UNIVERSITETET I OSLO
Institutt for informatikk

Videreutvikling av innleveringssystemet Joly

Masteroppgave

Marius Syverstad Fossum

12. november 2012

Forord

Jeg vil spesielt få takke min veileder Arne Maus for hans bidrag i diskusjoner og konstruktive tilbakemeldinger gjennom hele skriveprosessen.

Jeg vil også takke familie, venner og medstudenter som har vært støttende og hjelpsomme gjennom studietiden.

Sammendrag

Det å vedlikeholde systemer som tar i bruk åpen kildekode rammeverk viste seg å være krevende. I denne oppgaven har vi videreutviklet Joly med nye funksjonaliter hvor dette hovedsakelig dreier seg om å benytte Joly som et godkjentlistesystem. For å kunne gjøre dette var det nødvendig å forstå og oppdatere Joly, for så å videreutvikle med ny funksjonalitet. Å sette seg inn i et større system kan vise seg tidskrevende og innebære mye arbeid, men som er nødvendig for å kunne videreutvikle. Gjennom oppgaven vil det komme fram at bruken av åpen kildekode rammeverk medfører mer uforutsett vedlikeholdsarbeid enn man skulle forvente. Dette gjelder nødvendigheten for hyppig vedlikehold av systemer siden disse rammeverkene oppdateres og videreutvikles over tid. Dette er noe som systemer som tar i bruk disse rammeverkene må tilpasse seg, da det er ønskelig å kunne benytte siste oppdaterte versjoner av rammeverkene hvis disse adresserer et sikkerhetsproblem som vil kunne være skadelig for systemet om det ikke oppdateres. Deler av den nye funksjonaliteten som ble lagt inn i Joly er at gruppelærere kan godkjenne innleverte obligatoriske oppgaver fra studenter. Dette danner grunnlaget for generering av en godkjentliste over studenter som har bestått alle obligatoriske oppgavene i INF1000. Som dermed kan overleveres til det matematisk naturvitenskapelig fakultet slik at det er mulig å vite hvem som kan få gå opp til eksamen.

Innhold

Figurer

Tabeller

Kapittel 1

Introduksjon

I denne masteroppgaven skal vi se på hvordan det er å videreutvikle et system. Dette gjelder innleveringssystemet Joly som er et distribuert system for 500-600 brukere, som hviler på en rekke åpen kildekode rammeverk og verktøy som hovedsakelig er; Spring, Hibernate, MySQL og Jetty. Joly benyttes i forbindelse med nybegynnerkurset i java INF1000 ved Universitetet i Oslo. Studentene leverer via Joly og det utføres fuskesjekk mellom tidligere innleverte obligatoriske oppgaver. Fuskesjekken skiller Joly fra andre innleveringssystemer som benyttes på Universitetet i Oslo, og den forbindelse skal Joly videreutvikles til å fungere som et godkjentlistesystem.

For å kunne videreutvikle Joly var det nødvendig å gå igjennom en stegvis prosess som inkluderte å forstå og sette seg inn i Joly, spesifisere ny funksjonalitet, oppdatere rammeverk og verktøy og rettinger av eksisterende kildekode. Deretter implementere ny funksjonalitet og til slutt se hvilken effekt bruken av åpen kildekode rammeverk har på utviklingsprosessen av applikasjoner. Ettersom Joly ikke har vært vedlikeholdt de siste to årene var stegene som omhandlet oppdateringer og rettinger svært nødvendige.

1.1 Motivasjon

Joly har vært et velfungerende innleveringssystem ved Universitet i Oslo de tidligere årene. Men Joly manglet en form for godkjentliste funksjonalitet som andre konkurrerende systemer som er i drift på universitetet som f.eks. Devilry som er et innleverings- og godkjentsystem. Ikke minst var det viktig å få Joly videre-

utviklet, oppdatert og klart for drift til høsten 2012 hvor det skulle tas i bruk i INF1000. Joly ble valgt til fordel for Devilry for høsten 2012, fordi ved å implementere godkjentliste funksjonalitet så ville Joly kunne utføre lignende oppgaver som Devilry, men vil også ha fuskesjekk for innleveringer.

Med Joly som godkjentlistesystem skal gruppelærerne kunne enklere godkjenne innleverte obligatoriske oppgaver fra studenter. Det skal så kunne genereres en godkjentliste på slutten av semesteret som inneholder alle studenter som har bestått alle obligatoriske oppgaver i kurset INF1000 for det semesteret. Matematisk-naturvitenskapelige fakultet behøver denne listen for å vite hvem som har lov til å gå opp til eksamen på slutten av semesteret.

1.2 Oppsummering av kapitler

Kapittel 2

I dette kapitlet beskriver vi innleveringssystemet Joly i sin helhet, noe som hovedsakelig vil bestå av en detaljert beskrivelse av de største rammeverkene og en forklaring på hvilke verktøy som benyttes og ikke minst hvorfor de er nødvendige. Deretter gis en oversikt over hvordan disse kommuniserer med hverandre.

Kapittel 3

Med forståelsen og et overordnet blikk over hva som utgjør Joly som system, går vi her over til å skissere og forklare planlagte funksjonaliteter for å kunne gjøre Joly til å fungere som et godkjentlistesystem. Dette vil inkludere nye rapporter og potensielt nye sikkerhetstiltak.

Kapittel 4

Her fortsetter vi med å gjøre Joly klar for videre utvikling. Det blir gjort nødvendige oppdateringer når det gjelder rammeverk og verktøy som tas i bruk i Joly. Dette medfører også endringer i kildekoden til Joly ettersom oppdaterte rammeverk har endret deler av sin virkemåte.

Kapittel 5

I dette kapitlet gis en nærmere beskrivelse over hvilken funksjonalitet som til slutt ble implementert og hvordan dette ble gjort, og om det var mulig å få implementert dette i tide fram mot semesterstart. Det vil også foreligge detaljer rundt problemer som oppstod ved hver av funksjonalitetene og hvordan disse ble løst. Til slutt beskriver vi hvilket arbeid som bør gjøres, men som det ikke var tid til.

Kapittel 6

Avsluttende trekker vi fram hvordan åpen kildekode rammeverk påvirker utviklingsprosessen hele veien. Dette inkluderer da arbeidet rundt å vedlikeholde et system som benytter seg av åpen kildekode rammeverk, samt det å komme med ny funksjonalitet til et eksisterende system som benytter seg av slike rammeverk. Til slutt tar jeg opp erfaringer ved det å måtte lære seg å bruke disse åpen kildekode rammeverkene.

Kapittel 2

Innleveringssystemet Joly

I dette kapitlet skal vi se på historien bak Joly for å vise hvordan systemet har utviklet seg over tid. Dette vil innebære hva de eldre og nyere versjonene av Joly inneholder, i tillegg til tidligere og nåværende arbeid. Vi skal også se på rammeverkene og verktøyene som benyttes i Joly, og kommunikasjonen mellom de for å vise hvorfor de er nødvendige.

2.1 Tidligere og nåværende arbeid

Joly er et web-basert innleveringssystem som ble utviklet tilbake i 2006 av Hanne Vibekk og Therese Steensen[21]. Christian Kringstad Kielland[12] sto for algoritmen som først ble benyttet til fuskesjekk og brukergrensesnittet. Joly ble så tatt i bruk i nybegynnerkurset i Java (INF1000) ved Universitetet i Oslo. Med mulighet til å ta de som har jukset, som kan være studenter som har kopiert hele eller deler av en obligatorisk oppgave til en annen student. De obligatoriske oppgavene blir levert gjennom Joly og sammenlignet med alle andre eksisterende besvarelser i databasen på innleveringstidspunktet. Hvis en obligatorisk oppgave er for lik en annen obligatorisk oppgave vil det bli gitt beskjed til gruppelæreren for gruppen student var oppmeldt, samt foreleser som da kunne videre se på saken.

Senere viste det seg et behov for vedlikehold av Joly, noe Cato Morholt sto for. Han rettet opp diverse mindre problemer som var eksisterende i Joly. I tillegg til dette utviklet Jose Luis Rojas en ny algoritme som skulle kunne oppdage forsøk på juks bedre, og som også ville vise prosentvis likhetsgrad mellom andre obligatoriske oppgaver. Hovedgrunnen til dette var at i de omtendighetene hvor juks

ble oppdaget var det enklere overfor resten av universitetet å begrunne det ut ifra hvor prosentvis like besvarelsene var linje for linje.

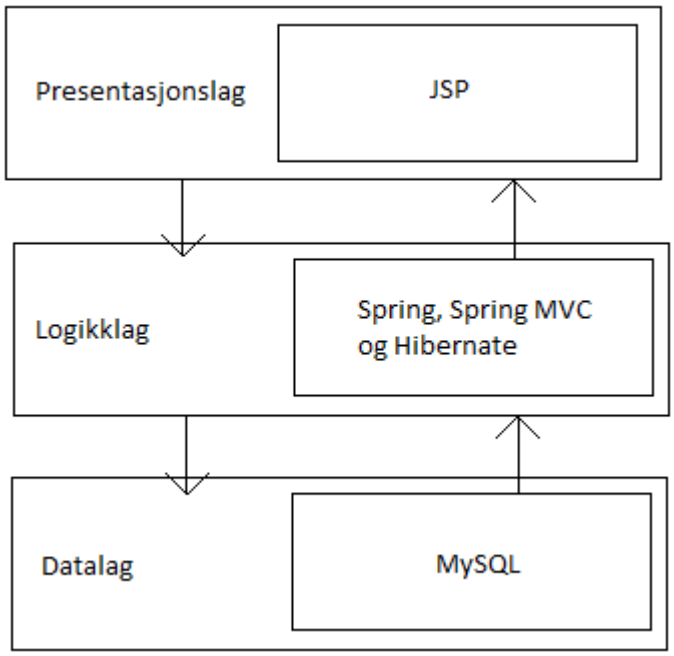
Etter en god stund ble det startet vedlikehold nok engang av Arne Skjærholt som innebar omstrukteringer og skifte over fra byggeverktøyet Apache Ant til maven. Det kom ingen fungerende versjon ut av dette arbeidet. Igjen er Joly under arbeid, i denne omgang er det Siamek Darisiro[5] som jobber med dette. Oppgaven hans er å oppdatere rammeverkene og verktøyene som blir brukt i forbindelse med Joly slik at det er opp til en viss standard.

Siden det eksisterer flere versjoner i omløp finner vi det nødvendig for ordens skyld å klare opp i hvilke disse er. Det begynner med første versjon av Joly utviklet av Hanne Vibekk og Therese Steensen som vil bli referert til som versjon 1.0. Versjon 1.5 vil være den som ble videreutviklet av Cato Morholt og Jose Luis Rojas. Til slutt har vi det påbegynte arbeidet med Joly av Siamek Darisiro som vil være versjon 1.6.

2.2 Joly sin arkitektur

Joly benytter seg av en såkalt trelagsstruktur. Dette innebærer at applikasjonen er delt opp lagvis for å skille mellom de forskjellige oppgavene som skal utføres i applikasjonen[26]. Her skiller vi mellom prestasjons-, logikk- og datalaget. Presentasjonslaget har ansvaret for å produsere og representere brukergrensnittet som brukeren vil se ved bruk av en nettleser. På logikklaget ligger implementasjen av metoder, funksjoner og spørringer mot databasen. Nederst i datalaget finner vi lagrede data som oftest vil være en database. Fordelene man ser med en slik lagfordeling er at det vil være veldig mye enklere å bytte ut et lag med en alternativ implementasjon uten å måtte endre hele applikasjonen. Dette gir økt gjenbruk av kode ettersom kun en mindre del av applikasjonen må endres. Hvis endringer skulle forekomme, om det skulle være spesifikasjoner for applikasjonen eller teknologiske framskritt så vil man forhåpentligvis unngå å lage en helt ny applikasjon.

Vi har tidligere nevnt at Joly benytter seg av en trelagsarkitektur, og som det er mulig å se av Figur 2.1 viser til hvilke verktøy som brukes hvor. På presentasjonslaget har vi JSP (Java Server Pages) som gir oss mulighet til visning av



Figur 2.1: *Trelagsstrukturen i Joly*

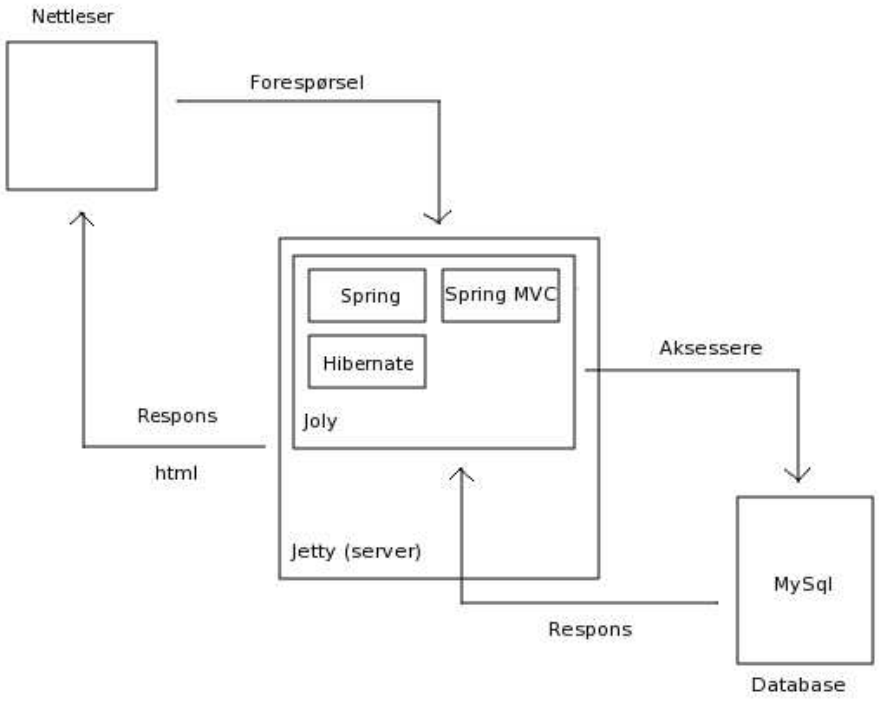
dynamisk innhold. Hvor så JSP formidler data fram og tilbake mellom logikklaget så har vi følgende; Spring Framework, Spring MVC og Hibernate. Til slutt er det datalaget hvor vi finner relasjonsdatabasen MySQL for lagring av data.

2.3 Åpen kildekode rammeverk og verktøy

Joly tar i bruk noen åpen kildekode rammeverk og verktøy som vi skal beskrive snart. Vi skal se på hva åpen kildekode rammeverk er og hva de generelt kan gjøre for oss. Først må vi forklare hva åpen kildekode programvare er. Åpen kildekode programvare er programvare som har kildekoden tilgjengelig for allmennheten. Slik programvare er gratis å ta i bruk og er lov å modifisere og kan muligens distribueres videre om lisensen for åpen kildekode programvaren tillatter dette[27]. Dermed er åpen kildekode rammeverk programvare som kan benyttes slik man vil under visse lisenser som begrenser hva som er lov. Åpen kildekode rammeverk består av kode og/eller biblioteker som tilbyr funksjonalitet som generelt gjelder for de fleste applikasjoner[6]. Med et slikt rammeverk slipper man å skrive kode som ofte tas i bruk, flere ganger. Dette vil derfor kunne potensielt kutte ned på

utviklingstiden for applikasjoner. Å ta i bruk slike rammeverk medfører stort sett at det er en programmeringstil og måte som må følges for at det skal fungere.

Vi har allerede vært inne på hvordan disse rammeverkene og verktøyene fordeler seg i arkitekturen tidligere, men ved å se på Figur 2.2 vil vi kunne se Joly fra et annet perspektiv som inkluderer alle de større rammeverk og verktøy. Figuren viser til hvordan samspillet mellom rammeverk og verktøy, hvor det er mulig å se at Jetty som web-server har oppgaven med å gjøre Joly tilgjengelig på web, motta forespørsler og håndtere deretter. Vi ser så at Spring, Spring MVC og Hibernate ligger som grunnlaget for Joly. Joly vil så generere respons til nettleseren avhengig av hvilken funksjonalitet som tas i bruk. Joly som applikasjon har full mulighet til å aksessere MySQL databasen etter behov.



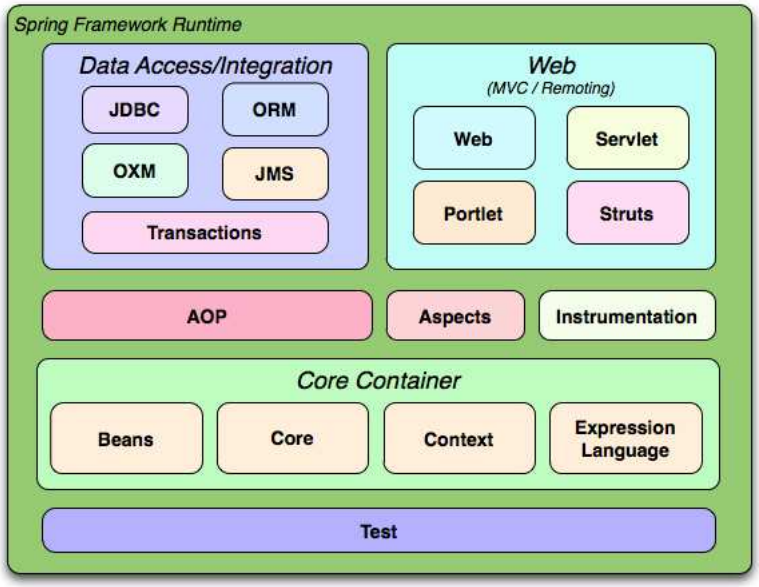
Figur 2.2: *Samspillet mellom rammeverk og verktøy brukt i Joly*

Vi vil nå se på de rammeverkene og verktøyene som Joly benytter i større detalj. Dette vil innebære hva det gjør og hvorfor vi trenger de forskjellige verktøyene. Ettersom Spring og Spring Web MVC er mer omfattende enn de andre rammeverkene og verktøyene vil disse bli dekket i større grad for å fremheve de

største egenskapene og fordelene de kan bringe til applikasjoner.

