.1 ble det nevnt at forfatteren ville bruke XP-prinsippene under utvikling, og dette er delvis fulgt opp. Prinsipper som refaktorering har blitt gjennomført både på den delen av eksisterende kodebase der det var nødvendig, og på egen kode. Som nevnt i forrige avsnitt ble også test-drevet utvikling brukt, men ikke i tilstrekkelig grad. Parprogrammering ble gjort sporadisk, etter som det var mer individuelt arbeid enn først antatt. Kontinuerlig integrasjon ble benyttet mot slutten av prosjektet ved hjelp av den kontinuerlige integrasjonserveren Hudson.

Av utviklingsteknologier har det stort sett vært problemfritt å jobbe med eclipse og subversion. De største problemene har oppstått på grunn av begrensninger satt på UiB sine maskiner. Brukere har blant annet ikke full tilgang, noe som gjør det vanskeligere å installere og prøve ny programvare. Den kontinuerlige utviklingsserveren Hudson kom med sent i prosjektet, og ble ikke en sentral del av utviklingen. Dersom den hadde vært med fra starten av ville utviklingen vært mer basert på tester, noe som hadde vært til det gode i det lange løp.

10.4.2 Om DPG og presentasjonsmønstre

Å jobbe med DPG og presentasjonsmønstre har vært interessant. Presentasjonsmønstersideen er en meget god idé, og ved å videreutvikle DPG tror forfatteren at DPG kan bli et kraftig verktøy for utvikling av vevsider som passer i til presentasjonsmønstre.

Svakhetene til DPG i versjon 2.1 og 2.0 er at det er såpass mange prosesser og teknologier som må gjennomgås og forstås for å kunne utvikle nye presentasjonsmønstre. Hvis det hadde vært mulig å automatisere deler av disse prosessene, hadde utvikling av nye mønstre blitt enklere. En løsning er å ha genererte standardløsninger i for eksempel Velocity og XSLT-transformasjonene, og dersom det ønskes mer avansert funksjonalitet kan standardløsningene overkjøres.

Det er iverksatt planer om å lage verktøy som gjør utvikling av nye mønstre til en enklere prosess, og en gang i fremtiden kan kanskje dette også gjøres av personer uten teknisk bakgrunn.

Tillegg A

Konfigurasjonsfil for Solr

A.1 Schema.xml

Filen *Schema.xml* er en konfigurasjonsfil som benyttes av Solr. Formålet med filen er å definere felt og felttyper, for så å koble disse sammen.

I denne filen defineres et skjema som kobler sammen hvilke filtre som skal brukes på forskjellige felt. Først defineres det forskjellige felttyper, hver felttype har en rekke filtre knyttet til seg. Deretter defineres feltene, og hvert felt knyttes opp mot en felttype. Mot slutten av filen defineres det hvilken operator som skal benyttes dersom serveren mottar forespørsler med flere søkeord, og hvilket felt som sørger for dokumentenes unikhet.

Listing A.1: Konfigurasjonsfilen Schema.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<schema name="dpg" version="1.2">
  <types>
    <!-- The StrField type is not analyzed, but indexed/stored verbatim.
       - StrField and TextField support an optional compressThreshold which
       limits compression (if enabled in the derived fields) to values
           which
       exceed a certain size (in characters).
    -->
    <fieldType name="string" class="solr.StrField" sortMissingLast="true"
        omitNorms="true"/>

    <!-- A text field that uses WordDelimiterFilter to enable splitting and
         matching of
        words on case-change, alpha numeric boundaries, and non-
            alphanumeric chars,
```

```
        so that a query of "wifi" or "wi fi" could match a document
            containing "Wi-Fi".
        Synonyms and stopwords are customized by external files, and
            stemming is enabled.
        -->
    <fieldType name="text" class="solr.TextField" positionIncrementGap="100
        ">
      <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <!-- Case insensitive stop word removal.
          add enablePositionIncrements=true in both the index and query
          analyzers to leave a 'gap' for more accurate phrase queries.
        -->
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="stopwords.txt"
                enablePositionIncrements="true"
                />
        <filter class="solr.WordDelimiterFilterFactory" generateWordParts="
            1" generateNumberParts="1" catenateWords="1" catenateNumbers="1"
             catenateAll="0" splitOnCaseChange="1"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.SnowballPorterFilterFactory" language="
            Norwegian" protected="protwords.txt"/>
      </analyzer>