2.3.1 Spring Framework

Spring framework er et åpen kildekode rammeverk under Apache license 2.0[18] som tilbyr en forståelig infrastruktur for å utvikle java applikasjoner. Spring tar hånd om infrastrukturen slik at man som utvikler kan fokusere på logikken til applikasjonen[11]. Spring tilbyr dette gjennom bl.a. Dependency Injection for løsere koblinger mellom klasser, i tillegg er det støtte for AOP (Aspect-Oriented Programming), både proxy- og AspectJ-baserte varianter og god støtte med mot andre åpen kildekode rammeverk som f.eks. Hibernate. Selv om Spring blir brukt vil ikke hovedlogikken-koden bære noe preg av å være utviklet under Spring, som betyr at denne koden vil fungere på egen hånd. Det tilbys også et fleksibelt webrammeverk for utvikling av MVC (Model-View-Controller) applikasjoner. Spring er også modulær slik at det er mulig å velge fritt mellom hvilke moduler man vil ta i bruk. Spring består av 20 moduler som vist på figur 2.3, fordelt på følgende 5 hovedgrupper[11]; Core Container, Data Access/Integration, Web, AOP and Instrumentation og Test.



Figur 2.3: *Moduler i Spring Framework*

Joly benytter en god del av dem, men ikke alle. De som utmerker seg mest i

en applikasjon som Joly er hele hovedgruppen *Core Container* og *Servlet* som er en del av hovedgruppen *Web*, dermed vil disse bli dekket i større grad litt senere. ORM (Object-relational mapping) modulen gjør det mulig for Spring å støtte integrering med objekt-relasjon mapping verktøy slik som; Hibernate, JDO og iBatis for å nevne noen. Med *Web* modulen er det mulig å benytte Spring sine egenskaper i et web utviklingsmiljø. AOP modulen tillatter bruk av dette sammen med Spring, noe som gjør det enklere f.eks. å implementere metoder som kalles ved hvert databasekall slik at det er mulig å føre logg over dette. *JDBC* (Java Database Connectivity) modulen tilbyr et abstraksjonslag som gjør at det ikke er nødvendig å gjøre all *JDBC* programmering selv, eksempler på oppgaver som Spring dermed gjør for oss er; Åpne/lukke kommunikasjon med databasen, håndtere transaksjoner og unntakshåndtering (exception handling). Mens vi selv må bl.a ordne parametere for kommunikasjon og spesifisere hver SQL spørring[10]. Ved å se på hva Spring består av ser vi at Joly ikke tar i bruk alt Spring har å tilby. Joly bruker hoveddelen som kjennetegner Spring som er Beans, Inversion of Control og Spring MVC.

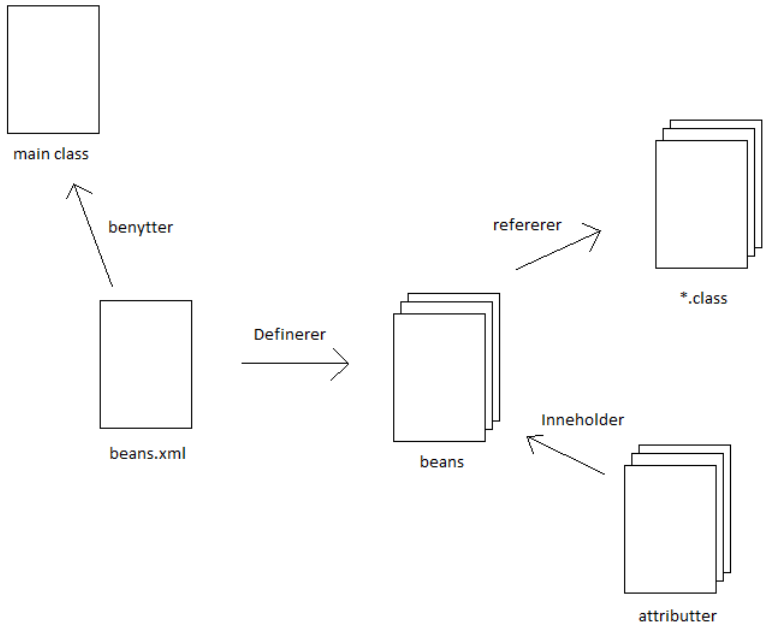
Inversion of Control

Siden en av fordelene ved å bruke Spring er Inversion of Control skal vi se nærmere på hva det er og komme med et eksempel på hvordan det brukes. Inversion of Control er objektorientert programmeringsmetode som går ut på å ordne relasjonene mellom objektklasser ved kjøring istedenfor for at det gjøres ved kompilering ved bruk av statisk analyse[24]. Inversion of Control går også under navnet Dependency Injection ettersom objektene blir opprettet og injisert via konstruktørargumenter.

Ved å ta i bruk Inversion of Control vil det også være lettere å bytte implementasjoner ettersom avhengighetene er definert i en konfigurasjonsfil. Dette lar seg gjøre fordi i stedet for at det er en kobling mellom implementasjoner så det er en kobling mot abstraksjoner i form av grensesnitt (interface) som implementasjonene må følge. Med andre ord så må programmet inneholde et sett allerede definerte metoder fra grensesnitt, men hvordan metodene blir implementert har ingenting å si.

Klassene spesifisert i denne filen kalles beans. På denne måten er det enklere å

gjøre endringer i parametere og relasjonene mellom disse klassene ettersom man kun behøver å forholde seg til en konfigurasjonsfil. Figur 2.4 viser hvor beans havner når det gjelder ved bruk i applikasjoner. Her er det mulig å se at beans representerer en klasse, og at beans blir konfigurert ved bruk av attributter som gjenspeiler hvilke andre klasser eller variabler den har relasjoner til.



Figur 2.4: *Dependency Injection med Spring*

Vi skal nå se nærmere på et eksempel over bruk av Inversion of Control med Spring, hvor vi går i dypere detalj enn tidligere på hvordan denne programmeringsmetoden kan gjøre jobben enklere. Den tar i bruk en konfigurasjonsfil som oftest går under navnet beans.xml. Denne filen inneholder "beans", som er linket til en klasse og tilhørende attributter. Eksempel på en slik fil er vises i Listing 2.1. Her vil *bean id* være hvordan main-klassen gjenkjenner denne beanen og brukes også hvis en annen bean benytter seg av denne beanen vil det refereres til denne id'en.

Listing 2.1: Eksempel på en konfigurasjonsfil

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 xsi:schemaLocation=" http://www.springframework.org/schema/beans
4  http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
5
6  <bean id="helloWorld" class="org.example.HelloWorld">
7   <property name="message" value="Hello World!"></property>
8  </bean>
9 </beans>
```

I Listing 2.2 kommer det fram Helloworld.java som det refereres til i beans.xml. Det viktigste å merke seg her er at istedenfor at String message er satt i HelloWorld.java, så har den en metode setMessage som settes under kjøring av Spring. helloworld variablen definert i beans.xml inneholder attributter med medfølgende data som blir injisert i applikasjonen når den eksekveres. Hvis man vil endre message variablen så behøver man ikke gjøre noe i HelloWorld.java, men gjøre det i beans.xml hvor den får dataene fra. Dette betyr at i stedet for å ha behov for å lete igjennom hele kildekoden for å endre en variabel som mest sannsynlig vil være nødvendig å endre over tid kan det gjøres i denne konfigurasjonsfilen. Det er derfor ikke nødvendig å rekompilere kildekoden ettersom variabelen bli injisert fra konfigurasjonsfilen. Nå er riktignok dette eksemplet kun for å illustrere hvordan Dependency Injection fungerer, men i en større applikasjon som Joly brukes dette til å sette relasjoner mellom klasser og hvilke klasser de er avhengig av. I stedet for at det er String variabel, så kunne det vært en klasse som kan over tid implementeres på forskjellige måter.

Listing 2.2: Klassen HelloWorld.java

```
1 public class HelloWorld {
2  private String message;
3
4  public void setMessage(String message) {
5   this.message = message;
6  }
7
8  public void display() {
9   System.out.println(message);
10  }
11 }
```

Det er så nødvendig i javakode å fortelle applikasjonen hvor den finner alle beans som kan gjøres slik det er vist i Listing 2.3 som stort sett gjøres i mainklassen. Main-klassen leser så konfigurasjonsfilen og kan benytte seg av disse klassene definert som beans. Dette er kun nødvendig å gjøre ved oppstart og hvis det er feil i konfigurasjonsfilen beans.xml vil ikke applikasjonen kjøre.

Listing 2.3: Klassen HelloWorldApp.java

```
1 public class HelloWorldApp {
2  public static void main(String[] args) {
3   ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml
        ");
4   HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorld");
5   helloWorld.display();
6  }
7 }
```

Dette var kun et lite eksempel på å vise til hvordan det enkelt er mulig å endre variabler uten at det har effekt på implementasjonen av logikken.

2.3.2 Spring Web MVC framework

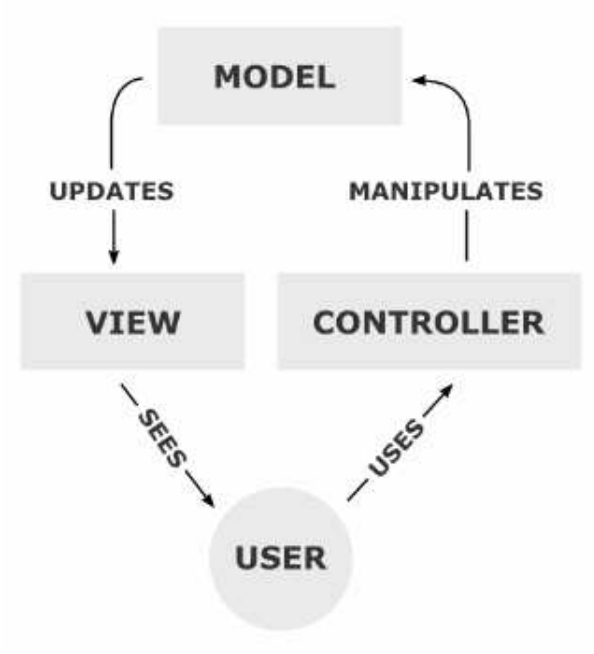
Spring MVC brukes til å utvikle web applikasjoner som kan benytte seg av bl.a. Inversion of Control som Spring bringer med seg. Spring MVC bygger på Model-View-Controller arkitektur som separarer programmet i forskjellige deler[25]; Model, View og Controller. Som det er mulig å se ut ifra Figur 2.5 viser den hvordan kommunikasjonen er mellom de.

Siden vi bruker Spring MVC tar vi i bruk deler av Spring og benytter oss av en konfigurasjonsfil som er relatert til deklarering av beans og Dependency Injection. For å se nærmere på dette går vi videre til å se på en bean som er deklarert i konfigurasjonsfilen Joly-servlet.xml for Joly. En bean er kun et objekt som blir injisert inn i applikasjonen ved oppstart. I Listing 2.4 ser vi en deklarering av en bean i Joly.

Listing 2.4: Konfigurasjonsfilen joly-servlet.xml

```
1 <bean id="AddEmployeeController" class="uio.ifi.joly.web.
      AddEmployeeController">
2  <property name="joly" ref="jolyFacade"/>
3  <property name="commandName" value="Employee"/>
4  <property name="formView" value="jolyadmin/AddEmployee"/>
5  <property name="successView" value="jolyadmin/AddedEmployee"/>
6  <property name="validator" ref="employeeValidator"/>
7 </bean>
```

Den blir gitt en id som vil være referansen den har om den skulle bli brukt av andre beans. Den blir så linket til en klasse og peker til hvor denne klassen ligger



Figur 2.5: *Model-Controller-View Diagram*

i filstrukturen. I tillegg til dette innehar klassen variabler som også kan settes via denne konfigurasjonsfilen. Den eneste variabelen i dette tilfellet som ikke er relatert til Spring er variabelen ved navn *joly*, denne sier noe om at i AddEmployeeController eksisterer det en variabel ved navn *joly* og at denne skal settes til referanseverdien jolyFacade som tidligere har blitt deklarert som bean. Dette krever en *setter* metode slik at Spring får gjort jobben sin og injisert objektet.

Så kommer dataene som er relatert til hvilken kontrollerklasse som benyttes. I dette tilfellet benyttes en SimpleFormController. Variabelen *commandName* er navnet på objektet som kan bli aksessert via en JSP som kontrolleren setter opp, som betyr at siden det er snakk om et Employee objekt vil det være naturlig å kunne aksessere variabler som f.eks. Employee.name. formView og succesView er ganske like i den forstand av at de spesifiserer en JSP som verdi. formView vil være den JSPen som vil bli returnert når en prøver å aksessere AddEmployeeControlleren via nettleser. Siden det her benyttes SimpleFormController, noe som for øvrig brukes i mesterparten av Joly, så innehar den metoden som kommer fram i Listing 2.5.

Listing 2.5: Metode fra klassen AddEmployeeController.java