      <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
            ignoreCase="true" expand="true"/>
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="stopwords.txt"
                enablePositionIncrements="true"
                />
        <filter class="solr.WordDelimiterFilterFactory" generateWordParts="
            1" generateNumberParts="1" catenateWords="0" catenateNumbers="0"
             catenateAll="0" splitOnCaseChange="1"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.SnowballPorterFilterFactory" language="
            Norwegian" protected="protwords.txt"/>
      </analyzer>
    </fieldType>
  </types>

  <!-- Fields for the different components in the presentation pattern
      spesification -->
  <fields>
    <field name="url" type="string" indexed="true" stored="true" required="
        true" />
    <field name="presentationId" type="string" indexed="true" stored="true"
        required="true" />
    <field name="page" type="text" indexed="false" stored="true" required="
```

```
        true" />
     <field name="view" type="text" indexed="false" stored="true" required="
        true" />
     <field name="content" type="text" indexed="true" stored="true" required=
        "true" />


     <!-- catchall field, containing all other searchable text fields (
        implemented
          via copyField further on in this schema  -->
     <field name="text" type="text" indexed="true" stored="false" multiValued
        ="true"/>

  </fields>

  <!-- Field to use to determine and enforce document uniqueness.
       Unless this field is marked with required="false", it will be a
          required field
    -->
  <uniqueKey>url</uniqueKey>

  <!-- field for the QueryParser to use when an explicit fieldname is absent
      -->
  <defaultSearchField>text</defaultSearchField>

  <!-- SolrQueryParser configuration: defaultOperator="AND|OR" -->
  <solrQueryParser defaultOperator="OR"/>

   <!-- copyField commands copy one field to another at the time a document
         is added to the index.  It's used either to index the same field
             differently,
         or to add multiple fields to the same field for easier/faster
             searching.  -->
   <copyField source="content" dest="text"/>
   <copyField source="page" dest="text"/>
   <copyField source="view" dest="text"/>
 </schema>
```

Tillegg B

Velocity mal for søk.

B.1 searchPageTemplate.vm

Dette tillegget viser malfilen i Velocity som må benyttes for å vise søkeboks og eventuelle søkeresultater. Det vises maksimalt 10 treff per side. Malen lister opp treffene. I tillegg får en vite hvilke treff som er hentet (for eksempel treff 10-20), hvor mange treff det var totalt, og lignende. Hvis det ikke er noen treff, får brukeren beskjed om dette. Det forrige søkeordet vil dukke opp i søkefeltet.

Listing B.1: Velocity mal for søk

```
#if( $latestMessagesView )
  <div id="content" class="tredelt">
#elseif( $reviewQuestionsView )
  <div id="content" class="simple">
#else
  <div id="content" class="standard">
#end

<div class="left">
  ${viewMenuContent}
  ${navigationBarContent}
</div>
<div class="middle">
  <div id="main-text">
    <div class="content">

      <h2>Sok!</h2>

      #set ( $currentNavigationPage = $session.getAttribute( "
          currentNavigationPage_${pid}" ) )
```

```
#set ( $currentSearchWord = $session.getAttribute( "
    currentSearchWord_${pid}" ) )
#set ( $searchResultString = "searchResults_${pid}" )
#set ( $listOfHits = $session.getAttribute( $searchResultString ).
    getHitBeans() )
#set ( $numberOfHits = $session.getAttribute( $searchResultString ).
    getTotalNumberOfHits() )
#set ( $from = ($currentNavigationPage * 10)+1 )
#set ( $to = $listOfHits.size() + ($currentNavigationPage * 10) )

  <form name="formSubmit" action="search.html" method="get">
    <input type="hidden" name="pid" value="${pid}" />
    <input type="hidden" name="page" value="searchPage" />
    <span id="localSearch">
    #if ($currentSearchWord)
      <input type="text" id="query" name="searchWord" value="${
          currentSearchWord}"/>
    #else
      <input type="text" id="query" name="searchWord" />
    #end
    </span>
    <input type="submit" value="Sok!" />
  </form>


#if (${numberOfHits} > 0)
  <p>Viser treff <strong>$from</strong> til <strong>$to</strong> av
      totalt <strong>$numberOfHits</strong> treff som inneholder <
      strong>${currentSearchWord}</strong>. </p>
#else
  #if ( ${currentSearchWord} )
    <p>Fant ingen treff pa sokeordet ${currentSearchWord}.</p>
  #else
    <p>Begynn soket ved a skrive inn s keord i s kefeltet, og trykk
    S k -knappen.</p> #end
#end