```
1 protected ModelAndView onSubmit(HttpServletRequest request,
      HttpServletResponse response,
2   Object command, BindException errors) throws Exception {
3   return new ModelAndView(getSuccessView(),"Assignment", assignment);
4 }
```

Denne fungerer slik at når et skjema i JSPen blir sendt inn blir denne metoden kalt. I dette tilfellet vil succesView verdien være den JSPen man blir sendt til. Her er formView og successView like så det gjør ingen forskjell, men det er fullt mulig å spesifisere en annen side om ønskelig. Siste variabelen *validator* referer til en klasse som er brukt til å validere data brukeren har tastet inn i brukergrensesnittet. Denne vil sjekke f.eks. at ingen felter er tomme eller et maksimalt antall bokstaver i et gitt felt osv.

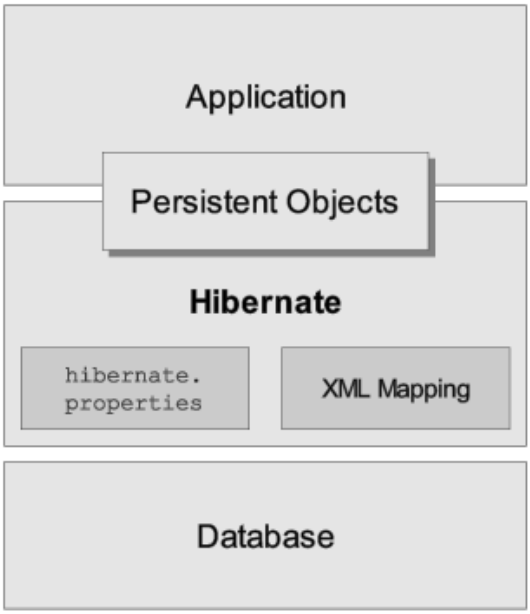
JSP og servlets

JSP (Java Server Pages) er akkurat det samme som html, men kan inneholde javakode i tillegg. Å tilby javakode i JSP betyr at det er mulig å generere dynamiske sider avhengig av hva brukeren taster inn. JSP må kompileres på lik ligne med annen javakode, og dette gjøres bl.a. første gang en side aksesseres etter endring av JSP kode. Vi vil komme tilbake til kompilering av JSP litt senere. I tillegg til JSP har vi servlets som er javakode, men som kan inneholde html. Med andre ord så kan JSP og servlets brukes til akkurat det samme som er å håndtere HTTP forespørsler slik at det vil vises brukergrensesnitt på en nettside. Begge kan utfylle hverandres oppgaver om det programmeres på den måten. Men dette er stort sett ikke ønskelig ettersom det å håndtere store deler av html-kode i javakode er uoversiktlig, på samme måte som å programmere prosessering av forespørsler med JSP er uoversiktlig og lite hensiktsmessig. Derfor egner servlets seg best til prosessesering av forespørsler, mens JSP er best på presentasjon[7].

Som det tidligere har kommet fram benytter Joly seg av JSP i presentasjonslaget, men servlets blir også tatt i bruk. Servlets håndterer og prosesserer forespørsler gjort ved spesifikke URLer (Uniform Resource Locator) som er satt opp i Joly. Disse servletene håndterer disse forespørslene og bruker JSP som bakgrunn til å generere en html-side som brukeren vil se som brukergrensesnittet.

2.3.3 Hibernate

Hibernate er det mest fleksible og funksjonsrike objekt/relasjon mapping løsning på markedet[8], hvor den tar hånd om mapping fra javaklasser til databasetabeller og fra Java datatyper til SQL datatyper. Den er lisensiert under LGPL (GNU Lesser General License), som betyr at den er gratis å ta i bruk i både utvikling og for produksjon. Med Hibernate skal utviklere slippe å bruke større deler av tiden sin til å selv programmere lagring relaterte oppgaver hvor SQL og JDBC (Java Database Connectivity) vanligvis brukes. Hibernate benytter HQL (Hibernate Query Language) ovenfor SQL, men gir brukeren fortsatt mulighet til å bruke SQL om ønskelig.



Figur 2.6: *Hibernate arkitektur på et høyt nivå*

Vi viser til Figur 2.6 som er hentet fra Hibernate sin egen dokumentasjon[13]. Her blir det presentert en høy-nivå figur av hva Hibernate er og hvor den passer inn i applikasjonen som en helhet. Hvor Hibernate opererer som en oversetter mellom applikasjonen og databasen slik at data er tilgjengelig og forståelig for begge parter. Dette drar Joly nytte av, for Hibernate gjør det mulig for objekter som er opprettet i applikasjonen under kjøring kan bli lagret i MySQL-database ved å gjøre konvertering mellom disse typene[9]. Vi vil komme nærmere innpå hvordan Hibernate fungerer i Seksjon 2.5.1 som omfatter kommunikasjon mel-

lom bruker og system. Derfor utelates det er forklaring og litt bakgrunnsinfo på hvordan Hibernate brukes i applikasjoner.

2.3.4 MySQL

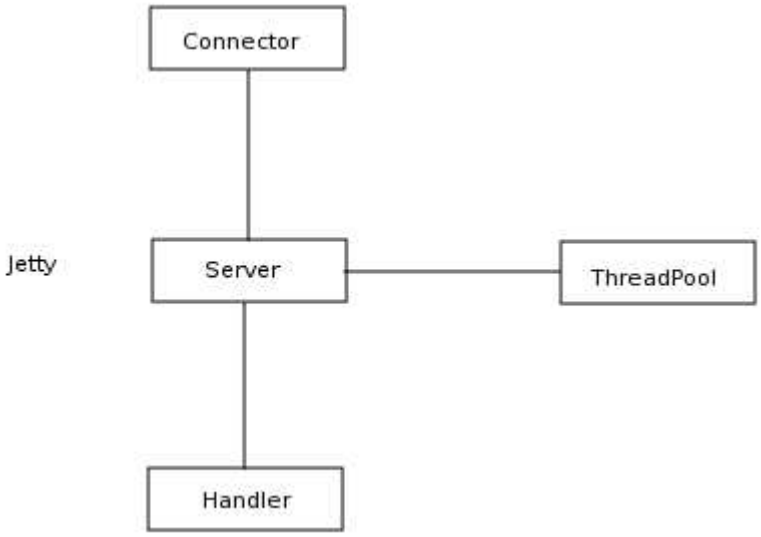
En database er en strukturert samling av data. For å kunne legge til, aksessere og behandle data i en slik database er det nødvendig med et databasehåndteringssystem slik som MySQL Server. MySQL databaser er relasjonsdatabaser som lagrer data i separate tabeller og setter relasjoner mellom disse. Slike tabeller består av rader og kolonner for oversikt over lagrede data. Det er mulig å sette forholdet mellom feltene som finnes i tabeller som kan være f.eks. en-til-en, en-til-mange, unik, valgfritt og "pekere" til andre tabeller. MySQL benytter seg av SQL (Structured Query Language) for aksessering av databaser, som for øvrig mest vanlige språket å ta i bruk for dette. MySQL programvaren er åpen kildekode programvare og er under en GPL (GNU General Public License) slik at det er mulig å gjøre endringer, men at det er begrenset hvordan det kan redistribueres og brukes. Hvis MySQL skal brukes i et kommersielt produkt er det nødvendig å kjøpe en komersiell lisens som støtter dette. MySQL database servere er raske, pålitelige og enkle å ta i bruk[22]. I tillegg gjør dens tilkoblingsmuligheter, hastighet og sikkerhet godt beregnet for aksessering av databaser over Internett.

Joly behøver å holde orden på svært mange innleveringer per semester og dette vil føre til at databasen vil bare bli større og større ettersom som tiden går. For at Joly skal kunne fungere optimalt, som er med fuskesjekk, er det nødvendig at vi får lagret innleveringer fra studenter slik at denne sjekken kan utføres og motvirke juksing. Derfor er det viktig at det benyttes et en relasjonsdatabase slik at det er mulig lagre alle typer data og relasjonene mellom dem, noe MySQL tilbyr.

2.3.5 Jetty

Jetty er et åpen kildekode prosjekt som tilbyr HTTP server, HTTP klient og javax.servlet container. Disse komponentene er tilgjengelig for kommersiell bruk og distribusjon. Den er lisensiert under både Apache License 2.0 og Eclipse Public License. Jetty Server fungerer som kobling mellom de andre komponentene som utgjør Jetty som vist på Figur 2.7. Jetty har en mengde *connectors* som er klare for å motta HTTP koblinger, for å sende de videre til *handlers* som tar imot disse forespørslene, prosesserer de og produserer respons. Denne jobben blir utført av

tråder tatt fra *threadPool* [3]. Som tidligere nevnt tilbyr Jetty en servlet container som da inneholder servlets. Servlets er et lite javaprogram som kjører på en webserver [15], og som ligger i *Handler* delen av Jetty klare for å håndtere forespørslene. Dette må ikke forveksles med *connectors* siden disse overleverer kun en forespørsel videre til først serveren, så til en *handler* som kan prosessere den.



Figur 2.7: *Jetty arkitektur*

Jetty benyttes i forbindelse med Joly som en HTTP server og javax.servlet container. Joly pakkes som WAR (Web application ARchive) og legges inn i en web applikasjonsmappe i Jetty sin filstruktur. Web applikasjonen vil så være tilgjengelig gjennom en HTTP klient, mest kjent som nettlesere. Slik Jetty benyttes i forbindelse med Joly på Universitet i Oslo er det kun mulig å aksessere Joly når man er på Universitet, fjerninnlogging eller igjennom VPN (Virtual Private Network).

2.4 Aksessering av nettsted

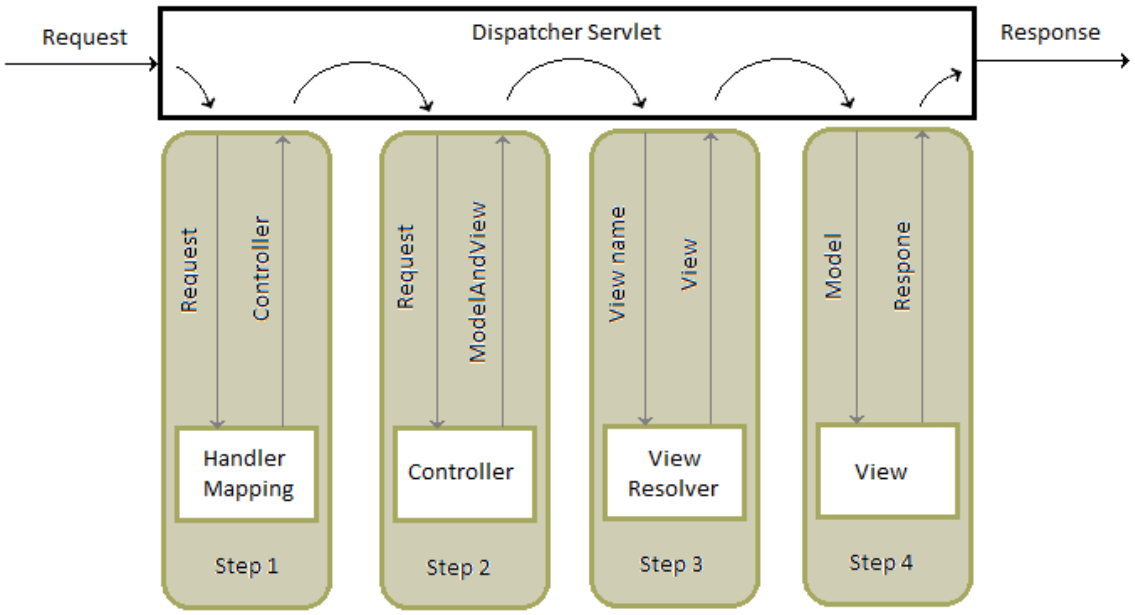
Vi vil her se hva som skjer ved aksessering av Joly som webapplikasjon. Dette vil dekke det tekniske vedrørende Jetty, hvordan Spring bidrar til å gi korrekt side tilbake i form av en html side ved å ta i bruk JSP. Vi vil dermed i neste seksjon se på en typisk interaksjon mellom Joly og bruker og hvordan resterende rammeverkene spiller inn.

Det å motta forespørsler og behandle disse står Jetty for. Den er helt nødvendig for å kunne ha en web applikasjon tilgjengelig på Internett. Det forutsettes at applikasjonen allerede kjører så vi vil ikke dekke denne prosessen ettersom det har liten relevanse. Etter oppstarten vil Joly være tilgjengelig via en URL og for Joly er dette *http://plagiat.ifi.uio.no:8080/*. Dette er siden studentene forholder seg til ved levering av obligatoriske oppgaven sin. Jetty mottar HTTP forespørsler og behandler de, og for enkelhetens skyld skal vi ikke gå inn på det tekniske some omhandler hvordan Jetty gjør alt dette. Det vi trenger å vite er at Jetty får inn en forespørsel på en URL og har oversikt over hvilke applikasjoner som den har tilgjengelig på nett. Hvis det benyttes en riktig URL så vil Jetty rute den videre til denne applikasjonen[3] som i dette tilfellet er Joly.

Når en forespørsel kommer inn til Joly vil den først bli fanget opp av en Dispatcher servlet som anses som en "front kontroller" som fungerer som et filter for alle forespørsler gjort mot Joly. Dispatcher servlet er noe som er tilgjengelig ved bruk av Spring MVC. I konfigurasjonen til Joly er det gjort klart at den vil respondere til alle URLer som ender med ".html" uansett om URLer fører fram eller ikke. I denne konteksten vil en servlet og kontroller ha samme betydning. Grunnen til disse to forskjellige termene forekommer av at Spring MVC omtaler disse servlets som kontrollere og bruker det gjennom hele dokumentasjonen som vil være hvorfor kontroller vil bli brukt mer aktivt når det beskrives sammen med Spring. Denne Dispatcher servleten er ansvarlig for å sende forespørsler videre til andre kontrollere som håndterer forespørslen så videre avhengig av hvilken URL brukeren forsøkte å aksessere. En kontroller har hovedsakelig oppgaven med å returnere en html-side til brukeren, men kan også inneholde prosessering av annen data og returnere forskjellige genererte html-sider basert på forskjellig data. Ettersom Dispatcher servleten er veldig sentral i hvordan Spring MVC returnerer en html-side skal vi se nærmere på hvordan den jobber på de forskjellige stegene. Når det kommer inn et kall på en URL vil den gå til Dispatcher servleten og følge rekkefølgen[14] som vist i Figur 2.8.

Steg 1: Sjekke handlermapping

Et kall i denne sammenhengen vil være når en spesifikk URL blir aksessert, dette genererer et kall på handlermapping som er spesifisert i konfigurasjonsfilen. Handlermapping definerer hvilke kontrollere som er assosiert med hvilke URLer



Figur 2.8: *Kallrekkefølge i dispatcher servlet*

for å så finne den riktige kontrolleren som skal aksesseres.

Steg 2: Finne riktig kontroller

Etter at den har funnet riktig kontroller i henhold til handlermappingen vil dispatcher servleten sende forespørselen videre til den spesifikke kontrolleren for behandling. Her vil kontrolleren utføre beregninger basert på data og til slutt returnere et ModelAndView objekt. Dette objektet inneholder Modeldata og View, hvor View vil peke til hvilket navn på en JSP den ønsker å returnere. For å få en bedre forståelse av hva modeldata er så er dette variabler som kan være et objekt, String osv. ved at disse er lagt til i modeldata gjør det mulig for JSPen å ta disse i bruk.

Steg 3: Sjekke Viewresolver etter navn

View som er en del av ModelAndView objektet nevnt tidligere bli kun oppgitt med navn uten prefiks eller suffiks. Så derfor er det nødvendig å konsultere View Resolver, som for øvrig også er definert og konfigurert i konfigurasjonsfilen. Denne Viewresolver vil lete etter hvor JSP-filene ligger med bakgrunn i de to parametere; prefiks og suffiks. Hvor prefiks som oftest vil peke til mappen med navn som

f.eks. */WEB-INF/jsp* som vil inneholde filene som utgjør brukergrensesnittet på en nettside. Suffiks er så og si filendelsen på filtypen som benyttes og vil derfor være .jsp. Ved å sette sammen prefiks, deler av filnavnet fra kontrolleren og suffiks så finner Dispatcher servleten fram til den riktige JSPen i filstrukturen.

Steg 4: Sende modelobjektet til view (JSP)

Nå som den har funnet riktig JSP som skal brukes vil den tilføye modeldata som ble returnert under Steg 2. Her ser vi en av styrkene ved å ta i bruk JSP siden de vil kunne dynamiske med avhengig av hva slags modeldata som blir returnert. Enkleste eksemplet på dette vil være at modeldata innholder data fra databaseaksesser, og slik data vil kunne være forskjellige fra tid til annen som gjør at den genererte siden fra JSP vil ikke alltid være det samme. Når JSPen har tatt i bruk modeldata, så vil dette produsere en html-side som brukeren så vil se.

2.4.1 Brukergrensesnitt med JSP

Ettersom Joly tar i bruk JSP i så stor grad som den gjør så er det interessant å se hvordan det blir håndtert og hva som skjer bak kulissene. Dette er også en fortsettelse for videre forståelse av servlets og JSP som vi beskrev i Seksjon 2.3.2, hvor JSP og servlets kan utføre samme jobb. Tabell 2.1 viser til hva som er tilfellet ved aksessering av JSP i forskjellige situasjoner[7]. Her antas det at JSP brukes som en servlet og derfor inneholder html-kode som normalt, men også javakode for prosessering av forespørsler. Men at selv om Joly tar i bruk servlets og JSP, som håndterer hver sin oppgave så vil kompilering av JSP fortsatt følge denne tabellen.

- **Side leses for første gang:** Når en nettside som benytter JSP aksesserer denne siden for første gang.
- **Server restartet:** Etter at webserveren er blitt restartet og webapplikasjonen starter på nytt.
- **Side modifisert:** Javakoden i JSPen er blitt endret.

	JSP oversatt til servlet	Servlet kompilert	Servlet lastet inn i servers minne	jspInit kalt	_jspService kalt
Side leses for første gang					
Forespørsel 1	Ja	Ja	Ja	Ja	Ja
Forespørsel 2	Nei	Nei	Nei	Nei	Ja
Server restartet					
Forespørsel 1	Nei	Nei	Ja	Ja	Ja
Forespørsel 2	Nei	Nei	Nei	Nei	Ja
Side modifisert					
Forespørsel 1	Ja	Ja	Ja	Ja	Ja
Forespørsel 2	Nei	Nei	Nei	Nei	Ja

Tabell 2.1: *Forespørsler av JSP i ulike situasjoner*

Ut ifra Tabell 2.1 ser vi *jspInit* og *_jspService* disse er spesifikke får i de tilfellene hvor en JSP opererer som en servlet. Hvor *jspInit* brukes for initialisering, mens andre benytter en *init* metode. Metoden *_jspService* kalles ved prosessering av forespørsler (både *GET* og *POST* forespørsler), hvor vanlige servlets benytter seg av *doGet* eller *doPost*. Det er viktig å merke seg at etter forespørsel 2 i de ulike situasjonene, så vil alle etterfølgende etterspørsler være helt like inntil det skjer en endring i systemet.

2.5 Kommunikasjon mellom rammeverk og verktøy

I denne seksjonen vil vi fokusere på kommunikasjonen mellom rammeverk og verktøy. Noe å legge merke til er at vi vil ikke komme inn på Spring Framework sin del her ettersom dette omfatter kun hvordan applikasjonen er bygget opp. For øvrig er deler av Spring som Joly tar i bruk dekket i tidligere seksjon. Joly benytter seg som beskrevet tidligere av en rekke rammeverk og verktøy, derfor er det interessant å se hvorfor de er nødvendige og hvordan interaksjonen mellom disse er. For å kunne illustrere flyten i dette vil vi se på en enkel oppretting av

et kurs. Med dette vil det være enklere å se kommunikasjonen mellom lagene i trelagsstrukturen.

2.5.1 Kommunikasjon mellom bruker og applikasjon

Ut ifra dette går vi over til hvordan det er når en bruker benytter seg av funksjonalitet på nettsiden. For dette eksemplet vil vi vise en forenklet versjon av oppretting av nytt kurs, som kun trenger kursnavn. Stort sett vil det være flere variabler enn kursnavn som f.eks. kurskode og semester, men for å illustrere prinsippet benytter vi kun en. Foreleseren finner fram til siden hvor det er mulig å opprette et kurs. Her vil det vises en tekstboks hvor man taster inn ønsket kursnavn og trykker dermed på knappen for oppretting av kurs. Listing 2.6 viser en liten del av en JSP som her inneholder en tekstboks. Denne tekstboksen vil kunne ta en String på 30 tegn, og data som blir tastet inn i dette feltet vil bli lagt inn i variabelen *course*. Siden *course* er i form-tags betyr dette at objektet Main sin course variabel blir satt til hva som er i tekstboksen. Dette *Main* objektet går under navn command-objekt og brukes til utveksling av data mellom JSP og kontroller. For at *Main* skal kunne brukes i JSPen må den ha blitt deklarert i kontrolleren.

Listing 2.6: Eksempel på bruk av command-objekt i JSP