#if( $searchView )
  #foreach($hit in $listOfHits)
    #set ( $velocityvalue = $velocityCount + 10 * ${
        currentNavigationPage} )
      <p id="searchHit">
        <h3><a href="$hit.getUrl()">$hit.getPage() -> $hit.getView()
            </a></h3>
        <div class="searchResult">$hit.getContext()</div>
        <div class="searchResult"><span id="searchLink">$hit.getUrl()
            </span></div>
      </p>
  #end
#else
  <p>Du har ikke sokt  p   noe enda.</p>
#end
```

```
        #if ($numberOfHits > 10)
          <p id="pageNavigation">Sidenavigering:
          #foreach ($page in $session.getAttribute( "navigationPages_${pid}"
              ))
            $page
          #end
          </p>
        #end

        </div>

      </div>
      <br/>
      <br/>
    </div>

    <div class="right">

      <div class="uib-module calendar">
        <h3>Siste meldinger</h3>
        <div class="content">
              #if( $latestMessagesView )
               ${latestMessagesView}
          #end
        </div>
      </div>
  </div>
  </div>
```

Bibliografi

[1] Snowball. http://snowball.tartarus.org/, 04 2010.

[2] Adobe. Adobe flash. http://www.adobe.com/flashplatform.

[3] Scott W. Ambler. Mapping objects to relational databases: O/r mapping in detail. http://www.agiledata.org/essays/mappingObjects.html, 2010.

[4] Karianne Berg. Persistensproblematikk i dynamic presentation generator, 2008.

[5] Emmanuel Bernard and John Griffin. *Hibernate Search In Action*. Manning, 2009.

[6] N. Borenstein. Mime (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.1862, 1992.

[7] Jan Bruusgaard. Norwegian stop word list. http://snowball.tartarus.org/algorithms/norwegian/stop.txt, 2005.

[8] Contentmanager.eu.com. What is a content management system, or cms? http://www.contentmanager.eu.com/history.htm, 2008.

[9] Microsoft Corporation. Microsoft silverlight. http://www.microsoft.com/silverlight.

[10] Kevin Cruickshanks. Verkty for generering av xml-baserte presentasjonar: Jpgen - java presentasjons generator, 2004.

[11] Dagbladet. Dagbladet.no. http://www.dagbladet.no/, 2010.

[12] The Spring Framework Reference Documentation. Scheduling and thread pooling. http://static.springsource.org/spring/docs/2.0.x/reference/scheduling.html, 2004.

[13] Fast employees. *Book of Search*. Fast Search Transfer ASA, 2006.

[14] Yngve Espelid. Dynamic presentation generator, 2004.

[15] evolvingweb. Ajax solr. http://github.com/evolvingweb/ajax-solr.

[16] Facebook. Model-view-controller. http://opengraphprotocol.org.

[17] Facebook. Facebook. http://www.facebook.com/, 2010.

[18] Flickr. Flickr. http://www.flickr.com/, 2010.

[19] Apache Software Foundation. Apache jackrabbit. http://jackrabbit.apache.org/.

[20] Apache Software Foundation. Apache lucene. http://lucene.apache.org.

[21] Apache Software Foundation. Apache nutch. http://lucene.apache.org/nutch/.

[22] Apache Software Foundation. Apache nutch authentication schemes. http://wiki.apache.org/nutch/HttpAuthenticationSchemes.

[23] Apache Software Foundation. Apache nutch post authentication. http://wiki.apache.org/nutch/HttpPostAuthentication.

[24] Apache Software Foundation. Apache solr. http://lucene.apache.org/solr/.

[25] Apache Software Foundation. Apache subversion. http://subversion.apache.org/.

[26] Apache Software Foundation. Apache tika - innholdsanalyseverktøy. http://lucene.apache.org/tika/.

[27] Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/.

[28] Apache Software Foundation. Search implementation. http://jackrabbit.apache.org/search-implementation.html.

[29] Apache Software Foundation. Solrj wikiside. http://wiki.apache.org/solr/Solrj.

[30] Apache Software Foundation. Apache logging services - log4j. http://logging.apache.org/log4j/index.html, 2010.

[31] Apache Software Foundation. The apache velocity project. http://velocity.apache.org/, 2010.

[32] The Eclipse Foundation. Eclipse. http://www.eclipse.org.

[33] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html.