```
1 <form:form commandName="Main">
2  <label>Tast inn kursnavn:</label>
3  <form:input cssClass="textfield" path="course" size="30" />
4  <input type="submit" name="invokeController"/>
5 </form:form>
```

Command-objektet som er deklarert i et *form* vil dermed bli sendt som parameter inn til kontrolleren. Et slik command-objekt kan se slik ut som vist i Listing 2.7 hvor den har en variabel *course* med tilsvarende setter og getter metoder.

Listing 2.7: Eksempel på command-objekt

```
1 public class MainCommand {
2  private String course;
3
4  public String getCourse(){
5   return course;
6  }
7
8  public void setCourse(Course c){
9   course = c;
```

```
10   }
11 }
```

Når en bruker velger så å trykke *send inn* for å lagre dette kurset vil dette gå videre til kontrolleren som vil prosessere denne forespørslen. Ut ifra Listing 2.8 er det mulig å se en slik kontroller, hvor metoden *submitCourse(..)* får inn dette command-objektet som ble brukt i JSPen som argument. Fra dette command-objektet får vi tak i kursnavnet som vi sender videre til logikkdelen slik at dette kan bli lagret som et nytt kurs.

Listing 2.8: Kontrollermetode for trykk av knapp

```
1 @RequestMapping(method = RequestMethod.POST, params="invokeController")
2 public String submitCourse(@ModelAttribute("Main") MainCommand command){
3  String courseName = command.getCourse();
4  \\Videre kall på logikkdelen
5  return "CourseAdded";
6 }
```

Kurset vil så bli lagt inn i databasen ved bruk av metoden som vises i Listing 2.9 som kalles fra logikken. Her kan vi se at det ikke blir tatt i bruk noe SQL eller HQL. Dette gjelder kun standardmetoder som; legg inn ny, oppdatere og slette. Hvis man f.eks. skal hente ut alle kursene som er opprettet er det nødvendig å ta i bruk HQL- eller SQL-spørringer.

Listing 2.9: Metode for å legge inn nytt kurs

```
1 public void insertCourse(Course course) throws DataAccessException {
2   getHibernateTemplate().save(course);
3 }
```

Grunnen til at Hibernate vet hvordan den skal legge inn dette objektet er på bakgrunn av filer som er mappet med annotasjoner som vist i 2.10. Selve klassen *Course* mappes mot tabellen *Course* som finnes i databasen, som blir gjort med annotasjonen *@Table*. For den ene variabelen *courseName* blir ved annotasjonen *@Id* mappet til primærnøkkel og annotasjonen *@Column* mapper den til riktig kolonne slik den er i databasen. Normalt sett vil vi ikke benytte kursnavnet til å være primærnøkkelen, men i dette eksemplet antas det at kursnavn er unikt.

Listing 2.10: Eksempel på mapping med Hibernate

```
1 @Entity(access = AccessType.PROPERTY)
```

```
2 @Table(name="Course")
3 public class Course {
4   private String courseName = "";
5
6   public Course() { }
7
8   @Id
9   @Column(name="courseName", nullable=false, length=30)
10  public String getCourseName() {
11    return courseName;
12  }
13
14  public void setCourseName(String courseName) {
15    this.courseName = courseName;
16  }
17 }
```

Når den er ferdig med å legge inn det nye kurset i databasen, vil programflyten gå tilbake til kontrolleren som vist i Listing 2.8. For dette eksemplet forutsetter vi at kurset blir lagt inn uansett som vil medføre retur av verdien *CourseAdded* som refererer til JSPen som skal returneres til brukeren. Det er f.eks. mulig å returnere en annen JSP side basert på om det var mulig å legge inn kurset eller ikke ved bruk av f.eks. en if-setning. Dermed har foreleser fått bekreftelse på at det nye kurset har blitt lagt inn.

2.6 Oppsummering

Nå har vi sett på hvilke rammeverk og verktøy som er nødvendig for at Joly skal være i drift. I tillegg har vi sett på kommunikasjonen mellom disse for å kunne få en bedre forståelse av hvor viktig oppgavene til de individuelle komponentene er. Med oversikt over Joly er det på tide å gå over til å se på hvilke nye funksjonaliteter som er planlagt for Joly.

Kapittel 3

Joly rapporter

I dette kapitlet vil vi legge fram en første spesifikasjon for ny funksjonalitet i Joly, slik at det skal kunne fungere som et godkjentlistesystem. Hvis tiden tillater det vil alle planlagte rapporter bli implementert. Den nye funksjonaliteten skal bidra til å presentere gode, informative og oversiktlige rapporter for de forskjellige brukergruppene. Noen av funksjonalitetene som er planlagt bygger på videre arbeid fra Siamek Darisiro[5] som tidligere jobbet med Joly. Her kom det fram at Joly kunne ha behov for bl.a. å koble seg på innloggingssystemet Universitetet i Oslo bruker, som vil kunne åpne muligheten for at studentene kan se sine innleverte obligatoriske oppgaver. Det nevnes også at det ønskelig å få til slik at gruppelærere kan laste ned alle obligatoriske oppgaver for valgt gruppe. I tillegg bør Joly ta i bruk HTTPS for økt sikkerhet for brukere. Dette er noen av utbedringene vi skal adressere og de vil derfor bli beskrevet nærmere. De nye funksjonalitetene som er planlagt er vist nedenfor, hvor vi starter med å se på rapportene som er planlagt.

- Rapporter
- Innloggingssystem
- HTTPS

3.1 Rapporter

I utgangspunktet var planen å finne 2 mulige rapporter for hver brukergruppe, men dette viste seg å være vanskeligere enn antatt for visse brukergrupper ettersom behovene er forskjellige. Dermed vil noen brukergrupper ha flere rapporter enn andre. Joly har følgende brukergrupper:

- Studenter
- Gruppelærere
- Forelesere

Deres rolle som bruker av Joly vil komme fram i de neste seksjonene som vi skal ta en nærmere titt på.

3.1.1 Studenter

Studentene er de hyppigste brukere ettersom de levere via Joly. Det er begrenset av hva slags rapporter studenter kunne ha bruk for, dermed vil vi bare ta sikte på implementere følgende rapport:

- Leveringsstatus på egne obliger

Ettersom studenter ikke besitter en brukerkonto som de kan logge seg inn på Joly med, så vil denne funksjonaliteten kun bli implementert om vi klarer å få Joly over på et nytt innloggingssystem.

Leveringsstatus på egne obliger

Denne rapporten skal gi oversikt over alle obligatoriske oppgaver som studenten selv har levet inn. For hver obligatoriske oppgave vil det vises følgende data; statusoversikt, innleveringstidspunkt, gruppenummer og kommentaren fra studenten. Det skal også være mulig å kunne få åpnet et nytt vindu for å se kildekoden for hver obligatoriske oppgave som studenten har levert inn. Dataene nødvendig for å få til noe som dette ligger allerede til rette i Joly, ettersom det blir opprettet og lagret i databasen ved hver innlevering fra en student. Rapporten er ment for studentene slik at de kan ha oversikt og kan fortløpende se når obligen er godkjent eller ikke. Selv om studentene har mulighet til å sjekke status på obligatoriske oppgaver via denne rapporten, så vil gruppelærerne uansett sende en mail om den er godkjent eller ikke via mail.

Som tidligere nevnt så kan vi ikke tillate å implementere en slik funksjonalitet før vi går over til et annet innloggingssystem. Med et annet innloggingssystem vil studentene bruke samme påloggingsinformasjon som de bruker for å logge seg på universitetets egne maskiner. Grunnen til dette er at det skal føles trygt for studentene å logge seg på en tjeneste de allerede en kjent med og at det er

nødvendig med innlogging ettersom det vil stå sensitiv informasjon i rapporten, i form av at kildekoden skal være tilgjengelig.

3.1.2 Gruppelærere

Den andre brukergruppen av Joly er gruppelærerne som står for retting av de innleverte obligatoriske oppgavene fra studentene. For å gjøre jobben til gruppelærerene enklere vil det være praktisk å ha rapporter som hjelper de å holder oversikten over hvem som har levert, bestått osv. som gir opphav til følgende rapporter:

- Oversikt over innleverte obligatoriske oppgaver
- Oversikt over de som er mistenkt for juks

Oversikt over innleverte obligatoriske oppgaver

Dette vil være en rapport med oversikt over alle studentene på gruppen som har levert inn sin obligatoriske oppgave. Hvor det er mulig å velge mellom de forskjellige obligatoriske oppgavene 1-4 som kurset har. Studentene på den valgte gruppen vil stå med innleveringstidspunkt, brukernavn og status. For status på obliger har vi følgende; godkjent, nytt forsøk, ikke rettet eller ikke godkjent. Samtidig skal gruppelærerne kunne endre obligstatus slik at innleveringer fra studenter er merket som f.eks. godkjent eller nytt forsøk. Dermed vil gruppelærerne se hvem som har levert og hvilke som har blitt rettet. Det vil også være mulig for gruppelærere å laste ned alle innleverte obliger på hver gruppe for fritt valgt obligatoriske oppgave.

Oversikt over de som er mistenkt for juks

Dette vil være en rapport som skal gi en oversikt over studenter hvor anti-fusk algoritmen har slått inn og er derfor mistenkt for juks. Når anti-fusk algoritmen slår inn så er det mistanke om at studenten har kopiert hele eller deler av en annen sin innleverte obligatoriske oppgave. Oversikten vil bli delt inn i de forskjellige obligatoriske oppgavene som kurset har, hvor de mistenkte vil vises med bl.a. brukernavn, innleveringstidspunkt og hvilken obligatorisk oppgave den er lik.

3.1.3 Foreleser

Den siste brukergruppen vi skal ta for oss er foreleseren som underviser kursmateriellet i INF1000 og har ansvaret for kurset. Vi så at foreleseren var en av brukergruppene som kunne ha behov og nytte av flere rapporter enn de andre. Derfor er det planlagt følgende rapporter:

- Frafall ved obligatoriske oppgaver
- Frafall på grupper
- Oversikt over retting av grupper
- Antall som har blitt tatt for juksing
- Godkjentliste

Frafall ved obligatoriske oppgaver

Denne rapporten skal vise frafallet for hver obligatoriske oppgave og totalt for hele semesteret. Her vil man kunne se prosentvis av de studentene som til slutt ikke fikk godkjent på de forskjellige obligatoriske oppgavene. I tillegg vil det vises prosentvis frafall i løpet av hele semesteret. Det vil være data som kan bli presentert i form av nedlastning av en tekstfil eller mulig visning av disse dataene ved navigering på siden. Denne informasjonen er relevant for foreleseren slik at det er mulig å se hvordan studentene gjør det mellom hver obligatoriske oppgave. Med slik informasjon vil foreleseren kunne se om visse obligatoriske oppgaver kan være vanskeligere enn andre og justere dette til neste semester om nødvendig.

Frafall på grupper

Denne rapporten vil være noe lik den forrige, men her vil vi fokusere på frafallet av studenter på gruppene gjennom semesteret. Det vil vises en oversikt for hver av gruppene hvordan frafallet er etter hver obligatoriske oppgave og på slutten av semesteret. Å vise til frafallet for individuelle grupper vil kunne gi foreleseren innsikt om at undervisningen på gruppen går som planlagt eller ikke. Med slik informasjon vil forelseren kunne ta grep og undersøke om det viser seg at frafall for en gruppe er høyere enn andre, og at det kan f.eks. ha noe med undervisningen å gjøre, men at dette ikke trenger å være eneste årsak til frafall.

Oversikt over retting av grupper

Dette vil være en rapport hvor foreleseren har mulighet til å se en oversikt over om gruppelærerne har rettet de innleverte obligatoriske oppgavene de er satt til å rette. I de fleste tilfellene har en gruppelærer ansvaret for å rette innleverte obligatoriske oppgaver av studenten på gruppen. I stedet for at det skal være nødvendig å sende mail til gruppelærerne om de har rettet obligene på gruppen eller ikke vil det vises i en oversikt. Denne oversikten vil vise til gruppenummer, brukernavnet til gruppelæreren og om alle obligene på gruppen har blitt rettet. En slik rapport vil hovedsakelig fungere som en form for overvåkning slik at foreleser enkelt kan se at gruppelæreren utfører jobben de er satt til, og hvis ikke vil foreleser f.eks. gi de en påminnelse om dette.

Antall som er blitt tatt for juksing

Denne rapporten vil vise en oversikt over studenter som har blitt tatt i juks i løpet av semesteret. Her vil det være mulig å se hvilke grupper studentene går på og på hvilken obligatorisk oppgave det gjaldt. Når det gjelder inndeling av grupper kan dette være av interesse for å se om noen grupper er mer preget av juksing enn andre. I tillegg vil det vises prosentvis hvor stor del av studentene som har jukset i det gitte semesteret. Denne rapporten kan nok virke lik ut som den som ble presentert i forbindelse med gruppelæreren, men den for gruppelæreren viser kun mistenkte studenter på sin egen gruppe. Mens denne rapporten tar sikte på de som er tatt i juksing og er blitt kastet ut av kurset.

Godkjentliste

Dette er en rapport hvor fakultetet vil få en liste over studenter som har fått godkjent på alle obligatoriske oppgaver i INF1000, slik at de vet hvem som har lov til å gå opp til eksamen. Listen vil bestå av brukernavnene til studenter som har fått godkjent på alle obligatoriske oppgavene. Her vil brukernavnet være nok til å identifisere studenter ettersom det er unikt på hele universitetet. Listen vil være tilgjengelig på selve siden, men det skal også være mulig å laste ned en generert godkjenliste på ønsket format.

3.2 Sikkerhet på web

Slik innleveringssystemet Joly fungerer i dag behøver studentene kun å vite sitt eget brukernavn for å kunne levere en obligatorisk oppgave, og evt. brukernavn og gruppenummer hvis studenten ikke er registrert i databasen. Så det er ingen krav om innlogging eller noe lignende for å verifisere identiteten sin. Dette åpner for mulige falske innleveringer fra studenter, men slik vi forstår det var dette aldri noe problem eller meldt som en feil av hverken studenter, gruppelærere eller foreleser. For Joly er det riktignok et behov for denne type innlogging for å beskytte sensitiv informasjon. Dette er fordi ingen studenter skal ha tilgang til andre studenters innleveringer. Dermed skal vi se på hvilke sikkerhetsmessige tiltak som vil gjøre bruken av Joly sikrere for alle brukere.

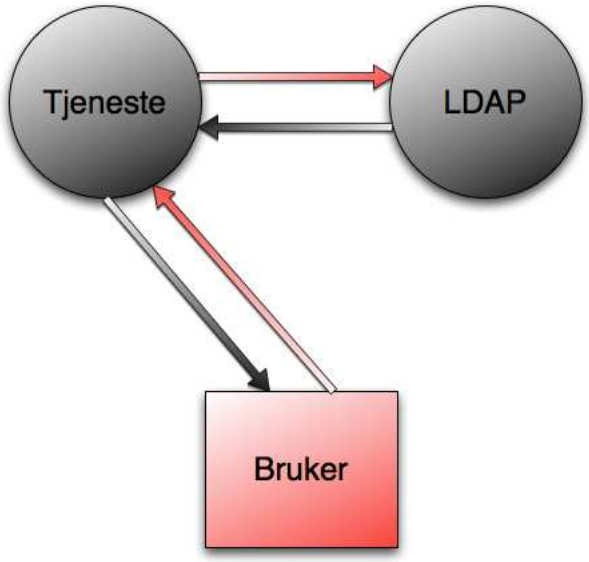
3.2.1 Innloggingssystem

Ved Universitetet i Oslo benyttes det to forskjellige innloggingstjenester for webtjenester; Feide og Weblogin. Feide brukes til å gi brukere en elektronisk ID på nasjonalt nivå på kryss og tvers av utdanninginstitusjoner. Weblogin er noe som universitet i Oslo bruker i tillegg siden Feide ikke tilfredstiller alle behov[16]. Vi velger derfor å gjøre et forsøk på å benytte Weblogin siden det er ikke behov for at Joly skal kunne aksesseres på nasjonalt nivå. Det vil bli utvekslet mail med USIT (Universitetets senter for informasjonsteknologi) som besitter informasjon om hvordan Weblogin skal integreres inn i applikasjoner. Behovet for et slikt innloggingssystem er i forbindelse med hvor studenter skal kunne ha mulighet til å logge seg på Joly for å se sensitiv informasjon som kildekoden de har levert inn. Slik det er i dag har ikke studenter mulighet til å logge inn på Joly, men kun levere uten pålogging. Dette er problemet vi vil adressere slik at det skal føles enda tryggere å levere inn sin obligatoriske oppgave.

Weblogin

Vi skal derfor se kjapt på hvordan Joly fungerer i praksis og så hvordan Weblogin opererer. Joly benytter sin egen database med data så brukernavn og passord vil derfor gå direkte igjennom applikasjonen og det gjøres sammenligning. Passord i Joly blir lagret som kun hashverdien av passordet så en viss for sikkerhet er implementert. Men applikasjonen tar fortsatt inn brukernavn og passord i klartekst for å så sjekke hashen mot den lagrede hashverdien i databasen. Joly beskriver

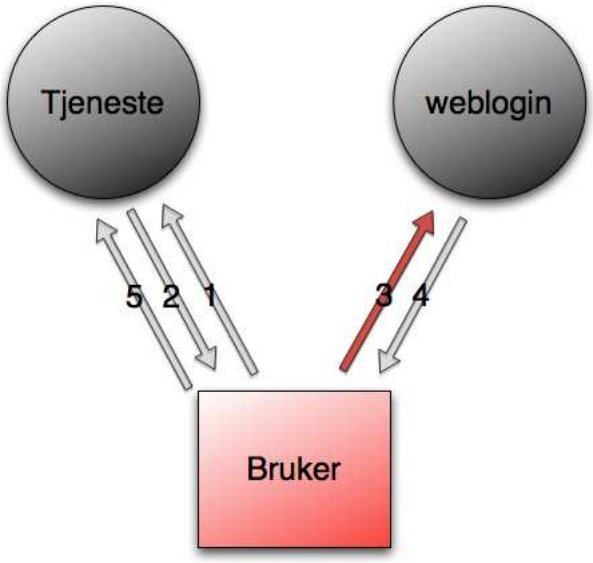
en mer tradisjonell autentiseringsmetode slik det er mulig å se av Figur 3.1 selv om den ikke er helt det samme. Forskjellen er at ifølge figuren vises det til at tjenesten tar inn brukernavn og passord i klartekst for å så sende det videre til en innloggingstjeneste som kan verifisere brukeren. Dermed får tjenesten tilgang til sensitive data, noe som gjør tjenesten mer risikabel å ta i bruk.



Figur 3.1: *Usikker innlogging*

Med Weblogin er denne innloggingsprosessen litt annet. Her kobles applikasjonen opp mot Weblogin som fungerer som autentisering hvor brukere logger seg på og ikke via applikasjonen. Av Figur 3.2 kommer dette fram ved at når det forsøkes å aksessere en tjeneste vil denne tjenesten sjekke om brukeren har logget seg på og besitter et *token* som bevis på at autentisering er utført og godkjent bruker. Besitter brukeren et *token* vil tjenesten videresende brukeren til Weblogin slik at det er mulig å logge seg inn. Ved suksessfull innlogging vil brukeren bli sendt tilbake til tjenesten med et *token* som bevis på dette og vil derfor ha tilgang til tjenesten. Det som har blitt oppnådd her er at tjenesten har nå ikke mottatt noen form for sensitiv informasjon, men har likevel fått tilgang til tjenesten via *token*.

Vi var inne på tanken om at et nytt innloggingssystem kunne være overflødig



Figur 3.2: *Innlogging med Weblogin*

sikkerhet med tanke på at Joly foreløpig er kun mulig å aksessere om man er på universitetet, VPN eller fjerninnlogging. Men i denne sammenhengen skal Weblogin være en innloggingstjeneste som studentene kjenner igjen og føler at den er trygg ettersom den er integrert på samme måte som mange andre av universitetets tjenester på web.

3.2.2 HTTPS

Joly benytter ikke HTTPS som gir totalt sett lavere sikkerhet og gjør Joly til et mulig mål for "man-in-the-middle attack" hvor en utenforstående kan lytte til kommunikasjonen mellom nettleser og applikasjon. Derfor valgte vi å se på saken om hvorvidt dette var mulig å få til for Joly, men først skal vi se på hva HTTPS er. HTTPS også kalt Hypertext Transfer Protocol Secure er en kommunikasjonsprotokoll som brukes for sikker kommunikasjon i nettverk[23]. Med HTTPS vil det foregå autentisering av nettside og web-server som man kobler seg til, slik at det er mulig å verifisere at den er det den utgir seg for å være. Dette betyr at hvor de brukes HTTPS vil det bli satt opp en sikker kommunikasjon hvis nettsiden eller web-serveren er sikre. Denne sikre kommunikasjonen vil være kryptert slik at det vil ikke være mulig for utenforstående å koble seg til for å stjele informasjon som blir utvekslet.

For å ta i bruk HTTPS betyr dette at det er nødvendig å rekonfigurere webserveren Jetty som Joly bruker. Dette vil innebære utveksling og oppretting av sertifikater som er nødvendig for å påvise det er kun riktig mottaker som får tilgang til informasjonen. Dette gjøres ved hjelp av at kommunikasjonen mellom nettleser og applikasjonen krypteres, som betyr at det vil ikke være mulig for utenforstående å få tilgang til informasjon uten kunnskap om krypteringsnøkkel. Hvorvidt dette blir mulig å ordne avhenger av tiden og hvor komplisert det vil være å få til.

3.3 Om forbedringer

Til tross for de planlagte forbedringer så vil det absolutte minimumet for å betrakte Joly som et godkjentlistesystem vil være hvis vi får fullført *oversikten over innleverte obligatoriske oppgaver* og *godkjentliste*. Vi har valgt å utelate noen forbedringer som ikke har noe for seg på det stadiet vi er nå. Dette gjelder å kunne tilby muligheten for en printet versjon eller få det i pdf format. I senere tid kan det være aktuelt å implementere rapporter som sammenligner semestre mot hverandre. Sistnevnte vil ikke bli implementert i denne versjonen av applikasjonen ettersom det vil være mangel på data til å gjøre en slik rapport informativt på dette stadiet.

3.4 Oppsummering

Vi har sett på hvilke rapporter og annen funksjonalitet vi ønsker å implementere i Joly. Om det er mulig å implementere alt som er spesifisert avhenger om tiden tillater det. Men før vi kan begynne med implementering behøver vi å oppdatere Joly, noe vi skal se nærmere på i neste kapittel.

Kapittel 4

Oppdatering av rammeverk og verktøy

I dette kapitlet skal vi se på oppgaven med å bytte ut rammeverk og verktøy som benyttes i Joly. Det ble tidligere gjort et forsøk på dette, som var delvis utført med Siamek Darisiro[5], men kom til kort med å bytte rammeverkene. Dette gjør at oppgaven faller på oss og vil innebære å finne ut hvilke rammeverk som benyttes i Joly og om disse har behov for å bli oppdatert. Rammeverkene vil bli byttet ut med potensielt nyere versjoner og senere testet. Det vi vil oppnå ved å oppdatere disse rammeverkene er for å gjøre Joly sikrere og for å gi oss en versjon av Joly som er klar for videreutvikling. I tillegg fant vi det nødvendig å ha en oversikt over det som vil være nåværende versjonene på rammeverkene, slik at det skal være lettere å vite om det behøves oppdatering av rammeverk. Avsluttende vil vi se på hvorvidt disse rammeverkene er verdt å bruke med hensyn på vedlikeholdet som medfølger.

4.1 Hvorfor oppdatere?

Rammeverk er som tidligere nevnt verktøy for å gjøre utvikling av applikasjoner raskere og for å ha full fokus på logikken og ikke alt det tekniske rundt det. Rammeverk består av ett eller flere biblioteker; Et bibliotek består igjen av en eller flere pakker, som så består av en eller flere klasser. Siden Joly teknisk sett består av både rammeverk og bibliotek gjør det at disse to vil bli brukt om hverandre. Derfor vil rammeverk bli brukt som en betegnelse av totalen av alle rammeverk og bibliotek brukt i Joly. Rammeverk og verktøy som har åpen kildekode har man

ingen garanti for at de er ferdigutviklet og feilfrie. Oppdateringer til rammeverk og verktøy skjer regelmessig og er ment for å adressere følgende problemer:

- **Feilretting og/eller forbedring:** Retting av feil i koden som skaper problemer eller forbedre hvordan disse fungerer.
- **Sikkerhetshull:** Retting av potensielle sikkerhetshull som er viktig slik at applikasjonen har mindre sjanse for å bli utsatt for angrep.
- **Nye funksjoner:** Implementasjon av nye funksjoner som skal gjøre utvikling av applikasjoner enklere.
- **Opprydding:** Omstruktureringer av hvor pakker og klasser ligger. Som bl.a. at de blir lagt til og/eller fjernet. I tillegg til evt. nye navn på pakker, klasser eller bibliotek.

Hovedgrunnen til å oppdatere disse rammeverkene er at Joly skal videreutvikles, og i denne forbindelsen er det uaktuelt at dette gjøres med tilnærmet over 5 år gamle rammeverk. Samtidig er det nødvendig å få skaffet seg oversikt over om Joly faktisk benytter alle de vedlagte bibliotekene, for å så fjerne de som er overflødige. Det er aktuelt å oppdatere fordi nyere versjoner av rammeverk retter opp problem som f.eks. sikkerhetshull. Hvis derimot en nyere versjon av et rammeverk introduserer ny funksjonalitet som Joly ikke benytter seg av, vil oppdatere ha liten verdi for oss. Ettersom det er mange rammeverk og et veldig stort tidsrom mellom versjoner ser vi det ikke hensiktsmessig å finne ut hvor mye nyere rammeverk hjelper Joly. Men at det er interessant å nevne at selv om rammeverkene blir oppdatert er det ingen garanti at alle innehar feilrettinger som styrker Joly sin sikkerhet. Selv om det er viktig å holde rammeverkene applikasjonen bruker vedlike, så kan oppdatering av rammeverk medføre problemer for applikasjonen, noe som vi skal se senere.Med tanke på videreutvikling så hadde vi to muligheter når det gjaldt rammeverkene:

1. Videreutvikle med **gamle** versjoner av rammeverkene
2. Videreutvikle med **nye**, oppdaterte rammeverk

Begge disse to mulighetene har fordeler og ulemper. Videreutvikler vi med eldre rammeverk så vil ny funksjonalitet bli implementert raskere, men ulempen er at når rammeverkene først blir oppdatert vil den nye funksjonaliteten kunne medføre mer arbeid med å rette opp dette i tillegg til eldre kode til ny syntaks. På den andre siden har vi å ha oppdatert rammeverkene først, som vil kunne gi

Joly økt sikkerhet, for å så videreutvikle. Dette vil kunne medføre muligheten for forsinkelse av ny funksjonalitet. Vi har her valgt å oppdatere rammeverk først for å begrense potensielle mengden følgefeil som vil kunne være et problem senere, hvis en slik oppdatering hadde blitt gjort i etterkant.

4.2 Finne rammeverk og verktøy

Når det har gått relativt lang tid mellom oppdatering av rammeverk og verktøy for en applikasjon, slik det har med Joly. Så kan det medføre at det blir potensielt vanskeligere å finne de nøyaktige bibliotekene man leter etter, noe som var tilfellet for oss. Biblioteker kan ha fått nytt navn, pakker har blitt fjernet eller endret. Planen er å oppdrive nyere versjoner av rammeverkene og verktøy Joly benytter seg av. Det er viktig å merke seg at fire av disse rammeverkene medfølger distribusjonen av verktøyet Jetty. I de neste seksjonene skal vi se på fremgangsmåten for å finne disse rammeverkene og verktøy. Samtidig ser vi på hvilke problemer som oppstod ved å bruke en slik fremgangsmåte. Deretter skal vi danne oss en oppdateringsplan som skal gi oss oversikt og grunnlaget for oppdatering av rammeverkene. Vi vil benytte oss av Eclipse, som er et integrert utviklingsmiljø (IDE) som gjør det enklere å holde oversikt over større prosjekter. Derfor vil det meste av feilsøkingen gå igjennom dette.

4.2.1 Fremgangsmåte for å finne bibliotek

For å kunne oppdatere Joly er det nødvendig å finne hvilke rammeverk som det er behov for. Utgangspunktet vi hadde var en liste over de fleste rammeverk samt en liten beskrivelse av dem, og rammeverkfilene vedlagt sammen med Joly. Det totale antallet biblioteker vi har er 34, og dette er inkludert bibliotekene fra Jetty.

Identifisere bibliotek og finne versjonsnummer

Den versjonen vi hadde tilgjengelig var fra tidligere arbeid utført av Siamek Darisiro[5], som går under versjon 1.6. For å identifisere bibliotekene i Joly 1.6 tok vi utgangspunkt i mappen som kommer vedlagt med Joly hvor alle bibliotekene ligger. Stort sett hadde bibliotekene et selvforklarende filnavn, men det var andre som ikke fulgte samme prinsipp. Dette gjorde det vanskeligere å finne en ny versjon til disse ettersom vi ikke visste hvilke biblioteket det var. De bibliotekene det var mulig å finne navnet på ble identifisert, så var det aktuelt å finne

ut hvilke versjoner Joly benytter seg av. Det var en større andel av bibliotekene som hadde versjonsnummeret inkludert i filnavnet enn de som ikke hadde det. Vi ønsket å finne ut versjonsnummerne for å se om det var den nyeste versjonen og om det var behov for å bytte den. For å kunne finne versjonsnummeret var det enkleste hvis dette allerede var inkludert i filnavnet. Andre muligheten var å se igjennom manifestfilen som var inkludert i biblioteket. Manifestfilen kunne inneholde informasjon som f.eks. hvilket kompileringsverktøy som ble brukt og versjonsnummeret, noe vi kan se av Listing 4.1, som viser til at dette er versjon 3.1 av Hibernate.

Listing 4.1: Utdrag fra manifestfilen fra Hibernate