[34] Martin Fowler. *Patterns of the Enterprise Application Architecture*. Pearson Education, Inc, 2003.

[35] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004.

[36] Fxdteam. Last.fm tag cloud. `http://fxdteam.com/lastcloud/index.php?act=new`, 2010.

[37] Erik Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[38] Verdens Gang. Vg nett. `http://www.vg.no/`, 2010.

[39] Andreas Girgensohn and Alison Lee. Making web sites be places for social interaction. 2002.

[40] Google. Google analytics. `http://analytics.google.com`, 2010.

[41] Google. Google chart tools. `http://code.google.com/apis/charttools/`, 2010.

[42] ICQ. Icq. `http://www.icq.com/`, 2010.

[43] Apple Inc. Quicktime. `http://www.apple.com/quicktime`.

[44] Google Inc. Google - søk og mer. `http://google.com/`.

[45] Sphinx Technologies Inc. Sphinx search. `http://sphinxsearch.com/`.

[46] Bjørn Ove Ingvaldsen. Multimedia i dynamisk presentasjons generator 2.0, 2008.

[47] Manfred Tscheligi Johann Schrammel, Michael Leitner. Semantically structured tag clouds: An empirical evaluation of clustered presentation approaches, 2009.

[48] Joomla! Joomla! - the dynamic portal engine and content management system. `http://www.joomla.org`, 2010.

[49] Last.fm. Last.fm. `http://last.fm`, 2010.

[50] Kristian Skønberg Løvik. Webucator 3.0 - brukerhåndtering og aksesskontroll for dpg 2.0, 2008.

[51] Robert C. Martin. *Clean Code*. Prentice Hall, 2009.

[52] Microsoft. Windows live messenger. `http://events.no.msn.com/messenger/`, 2010.

[53] Sun Microsystems. Code conventions for the java programming language. `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`, 1999.

[54] Khalid Mughal. *Annotations*. 2007.

[55] Khalid Azim Mughal. Presentation patterns, composing web-based presentations., 2003.

[56] Ira R. Forman og Nate Forman. *Java Reflection in Action*. Manning Publications Co., 2004.

[57] Oracle. Hudson - extensible continuous integration server. http://hudson-ci.org/.

[58] Oracle/Sun. Java content repository - spesifikasjon. http://jcp.org/aboutJava/communityprocess/final/jsr170/index.html.

[59] Erik Hatcher Otis Gospodnetic. *Lucene in Action*. Manning Publications Co., 2005.

[60] Open Source Project. Compass. http://www.compass-project.org/.

[61] Open Source Project. Xapian. http://xapian.org/.

[62] John Resig. jquery - javascript library. http://jquery.com.

[63] Bjørn-Kristian Sebak. Dynamic presentation generator 2.0 - utvikling av ny dynamisk presentasjonsgenerator og presentasjonsmønsterspesifikasjon, 2008.

[64] semanticweb.org. The semantic web. http://semanticweb.org/wiki/Main_Page.

[65] Peder Lång Skeidsvoll. Støtte for rike klienter i dpg, 2010.

[66] Amol Sonawane. Using apache lucene to search text. http://www.ibm.com/developerworks/java/library/os-apache-lucenesearch/index.html.

[67] Spring. Spring framework. http://www.springsource.org/, 2010.

[68] Terracotta. Quartz-schedueler. http://www.quartz-scheduler.org/.

[69] Bill Thompson. Is google good for you? http://news.bbc.co.uk/2/hi/technology/3334531.stm.

[70] IETF Tools. Internet relay chat protocol. http://tools.ietf.org/html/rfc1459, 1993.

[71] Twitter. Twitter. http://www.twitter.com/, 2010.

[72] Universitetet i Bergen. *Rossini, Alessandro and Liberati, Graziano*. Institutt for informatikk, 2006.

[73] Usenet. Usenet - user network. http://www.usenet.com/usenet.html, 2010.

[74] W3C. Html 4.01 specification. http://www.w3.org/TR/REC-html40/.

[75] W3C. Xhtml™ 1.0 the extensible hypertext markup language (second edition). http://www.w3.org/TR/xhtml1/.

[76] World Wide Web Consortium (W3C). Xsl transformations (xslt) - version 1.0. http://www.w3.org/TR/xslt, 1999.

[77] Wikipedia. Web 2.0. http://en.wikipedia.org/wiki/Web_2.0.

[78] Wikipedia. Model-view-controller. http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller, 2010.

[79] Wordpress. Wordpress > blog tool and publishing platform. http://wordpress.org/, 2010.