```
1 Manifest-Version: 1.0
2 Ant-Version: Apache Ant 1.6.5
3 Created-By: 1.4.2_09-b05 (Sun Microsystems Inc.)
4 Hibernate-Version: 3.1.
```

Problemet vi hadde var at i de tilfellene hvor det var usikkerhet rundt versjonsnummeret var manifestfilen eneste måte å kunne få tak i informasjon om biblioteket. Selv om det var en nødløsning for å finne informasjon så var det ikke alle bibliotekene som hadde tilstrekkelig informasjon vedlagt med dem, hvis noe i det hele tatt. Dette gjaldt stort sett de bibliotekene som hadde lite selvforklarende filnavn og ikke noe versjonsnummer inkludert. Så det ble betraktelig vanskeligere å finne ut hvilket bibliotek det var, slik at vi kunne finne erstatninger. I de fleste tilfellene sto versjonsnummeret i manifestfilen, men at det skilles mellom spesifikasjons- og implementasjonsversjon, og hvor dette oppstår er det blitt oppført implementasjonsversjonen.

Finne nye bibliotek

Siden Joly benytter seg av Spring Framework var det naturlig å hente de nyeste tilgjengelige filene. På Spring sin egen hjemmeside distribuerer de Spring Framework og en sammensetting av Spring, inkludert de fleste åpen kildekode biblioteker som normalt brukes sammen med Spring. Dette ble derfor hovedkilden til hvor vi ville hente bibliotekene vi trengte. Det viste seg i etterkant at sammensetningen var opptil ett år gammelt, noe som gjorde at den ikke inkluderte nyeste versjon av alle bibliotekene. Ett av disse var bl.a. Hibernate som i denne distribusjonen var en tidlig versjon av Hibernate 3. Vi valgte derfor å hente den nyeste versjonen av Hibernate 3 som er tilgjengelig, med tanke på at Hibernate 3 var

den siste som ble distribuert i en sammensetting med Spring, og at den derfor kunne være godt integrert og velfungerende. Den nyeste versjonen av Hibernate 3 er 3.6.10 som ble sluppet 9. februar 2012. Mens Hibernate 4, versjon 4.0.0 ble sluppet 15. desember 2011, og er nå i 4.1.4 som ble sluppet 31. mai 2012. Det kan tyde på at det er relevante forskjeller mellom disse to versjonene av Hibernate, uten at vi skal gå nærmere inn på det her. Dette gjorde det aktuelt å beholde Hibernate 3 istedenfor Hibernate 4 ettersom Hibernate 3 allerede er godt etablert. Det andre er at ved å holde oss til Hibernate 3 lar vi være å introdusere noe nytt med tanke på at Joly allerede fungerer med Hibernate 3, og siden dette kan medføre mer arbeid og potensielle feil.

I de tilfellene hvor filnavnet på bibliotekene var lite forklarende var det nødvendig å se på pakkestrukturen og hvilket pakkenavn de lå under for å så søke etter de på denne måten. Stort sett var alle bibliotekene tilgjengelig fra sammensettingen av biblioteker hentet fra Spring Framework sin hjemmeside. Så det ble en del arbeid med sammenligning av de bibliotekene det var usikkerhet rundt. De fleste ble funnet på denne måten, og de som ikke var mulig å finne ble satt opp for mulig fjerning. Alle rammeverkene som Joly bruker og de som ble funnet ny versjon av vil bli presentert i Seksjon 4.2.3. Dette vil danne grunnlaget for en oppdateringsplan, som skal være ment som oversikt og veiledning rundt oppdateringen av rammeverk i Joly.

Det var ett fåtall av bibliotekene som etter denne prosessen sto uten en nyere versjon, p.g.a. at det ikke var mulig å finne ut hvilket bibliotek det var. Derfor valgte vi å planlegge og fjerne dem i Joly 1.7 for å finne ut om de var nødvendige. Dette ble gjort slik at det ikke ble brukt unødig lang tid på å finne ut hvilke biblioteker det var, for å så finne oppdaterte versjoner av dem. Vi vil komme tilbake til dette i Seksjon 4.4.1 hvor det vil komme fram om våre antakelser om disse bibliotekene var korrekte.

Jetty er som tidligere nevnt et verktøy Joly bruker til å sette opp applikasjonen på internett. Jetty står for fire av bibliotekene Joly trenger, som gjorde det aktuelt å oppdatere disse. De forsøkene som ble gjort med Jetty var å hente den nyeste versjon tilgjengelig som var 8.0.0. Her viste det seg at det hadde vært betydelige forandringer ettersom ingen av de eldre bibliotekene var mulig å finne i den nyeste distribusjonen av Jetty. Vi skal komme tilbake til dette i Seksjon 4.5 som

omhandler Jetty.

Testing av bibliotek

Verktøy brukt i denne sammenhengen var Eclipse, dette ga en bedre oversikt over prosjektet, filstrukturen, pakkeoversikt og direkte tilbakemelding om feil i koden (ettersom Eclipse kompilerer filer i bakgrunn ved hver lagring). Dette er kun en liten del av det Eclipse har å tilby, men dette var hovedsakelig motivasjonen for å bruke det under testing som dette. Testingen bestod av å fjerne en eldre versjon av ett bibliotek for å så legge til den nye versjonen, for å se om det oppstod noen feil i kildekoden, slik at vi kunne anslå at det var potensielt riktig bibliotek. Under denne testingen ble ikke Joly kjørt siden det var kun ment for å se om det oppstod feilmeldinger når vi fjernet eldre bibliotek og erstattet dem med nyere bibliotek.

Ettersom flere og flere bibliotek ble funnet var disse lagt inn i en egen mappe under web-delen av Joly. Dette var for å få bedre oversikt og mest for ikke å forveksle disse nyere bibliotek med eldre. For å være sikker på at biblioteket som ble funnet var riktige ble ett eldre bibliotek fjernet fra byggestien (build path), som ville i det fleste tilfeller få Eclipse til å vise feil i prosjektet og hvilke kildekodefiler dette gjaldt. I byggestien ligger bibliotek som man vil at Eclipse skal inkludere når den kompilerer. Feilene som da oppstod var hovedsakelig at Eclipse ikke kunne finne en pakke som var importert. På dette punktet ble så det nyere biblioteket lagt til i byggestien som forteller Eclipse at den nå har tilgang til nye klasser og metoder. Hvis dette var det riktige biblioteket ville alle eller de fleste feilene bli borte. I visse tilfeller hvor import-setningene i starten av filen var i orden, oppstod det allikevel feilmeldinger i kildekoden. Det største problemet som oppstod var å finne en erstatning for ejb3-persistence. Denne ga feil i opptil flere filer i Joly sin kode som bl.a. feil angående annotasjoner og import-setninger, noe som ga oss derfor inntrykk om at det kunne være feil bibliotek.

Det viste seg krevende å finne bibliotek når utgangspunktet var alt annet enn optimalt. Sånn totalt sett fant vi mange rammeverk som vi antar kan være de riktige og som vi senere skal se om de faktisk var det. Vi valgte å gjøre det slik og vil diskutere erfaringer rundt denne fremgangsmåten mot slutten.

Problemer

Det oppstod litt problemer under testing av bibliotek som forrige seksjon beskrev. Under utførelsen av test-bytting av bibliotekene valgte vi å opprette en ny mappe som skulle inneholde de nye versjonene av bibliotekene. Dette var for å skille de eldre og nye biblioteken slik at de ikke ble blandet sammen og skapte problemer eller forvirring. Som tidligere nevnt benyttet vi Eclipse til dette, hvor vi en etter en fjernet et eldre bibliotek og la til et nyere bibliotek i byggestien. Ved kjøring av ant-skriptet som bygger WAR-filen av Joly oppstod det et problem siden de nyere bibliotekene lå i en annen mappe enn de eldre. Dette ant-skriptet spesifiserte at den skulle alltid benytte den mappen som de eldre bibliotekene lå i, noe vi ikke visste. På grunn av dette ble det litt ekstra arbeid fordi det var nødvendig å gå tilbake til starten før noen endringer ble gjort for å begynne på nytt igjen.

4.2.2 Endringer i rammeverkstruktur

I leting etter nyere versjoner av eksisterende rammeverk i Joly 1.6 oppdaget vi at det hadde skjedd endringer i selve rammeverket. Dette gjorde det vanskeligere å finne ut hvilke biblioteker vi hadde, samt finne nyere versjoner til disse. Disse endringene oppdaget vi under leting og evt. sammenligning med tidligere og nyere biblioteker for å fastslå at vi forhåpentligvis hadde lokalisert de riktige bibliotekene. De fleste sammenslåinger og omstrukteringer av rammeverk vil komme tydelig fram av Tabell 4.1.

Spring Framework har blitt delt opp i 12 andre biblioteker. I Joly 1.6 bestod Spring kun av ett bibliotek som inneholdt all funksjonaliteten, men dette er blitt endret til å bestå av langt flere og slik vil det bli i Joly 1.7. Dette gjorde det utfordrende å finne ut hvilke spesifikke bibliotek Joly hadde behov for. Det ble dermed gjort ved at de nyere Spring bibliotekene ble lagt til en etter en for å se om det rettet opp feilmeldingene som oppstod når den eldre versjonen ble fjernet. Det var ingen store problemer rundt dette, annet enn at det ene biblioteket hadde byttet navn fra *dao* til *transaction*. Det ble oppdaget underveis på bakgrunn av mangel på klassen *DataAccessException* som ble funnet i transaction-biblioteket av Spring. Ved å ta i bruk nyeste versjon av Spring viste det seg at SimpleFormController klassen hadde blitt utdatert (deprecated). Dette er en klasse som er hyppigst brukt i Joly når det gjelder kontrollere. Vi vil komme tilbake til hva vi valgte å gjøre med dette i Seksjon 4.6.

En erstatter for biblioteket *ejb3-persistence* viste seg å gi mer problemer enn de andre. For å finne ut hvilket bibliotek dette var måtte vi ned på pakkenivå å se på hvilke pakker og klasser den bestod av. På pakkenivå gikk biblioteket *ejb3-persistence* under navnet *javax.persistence*. Ved forsøk på bytting viste det seg at en pakke ikke eksisterte lenger og at forskjellige annotasjoner ikke lenger var gyldige. Det ble gjort noen videre forsøk med noen andre bibliotek, men minst feil ble produsert med biblioteket *javax.persistence*. Derfor antok vi at dette kunne være riktige biblioteket, men at modifikasjoner til hvordan annotasjoner skal brukes kan ha bli endret.

De mindre endringene sto Hibernate og aspectj.tools for. Bibliotekene *aspectjrt* og *aspectweaver* i Joly 1.6 er sammenslått til ett bibliotek og går under biblioteknavnet *aspectj.tools*. Dette er i likhet med *hibernate-core* og *hibernate-annotations* som er samlet under biblioteket *hibernate3*. Her ser vi antydninger til problematikken rundt det å bruke rammeverk i applikasjoner med lite eller ingen vedlikehold. Det viser seg at med endringer i strukturen kan det fort bli vanskelig å vite hva som er hva etter mange år.

4.2.3 Oppdateringsplan

Siden det er fordelaktig å ha en plan for hva som skal gjøres, ble dette utarbeidet og vises i Tabell 4.1. Dette er for at det skal være enkelt å se hva som skal byttes ut og hvilken rekkefølge det skal foregå. Den er ment som en oversikt over biblioteker og hvilke versjoner som ble brukt i Joly 1.6, og som skal brukes i Joly 1.7. Tallene 1 til 9 indikerer trinn for utbytting av bibliotek, der trinn 1 viser til de første bibliotekene som skal byttes ut, deretter trinn 2 osv., helt til alle er blitt byttet ut. Tabellen er også ment for å vise hvor utdarte noen av disse rammeverkene egentlig er, og derfor er versjonsnummer inkludert.

Det er noen mangler i tabellen ettersom det ikke var mulig å finne versjonsnummeret til alle eldre biblioteker. Så der hvor versjonsnummeret ikke er kjent er det plassert et spørsmålstegn. Ved å se på tabellen legger man merke til at det står *fjerne* og *uendret* i Joly 1.7 sin kolonne. Disse indikerer at bibliotek merket *fjerne* vil bli forsøkt fjernet i Joly 1.7 på bakgrunn av tidligere forsøk på å fjerne dem og at dette ikke ga synlige feil i koden. Når det gjelder *uendret* er dette bi-

	Joly 1.6	Versjon	Joly 1.7	Versjon
1	aspectjrt	1.5.3	org.aspectj.tools	1.6.6
	aspectweaver	1.5.3		
	ejb3-persistence	?	javax.persistence	2.0.0
	antlr	2.7.6	antlr	2.7.7
2	hibernate3	3.1	org.hibernate	3.6.10
	hibernate-annotations	3.1beta7		
	ehchache	1.1	net.sf.ehcache	1.6.2
	jstl	1.1.2	javax.servlet.jsp.jstl	1.1.2
3	jta	?	javax.transaction	1.1.0
	junit	?	junit	4.10
	dom4j	1.6.1	org.dom4j	1.6.1
	commons-collections	2.1.1	org.apache.commons.collections	3.2.1
	commons-logging	1.0.4	org.apache.commons.logging	1.1.1
4	mysql-connector	3.1.11	mysql-connector-java	5.1.19
	commons-io	1.3.2	org.apache.commons.io	1.4.0
	commons-fileupload	1.2	org.apache.commons.fileupload	1.2.0
	postgresql	8.0-319	Fjerne	
	log4j	1.2.11	org.apache.log4j	1.2.15
5	standard	1.1.2	org.apache.taglibs.standard	1.1.2
	asm	2.2.3	Fjerne	
	asm-commons	2.2.3	Fjerne	
	jdbc-stdext	2.0	Fjerne	
6	spring	2.0.6	org.springframework.beans	3.1.0
			org.springframework.context	3.1.0
			org.springframework.context.support	3.1.0
			org.springframework.core	3.1.0
			org.springframework.jdbc	3.1.0
			org.springframework.orm	3.1.0
			org.springframework.transaction	3.1.0
			org.springframework.web	3.1.0
			org.springframework.web.servlet	3.1.0
7	mail	1.3.2	javax.mail	1.4.0
	activation	1.0.2	javax.activation	1.1.0
	c3p0	0.9.0	com.mchange.v2.c3p0	0.9.1.2
	jakarta-oro	2.0.8	org.apache.oro	2.0.8
8	xml-apis	?	org.apache.xmlcommons	1.3.3
	acegi-security	1.0.9	Fjerne	
	cglib-nodep	2.1_3	net.sf.cglib	2.1.3
9	javax.servlet	2.4	Uendret	
	org.mortbay.jetty	5.1.12	Uendret	
	org.mortbay.jmx	?	Uendret	
	xmlParserAPIs	?	Uendret	

Tabell 4.1: *Plan for bytting av rammeverk, inkl. versjonsnummer for Joly 1.6 og 1.7*

bliotek som forblir uendret i Joly 1.7. Bibliotekene dette gjelder er forholdsvis de som distribueres med Jetty. Ettersom Jetty ikke skal byttes ut i første omgang vil disse forbli urørt inntil videre. Hvis et bibliotek allerede er av nyeste versjon vil det ikke bli byttet ut. Dette betyr at hvis et bibliotek er av ett gitt versjonsnummer og dette ikke har endret seg så vil versjonsnummer bli notert, men ingen bytting av bibliotek vil bli utført i oppdateringsprosessen.

Det var ønskelig å kunne oppdatere alle bibliotekene til nyere versjoner, men mangel på kunnskap og dokumentasjon rundt de eldre bibliotekene gjorde det vanskelig å finne alle bibliotekene. Vi vil derfor senere i kapitlet se om vi fant nyere versjoner på de mest nødvendige bibliotekene, og om det er mulig å klare seg uten alle bibliotekene. Det kan være mulig at vi må ta i bruk eldre bibliotek i kombinasjon med nyere bibliotek, og det er usikkerhet rundt om de nye vil være kompatible med eldre.

Måten å finne tidligere rammeverk basert på navn og litt bakgrunnsinformasjon viste seg å være en helt fungerende metode. Som evaluering av kun metoden i seg selv ville det vært veldig lite som kunne blitt gjort annerledes. Med veldig liten erfaring innenfor rammeverk viste det seg veldig vanskelig å finne konkrete oppdateringsnotater på endringer gjort i rammeverk, hvertfall etter så lang tid. I neste seksjon vil vi se videre på om vi har funnet riktige rammeverk eller ikke, og på selve oppdateringen.

4.3 Oppdatering av kildekode

Vi har nå funnet alle rammeverkene vi hadde mulighet til og skal nå se videre på endringer, feilmeldinger og problemer som oppstod ved bytting. Planen for bytting som ble lagt fram i Seksjon 4.2 og som var utgangspunktet for hvordan byttingen skulle foregå. Det ville starte fra trinn 1 til og med trinn 8 som vist Tabell 4.1. Ved bytting ble nyere utvalgte bibliotek kopiert over til mappen som inneholder bibliotekene Joly bruker. Den eldre versjonen ble flyttet over til en annen mappe slik at det var mulig å gå tilbake om det skulle vise seg å oppstå problemer. Etter at hvert trinn hadde blitt utført ville Joly bli kompilert ved hjelp av et ant-script og videre kjørt med Jetty. Vi så det svært sannsynlig at ved bytting av så mange rammeverk ville det oppstå feilmeldinger og diverse andre feil som Seksjon 4.2.2 indikerte. Joly ville så bli testet for enkel funksjonalitet i den grad dette var mulig

(hvis Joly ikke kunne kjøres var det ikke mulig å teste). De planlagte testene var som følger:

1. **Innlevering av en oblig** uten at den produserer noen feil, denne tester input fra bruker, til kontroller og aksess av database.
2. **Innlogging som administrator** og benytte seg av diverse funksjonaliteter som er tilgjengelig. Som oversikt over grupper, oversikt over obliger osv.

Dette vil være den enkleste og tidsparende måten å teste Joly på ettersom det ikke er skrevet tester til Joly. Så det vil være på bakgrunn av dette vi vil anslå om Joly fungerer som det skal. Det er viktig å presisere at det var ikke mulig å få testet før alle feilene og endringene ble rettet, som beskrives i de neste seksjonene. Men hver av seksjonene vil fortsatt inneholde informasjon om hva som ble testet for å fastslå at Joly 1.7 fungerer som den skal. Testing ble gjort for å være sikker på at endringene ikke hadde negativ effekt på Joly, i den formen av at funksjonalitet ikke fungerer som det skal.

4.3.1 Første endring (Entity, Id, JoinTable og Basic annotasjon)

Bytte av bibliotek ble gjort i henhold til Tabell 4.1. Endringene i trinn 1 medførte feil i ni filer tilhørende pakken uio.ifi.joly.domain, som har med mapping til databasen å gjøre. Alle feilene som oppstod hadde med annotasjoner å gjøre. Antall feil som oppstod i disse filene rangerte mellom to og fire, og var som følgende:

- javax.persistence.GeneratorType ikke funnet
- Attributten access er udefinert for annotasjonen av type Entity
- Attributten generate er udefinert for annotasjonen av type Id
- Attributten table er udefinert for annotasjonen av type JoinTable
- Attributten temporalType er udefinert for annotasjonen av type Basic

Dette tyder på at det er skjedd en del endringer i henhold til hvordan annotasjoner brukes i nyere versjoner av rammeverk. For å finne en løsning på dette ble det nødvendig å konsultere seg med Hibernate sin dokumentasjon over bruk av Annotasjoner[1]. Dokumentasjonen viste til bruken av diverse annotasjoner og det var derfor behov for å prøve og se likheten mellom tidligere annotasjoner og hvordan de brukes nå. Mulige løsninger på problemene kommer fram av Tabell 4.2. Dette gir kun løsninger på fire av fem problemer som er nevnt i tabellen.

Joly 1.6	**Joly 1.7**
@Id(generate = GeneratorType.AUTO)	@Id @GeneratedValue
@Basic(temporalType = TemporalType.TIMESTAMP)	@Temporal(TemporalType.TIMESTAMP)
@Entity(access = AccessType.PROPERTY)	@Entity @Access(AccessType.PROPERTY)
table=@Table(name="Group_Student")	name="Group_Student"

Tabell 4.2: *Endring av annotasjoner fra Joly 1.6 til 1.7*

Men ettersom GeneratorType er erstattet med GenerationType[2] og den ikke er nødvendig for de andre endringene er den derfor ikke inkludert i Tabell 4.2.

Med det går vi over til de nødvendige endringene. Problemet for Entity annotasjonen var følgende; Klassen Entity har ikke lenger en variabel access som er av typen AccessType, og aksepterer derfor ikke dette som parameter. Den benytter seg nå av en ny annotasjon Access som spesifiserer AccessType på den måten istedenfor gjennom Entity annotasjonen. Dette kommer fram av Listing 4.3. Dette viser til kun en av fire endringer som ble utført. Resten av disse finnes i Appendiks A, Listing A.3 - A.8.

Listing 4.2: Joly 1.6 Entity annotasjon

```
1 @Entity(access = AccessType.PROPERTY)
2 @Table(name="Course", uniqueConstraints
3     = {@UniqueConstraint(columnNames={"semester", "courseCode"})})
4 public class Course {
5   ...
6 }
```

Listing 4.3: Joly 1.7 Entity annotasjon

```
1 @Entity
2 @Access(AccessType.PROPERTY)
3 @Table(name="Course", uniqueConstraints
4     = {@UniqueConstraint(columnNames={"semester", "courseCode"})})
5 public class Course {
6   ...
7 }
```

Etter at endringene var på plass var det mulig å kompilere Joly med antscriptet som gjør denne jobben. Dette åpnet muligheten for videre oppdatering og

testing, og ikke minst se om de endringene som er blitt gjort er korrekte. Det viste seg at ikke alt ble utført helt riktig, slik neste seksjon vil beskrive.

4.3.2 Andre endring (TemporalType annotasjon)

Når det var igjen mulighet for å kunne kjøre Joly oppstod det diverse feilmeldinger av mangler på bibliotek og dermed en feilmelding angående annotasjonen Temporal som vil derfor bli det neste vi ser på. Disse feilmeldingene vil bli kommentert i Seksjon 4.4.3. Feilmeldingen uttrykte at annotasjonen Temporal kan kun settes over metoder med returtype av typen java.util.Date eller java.util.Calendar. Som gjorde at det var behov for å endre returtypen fra Timestamp til java.util.Date. Dette var en liten endring fra slik det er satt opp i Listing A.8 til slik det er vist i Listing 4.4.

Listing 4.4: Joly 1.7 Temporal annotasjon

```
1 @Temporal(TemporalType.TIMESTAMP)
2 @Column(name="deliverytime", nullable=false)
3 public Date getDeliverytime() {
4   return deliverytime;
5 }
```

Denne metoden benyttes i forbindelse med å hente ut innleveringstidspunktet for en oblig fra databasen. Endringen som er blitt gjort kunne kanskje medføre at det ble feil når dato skal hentes ut fra databasen. I denne forbindelse gjorde vi et par tester hvor det ble gjort innleveringer og sjekket etterpå at disse ble hentet ut riktig. Dette ble gjort ved å legge inn en linje som ville skrive ut hva variabelen *deliverytime* var i både setter og getter metoden for *deliverytime*. Resultatet av testene var at datoene ble satt og hentet ut helt korrekt.

4.3.3 Tredje endring (LazyCollection)

Dette problemet vi støtte på her var det som ga mest problemer ettersom feilmeldingen var noe som vi ikke hadde vært borti før og som ikke var enkel å forstå med det første. Det som ble oppgitt som feilmelding var følgende: *"HibernateException: cannot simultaneously, fetch multiple bags"*. Vi var usikre på hva problemet kunne være så det ble gjort antakelser i forhold til hva som kunne være en mulig løsning. Feilmeldingen vi fikk i denne sammenhengen kunne assosieres med Hibernate,

og vi ville derfor forsøke å se om nyeste versjonen av Hibernate ville ordne problemet. Dermed ble det gjort forsøk med å bytte ut Hibernate 3.6.10 med Hibernate 4.1 for å se om dette ville føre fram. Dette resulterte i nye feilmeldinger som skyldtes migrering fra Hibernate 3 til 4. På dette tidspunktet valgte vi å gå tilbake til Hibernate 3.6.10 ettersom det virket som en dårlig idé å introdusere nye feil og problemer før vi hadde ordnet de allerede eksisterende feilene.

Problemet viste seg å være at Hibernate ikke klarte å hente ut flere *collections* samtidig hvis flere var deklarert med fetchtype som eager. Lazy og eager har noe med når uttrykket vil bli evaluert, hvor lazy vil først evaluere et uttrykk når den trenger det, på den andre siden vil eager evaluere uttrykket med engang uavhengig av når det skal brukes. Av Listing 4.5 er det mulig å se hvordan det var implementert i Joly 1.6. Hvordan dette så ble rettet opp kommer fram av Listing 4.6, hvor det er mulig å se at selv om *fetch* er fjernet fra annotasjonen ManyToMany så har metoden nå en ny annotasjon ved navn *LazyCollection*. Den er da satt til å ikke benytte seg av lazy og vil derfor benytte seg av eager. Dette betyr at det faktisk ikke har skjedd en endring i hvordan den vil hente ut *collections*. Det er litt usikkert hvorfor Hibernate godtar dette slik det er gjort. Den skal fortsatt gjøre det eager, men ved å endre til en annen annotasjon ble problemet løst.

Listing 4.5: Joly 1.6 LazyCollection annotasjon

```
1 @ManyToMany(targetEntity=Employee.class, fetch=FetchType.EAGER)
2 @JoinTable(name="Employee_Course",
3   joinColumns={@JoinColumn(name="courseID")},
4   inverseJoinColumns={@JoinColumn(name="username")})
5 public Collection<Employee> getEmployees(){
6   return employees;
7 }
```

Listing 4.6: Joly 1.7 LazyCollection annotasjon

```
1 @ManyToMany(targetEntity=Employee.class)
2 @LazyCollection(LazyCollectionOption.FALSE)
3 @JoinTable(name="Employee_Course",
4   joinColumns={@JoinColumn(name="courseID")},
5   inverseJoinColumns={@JoinColumn(name="username")})
6 public Collection<Employee> getEmployees(){
7   return employees;
8 }
```

4.3.4 Fjerde endring (ManyToMany, MappedBy)

Dette var den siste feilrettingen som måtte til for at Joly 1.7 skulle kjøre. ManyToMany annotasjonen vist i Listing 4.7 brukes til å spesifisere hvilke felter som skal inkluderes i mange-til-mange relasjon. *mappedBy* er variabelen som skal hentes fra klassen *targetEntity* variabelen. Dette betyr at ifølge Listing 4.7 skal variabelen *employees* hentes fra klassen *Student*. Feilen som oppstod i Joly var at variabelen *employees* eksisterer ikke i klassen *Student*, men i *Course* klassen. Dette gir opphav til nye endringer som vist i Listing 4.8, hvor da dette løste feilen som oppstod.

Listing 4.7: Joly 1.6 ManyToMany annotasjon

```
1 @ManyToMany(targetEntity=Student.class, mappedBy="employees")
2 @LazyCollection(LazyCollectionOption.FALSE)
3 public Collection<Course> getCourses(){
4   return courses;
5 }
```

Listing 4.8: Joly 1.7 ManyToMany annotasjon

```
1 @ManyToMany(targetEntity=Course.class, mappedBy="employees")
2 @LazyCollection(LazyCollectionOption.FALSE)
3 public Collection<Course> getCourses(){
4   return courses;
5 }
```

Tidligere versjon Joly 1.6 ga aldri denne feilmeldingen ved oppstart. Det er veldig spesielt at Joly 1.6 fungerte som den skulle, men vi vil ikke gå noe særlig videre med å finne ut grunnen ettersom feilen er rettet i nyeste versjon av Joly. Samtidig som at det vil ha lite for seg å finne og rette feil i eldre versjoner.

4.4 Overordnede endringer i Joly

Det ble gjort mange endringer i Joly gjennom oppdateringsfasen, men vi har hittil utelatt filendringer, noe som skal beskrives i de neste avsnittene. Det vil også bli inkludert feilmeldinger som oppstod selv om noen av de er allerede dekket i tidligere seksjoner så var omfanget av feilmeldinger større enn det de tidligere seksjonene gir uttrykk for.

4.4.1 Fjerning av rammeverk

De bibliotekene som ikke produserte noen feil ved fjerning tidligere ble satt opp for fjerning i Joly 1.7 i henhold til Tabell 4.1. Bakgrunnen for fjerning av bibliotekene var for å finne ut om alle bibliotekene som kommer med Joly var nødvendige eller ikke, slik at det var mulig å rydde opp. Ettersom det var forvirrende at det lå vedlagte biblioteker som det var usikkerhet rundt om dem faktisk var i bruk. Totalt var det fem biblioteker som skulle fjernes, og vi skal her komme innpå resultatet av dette og hvordan det ble gjort. Disse fem bibliotekene som tidligere nevnt ga ingen feil ved testfjerning slik det ble beskrevet i Seksjon 4.2.1:

- postgresql
- asm
- asm-commons
- jdbc2_0-stdext
- Acegi-security

Fjerningen av utvalgte biblioteker fulgte rekkefølgen gitt i henhold til Tabell 4.1, selv om tidligere planen for utbytting ikke gikk som planlagt. Dermed ble postgresql fjernet først, dette er et bibliotek som benyttes i forbindelse med kommunikasjon med postgresql-databaser. Ettersom Joly ikke benytter postgresql-database ga ikke fjerning av dette bibliotek noen problemer.

Derimot var det større usikkerhet rundt asm og asm-commons bibliotekene, på dette stadiet var det ingen erstatter til disse bibliotekene så de ble satt opp for fjerning på bakgrunn av kriteriet gitt tidligere for fjerning av bibliotek. asm og asm-commons kan brukes til å modifisere eksisterende klasser eller dynamisk generere klasser, direkte på binærform[4]. Fjerning av dette biblioteket viste seg lite problematisk, men det viste seg senere at vi under hele oppdateringsprosessen tilfeldigvis hadde inkludert et annet bibliotek som inneholdt asm og asm-commons. Biblioteket asm fra Spring Framework inneholder begge de tidligere nevnte bibliotekene. Med dette ble asm og asm-commons derfor teknisk sett ikke fjernet, men oppdatert, noe som var det originale målet for oppdateringsprosessen.

Det neste biblioteket var det litt usikkerhet rundt, og dette var jdbc-stdext også kalt Java Database Connectivity Standard Extension API. Denne er en av to

pakker i JDBC 2.0, hvor den andre er core JDBC package. Den sistnevnte inneholder de mest vanlige metodene for hva man vil gjøre med en database. Hvor jdbc-stdext utvider dette spektret med mer funksjonalitet[17] som f.eks. Connection pooling, rowsets og distributed transaction. jdbc-stdext kom tidligere vedlagt i distribusjonen av Hibernate[21], dette tyder på at den ble brukt av eldre versjoner av Hibernate. Dette biblioteket kom ikke distribuert med Hibernate 3.6.10, som tas i bruk i Joly 1.7. Det kan hende at dette medførte at fjerning av biblioteket gikk bra, men at det er mulig at Hibernate ikke bruker det lenger eller at det ligger vedlagt i en anne pakke som Hibernate bruker.

Acegi-security er det siste biblioteket som skulle fjernes. Dette biblioteket går i dag under navnet Spring Security[28] og benyttes til økt sikkerhet når det gjelder autentisering og autorisering for applikasjoner. Dette biblioteket benyttes ikke i Joly, men det var tiltenkt at det skulle tas i bruk ved videreutvikling. Å fjerne det ble gjort uten at noen nye problemer oppstod.

4.4.2 Nye rammeverk i Joly 1.7

Under bytting av rammeverk ble det oppdaget at flere nyere bibliotek etterspurte andre bibliotek som ikke allerede var i Joly. Disse bibliotekene ble lagt til når det oppstod feilmeldinger under forsøk av kjøring av Joly 1.7. De nye bibliotekene som tidligere ikke fantes i Joly 1.6, men som nå er en del av Joly 1.7 kommer fram av Tabell 4.3. Totalt var det nødvendig med syv nye bibliotek for at Joly skulle kjøre ordentlig.

Joly 1.7	**Versjon**
slf4j.api	1.5.3
slf4j-nop	1.5.3
javassist	3.16.0
org.springframework.asm	3.1.0
org.springframework.aop	3.1.0
org.aopalliance	1.0.0
org.springframework.expression	3.1.0

Tabell 4.3: *Helt nye bibliotek i Joly 1.7*

4.4.3 Eksempler på feilmeldinger

Ved bytting av biblioteker som er spesifisert i Trinn 1 i oppdateringsplanen fulgte det med en del feilmeldinger under forsøk på oppstart av Joly. Ett fåtall av disse feilene gjengis i Tabell 4.4 og består i korte trekk av en kjedereaksjon av mangel på klasser og metoder i eldre bibliotek. Dette er ikke fullstendige feilmeldinger i den formen de ble gitt, men er blitt skrevet om for bedre lesbarhet, slik at det er mulig å skjønne hva feilmeldingen antydet.

Feilmelding
Mangler QuerySecondPass fra org.hibernate
Mangler LRUMap fra org.apache.commons
Mangler MethodFilter fra javassist.util
Problem med oppretting av beans med Spring
Mangler org.springframework.asm
Configuration attributt ikke definert

Tabell 4.4: *Eksempler på feilmeldinger ved oppstart av Joly 1.7*

4.5 Jetty

Det ble gjort forsøk på å få Jetty over på nyeste versjon. Dette viste seg å være vanskeligere enn forventet ettersom det er tydelige forskjeller mellom Jetty versjon 5 og 8. Jetty 5 er konfigurert for Joly med en konfigurasjonsfil, denne filen brukes til å konfigurere klassen *Server* for å kunne sette opp f.eks. URL-mapping og HTTPS. *Server* klassen finnes i Jetty 8 under ett annet navn, men det gjør derimot ikke attributter eller subklasser som skal kunne konfigureres. Joly benytter seg altså av fire biblioteker som medfølger distribusjonen av Jetty, som er følgende:

- javax.servlet
- org.mortbay.jetty
- org.mortbay.jmx
- xmlParserAPIs

Det er derfor nødvendig å finne erstatninger for disse fire bibliotekene. Vi mistenker at det har skjedd omstruktureringer i bibliotekene som gjør at de kan muligens gå under andre navn eller at klasser kan ha blitt fjernet. I tillegg til dette må det mest sannsynlig gjøres endringer i konfigurasjonsfilen på grunn av

forandringer i biblioteker. Dette framstår som en tidkrevende oppgave i seg selv, så vi velger ikke å gjøre noe med det nå ettersom Joly 1.7 fungerer med Jetty 5 og de nye rammeverkene. Det hadde vært ønskelig om Jetty var oppdatert samtidig som rammeverkene for å være konsistent når det gjelder vedlikehold.

4.6 Utdaterte moduler

Etter en stund kan utviklerne som står bak åpen kildekode rammeverk og bibliotek gjøre endringer i form av pakker eller klasser som vil bli fjernet i nærmeste fremtid. Som utvikler betyr dette at man burde avvikle å bruke denne pakken eller klassen i applikasjonen, for å gå over til ny metode for hvordan det skal utføres. Dette var tilfellet for mange av kontroller-klassene i Joly.

Etter at Springbibliotekene ble oppdatert i Joly 1.7 medførte dette at SimpleFormController fra web-modulen til Spring ble utdatert. SimpleFormController ble fra og med Spring 3.0 utdatert til fordel for annotasjonsbaserte-kontrollere[20]. Dette betyr at den burde avvikles for bruk i applikasjoner, inkludert Joly. Ut ifra tidligere API dokumentasjon[19] vil det kunne stå API'et hvilken versjon av Spring de vil bli fjernet helt. For Spring 3.1 API står det ikke nevnt når SimpleController vil bli fjernet. Dette kan tyde på at SimpleFormController vil kunne brukes i en god stund framover. Om dette ikke er tilfellet så må det gjøres endringer i alle klassene som benytter SimpleFormController. I Joly er dette de fleste kontrollerne og byr derfor på en del arbeid og utfordringer.

Endringen består hovedsakelig av å gå over til annotasjonsbaserte-kontrollere. Dette vil være nødvendig for omlag 25 av filene i Joly som benytter seg av SimpleFormController. Dette vil nok være tidkrevende og vi velger derfor beholde SimpleFormController på bakgrunn av at det ser ikke ut til at den vil bli fjernet med det første. Selv om de eldre filene ikke blir oppdaterte så velger vi å benytte oss av annotasjoner for kontrollere i videreutvikling i forbindelse med ny funksjonalitet. Dette er for ikke å lage mer arbeid enn nødvendig når SimpleFormController blir oppdatert på ett senere tidspunkt.

4.7 Oppsummering

Alle feilene som oppstod i kildekoden under oppdateringsprosessen har blitt rettet slik at Joly 1.7 kjører nå med nye rammeverk og fungerer slik det skal hvis vi skal gå ut ifra testingen som er blitt gjort. Joly 1.6 hadde totalt 34 biblioteker, dette har nå økt til 42 biblioteker for Joly 1.7. Vi har dermed en oppdatert versjon av Joly som vil bli utgangspunktet for videre utvikling.

4.7.1 Teknisk

Hovedårsaken til feilene i kildekoden skyldtes javax.persistence og Hibernate, m.a.o. det som hadde med databaseaksess å gjøre. Dette ble rettet opp i Seksjon 4.3 og skal fungere. Det oppstod også nye problemer som at klassen SimpleFormController som benyttes i nesten alle kontrollerne i Joly, er utdatert for den nye versjonen av Spring som er tatt i bruk. Den kan fortsatt brukes inntil videre, men java-filene som bruker denne burde oppdateres etter hvert. Det andre problemet var at det var vanskelig å finne bibliotek i trinn 9 i Tabell 4.1, disse medfølger Jetty og var ikke det samme for nyere versjon.

De rammeverkene som nå finnes i Joly 1.7 er de som er i henhold til Tabell 4.1, men som da ikke står med uendret eller fjerne, i tillegg gjelder dette bibliotekne vist i Tabell 4.3. Disse er nå det absolutte minimumet av rammeverk og bibliotek Joly behøver for å fortsette og være operativt. Joly vil fortsette å bruke Jetty 5 ettersom de eldre bibliotekene Joly 1.6 benytter seg av ikke lenger eksisterer i Jetty 8. Jetty 5 er uheldigvis utdatert, men den ser ikke ut til å ha noen problemer med å kjøre Joly 1.7 tross nye rammeverk. Det som er å utsette på denne oppdateringsprosessen er mangel på testing. Joly har blitt testet med enkel funksjonalitet og det fungerer slik de skal, men det er ukjent for oss om det kan ha oppstått andre problemer eller feil som vi ikke vet om. Selv om det er gjort en del oppdateringer til Joly gjenstår det migrering fra Hibernate 3 til Hibernate 4, utbytting av Jetty og retting av utdaterte metoder i forbindelse med Spring. Dette er uheldig, men ettersom Joly 1.7 fungerer slik det er så vil vi ikke gå videre med andre oppdateringer på grunn av mangel på tid.

Fremgangsmåten for å finne rammeverk til å oppdatere de viste seg å fungere veldig bra. Tanken bak å lage en plan for hvilken rekkefølge rammeverkene skulle byttes i var i etterkant unødvendig med tanke på at det oppstod flere og flere

feilmeldinger som hadde med avhengigheter til diverse andre bibliotek som den ikke fant. En mulig bedre løsning for å kunne sette opp en slik plan kunne vært å forsøke og identifisere hvilke biblioteker som kanskje avhenger av hverandre, for å så bytte de samtidig; men dette i seg selv kunne nok tatt i overkant for lang tid og være en unødvendig prosess.

Generelt sett fikk vi gjort det som var oppgaven innenfor en kort tidsperiode. Selv om hvis man ikke oppdaterer rammeverk med korte mellomrom vil det nok kunne skape større problemer. De løsningene som ble presentert i Seksjon 4.3 var resultatet av prøving og feiling av litt forskjellige annotasjoner som virket sannsynlig ville fungere, noe det til slutt gjorde. Under et tidspunkt i oppdateringen ble det eksperimentert med Hibernate 4, dette ble så avsluttet p.g.a. andre diverse feil oppstod ved bytting og vi fant det unødvendig å introdusere flere faktorer til det som allerede var forvirrende av hva som skulle rettes opp.

4.7.2 Generelle rundt bytting

Etter å ha gått igjennom en oppdateringsprosess, sitter vi igjen med tildels oppdatert og fungerende versjon av Joly. Det ble nødvendig å oppdatere filer med ny syntaks og gjøremåte, p.g.a. at Joly benytter seg av rammeverk. Dette viser til at det kan oppstå problemer ved bruk av åpen kildekode rammeverk, hvis applikasjonen ikke følges opp med kontinuerlig vedlikehold. Etter 5 år (hvis ikke mer) ble Joly oppdatert, og i løpet av denne perioden hadde flere rammeverk blitt oppdatert. Noe som medførte forandringer i måten annotasjoner skulle benyttes.

Spørsmålet er om vi skal klandre rammeverkene for at de videreutvikles og vedlikeholdes, eller om det skal stilles spørsmål ved mangel av rutinemessig oppdatering av Joly. Før vi svarer på dette er det viktig å se på hvordan disse rammeverkene egentlig påvirker applikasjonen. Som det var mulig å se krevde tidligere eldre versjoner en spesifikk syntaks for å fungere. Ved nyere versjoner hadde dette blitt endret som medførte høyst nødvendig endringer for at Joly skulle bli operativt igjen. Dette virker nesten mot sin hensikt med tanke på at rammeverk skal gjøre utvikling lettere. At det gjøres endringer mellom rammeverk-versjoner som kan medføre at brukeren må skrive om logikken i en applikasjon er noe som burde gjøre at lysten til å bruke rammeverk minker. For å komme tilbake igjen

på selve problemet som oppstod med Joly så ville nok mye av forvirringen rundt de forskjellige rammeverkene kunne ha blitt unngått hvis Joly hadde hatt regelmessige oppdateringer. Riktignok fikk vi en oppgave på grunn av dette, men at det sees på som lite fornuftig at en slik oppdatering ikke er gjort tidligere.

Til slutt kan det nok sies at hvis man ikke har planer om å regelmessig vedlikeholde applikasjoner så vil det være lurt å styre unna slik åpen kildekode rammeverk. Men i de situasjoner hvor vedlikehold prioriteres vil det nok ikke være problematisk å ta i bruk rammeverk. For tross nyere versjoner av rammeverk og behovet for å oppdatere kildekode så vil det ikke ha gått flere år imellom, så det å finne informasjon om hvordan man går over fra versjon til versjon vil potensielt være lettere. Det finnes nok ikke noe fasitsvar på hva regelmessig vedlikehold er, men for Joly kan det antydes at 1-2 ganger i løpet av ett år burde holde.

Kapittel 5

Joly som godkjentlistesystem

I dette kapitlet skal vi gå nærmere inn på utvidelser for Joly. Det er planlagt at Joly skal kunne benyttes som et godkjentlistesystem og i denne forbindelse vil det bli utviklet et brukergrensesnitt for at gruppelærere enkelt skal kunne rette og eventuelt godkjenne innleverte obligatoriske oppgaver. Dette vil bygge videre på det som var presentert i Seksjon 3.1. Vi skal i de neste seksjonene se på beskrivelsen og implementasjonen av følgende funksjonaliteter:

- Oversikt over innleverte obligatoriske oppgaver
- Godkjenning av eldre obliger
- Godkjentliste
- Bytting av gruppe for studenter

Dette er funksjonalitet som vil være tilgjengelig for følgende innloggede brukergrupper; Administrator, foreleser og gruppelærer. Fleste av disse er ment for gruppelærere med unntaket av godkjentlisten som fakultetet har interesse av, men som står under funksjonalitet for foreleser. Gjennom hele dette kapitlet vil vi kalle de obligatoriske oppgavene som studentene leverer inn for *obliger*. Mens obligatoriske oppgaver opprettet av foreleser vil være akkurat det samme, altså obligatorisk oppgave.

5.1 Databaseendringer

Joly mangler foreløpig et felt tilknyttet innleverte obliger som tilsier om den er godkjent. En utvidelse av tabellen *Studentsolution* vil løse dette problemet og vi velger derfor å utvide denne eksisterende tabellen med en ny kolonne kalt *status*.

Med denne nye kolonnen vil vi kunne finne ut om statusen for innleverte obliger er f.eks. godkjent eller ikke. Av Tabell 5.1 kan vi se hvilke statuser denne nye kolonnen skal inneholde. Vi velger å benytte oss av tallverdier for disse, av typen int. Standardverdien til eksisterende rader vil bli satt til *ikke rettet* ettersom det er variasjon mellom innleverte obliger som er *godkjent* og *ikke godkjent*, men som vi skal se senere valgte vi å gjøre det på en annen måte.

Status	**Statusnr.**
Godkjent	0
Levere på nytt	1
Ikke rettet	2
Ikke levert	3
Ikke godkjent	4

Tabell 5.1: *Statusverdier*

Med den nye kolonnen **status** vil tabellen **Studentsolution** bli slik det kommer fram av Tabell 5.2.

Studentsolution
studentsolutionID
text
deliverytime
bytesize
assignmentID
username
status

Tabell 5.2: *Tabell Studentsolution*

Det ble tidligere vurdert at det ikke var behov for alle statusene som vi kan se i Tabell 5.1, noe vi skal komme tilbake til hvorfor i Seksjon 5.2.1. I Joly er det ingen mulighet til å assosiere gruppelærere med kursgrupper, så tanken var at gruppelærere skulle kun få opp sin egen gruppe som det var mulig å gjøre endringer på. Dette ble senere forkastet med tanke på at det kunne hende at gruppelærere fikk ansvaret for to grupper eller måtte hjelpe med retting på en spesifikk obligatorisk oppgave av ulike årsaker. Ved å unngå at gruppelærerne har kun ansvar for sin gruppe og er låst til det, slipper administrator eller foreleser å legge til denne relasjonen hver gang det ville vært nødvendig.

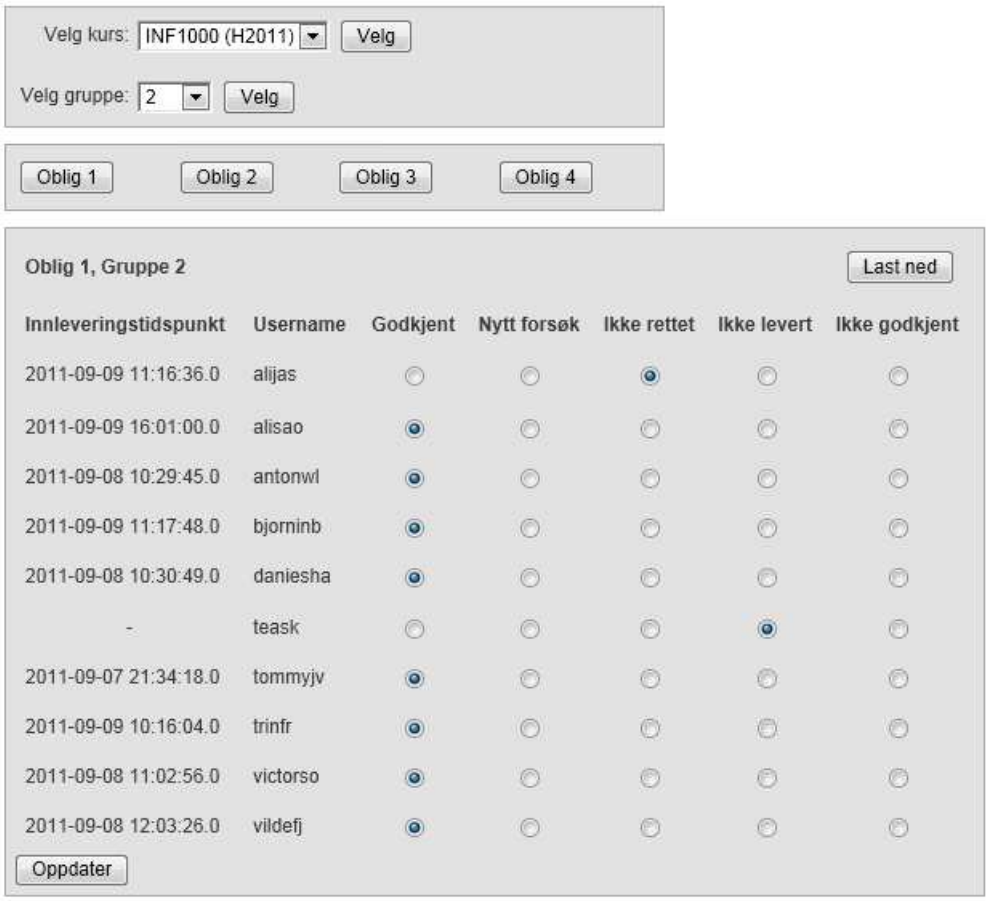
5.2 Oversikt over innleverte obliger

Denne funksjonaliteten er den som er mest ønskelig å kunne ha inkludert i innleveringssystemet Joly. Med et slikt oppsett vil gruppelærere gjennom et enkelt brukergrensesnitt kunne endre status på en oblig, om den er *ikke levert*, *godkjent* osv.. For så å generere en godkjentliste på slutten av semesteret som vil inneholde alle studentene som har fått godkjent alle obligatoriske oppgaver ved kurset, en funksjonalitet vi skal komme tilbake til i Seksjon 5.4. Denne nye funksjonaliteten er viktig fordi det er et stort antall studenter og et automatisert system som dette vil gjøre jobben enklere. Det første som møter brukere av denne funksjonaliteten er det mulig å se i henhold til Figur 5.1. Her er det valgt å la det være uten restriksjoner av hvilke kurs og grupper det er mulig å aksessere.

Figur 5.1: *Hovedside for visning av obliger*

Valg av kurs og gruppe vil medføre resultat lignende Figur 5.2, her vises alle studentene på en gruppe med tilhørende status på innleverte obliger i form av radiobuttons. Det er mulig å bytte mellom obligatorisk oppgaver til kurset. Hvis en obligatorisk oppgave ikke er opprettet vil den ikke vise noe. Det også mulig å laste ned alle innleverte obliger for en gruppe ved å trykke på knappen *Last ned*. Sistnevnte funksjonalitet er implementert med tanke på antall studenter som er på hver gruppe og antall innleveringer som blir gjort ved hver obligatorisk oppgave. Ved hver innlevering fra studenter vil det sendes en mail til gruppelæreren som inneholder informasjon som; brukernavn, innleveringstidspunkt, vedlagt oblig osv. Det kan derfor være tidkrevende for gruppelæreren å finne nøyaktig den siste innleverte oppgaven fra alle studentene. Men med muligheten til å laste ned alle obligene vil de ikke trenge å gå igjennom innboksen for å finne alle innleverte obliger.

Ved nedlastning er det mulig å laste ned et zip-fil som inneholder alle innleverte obligene til studentene for denne obligatoriske oppgaven. Disse filene har



Figur 5.2: *Visning av obliger etter valgt kurs og gruppe*

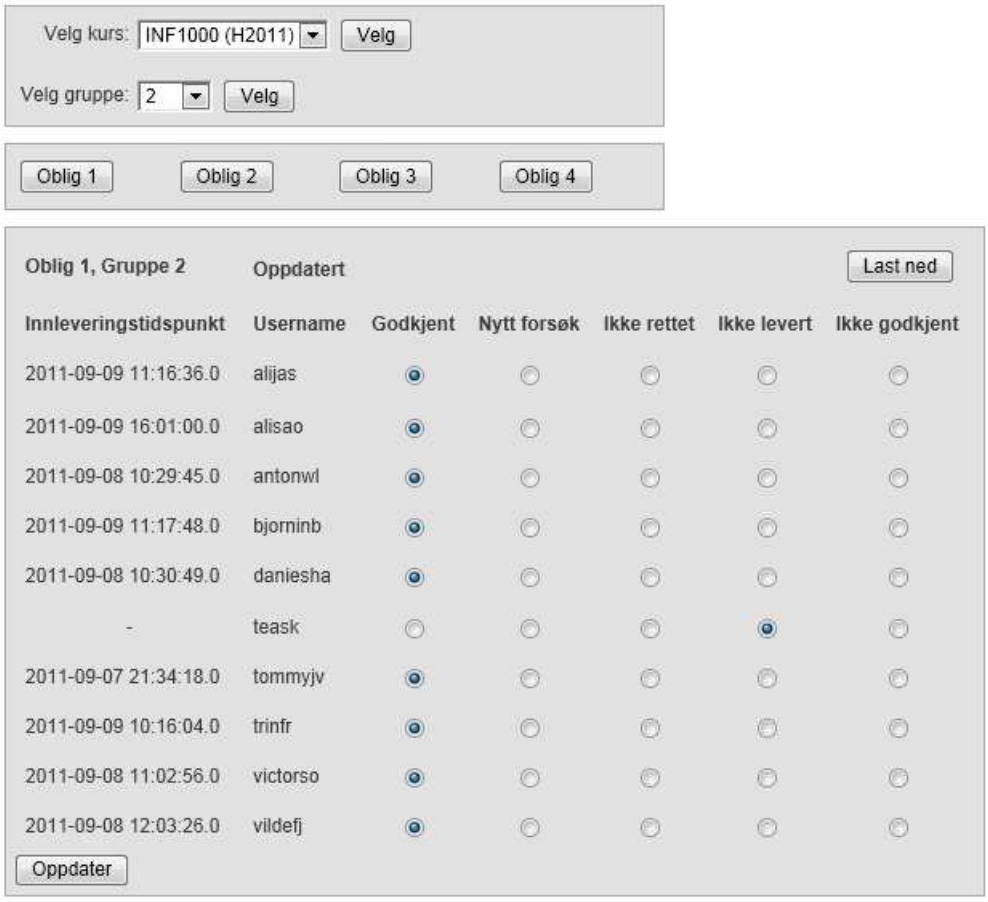
filnavn tilsvarende brukernavn fra studenter, slik at det skal være lett å vite hvem som er hvem. Dette vil medføre at gruppelærerne må endre filnavn for å kompilere og kjøre programfilen, men det er i grunn ingen mer delikat måte å gjøre dette på hvis det samtidig skal kunne være oversiktlig for gruppelæreren. Brukernavn og gruppe er nevnt i filen så gruppelæreren mister på ingen måte oversikt over hvilken student som blir rettet.

Listen av innleverte obliger blir hentet fra databasen og baseres som sagt på valgt kurs og kursgruppe. Først vil den finne alle studenter som er på denne gruppen, så hentes alle obliger for obligatorisk oppgave 1 ut, som er standard operasjonen. Som det er mulig å se ut ifra Figur 5.2 er det mulig å bytte mellom obligatorisk oppgave 1-4 ved å trykke på knapper. Når alle obligene har blitt hen-

tet ut vil den beholde den siste obligen levert av hver student. Dette blir gjort ved at alle andre innleveringer med et innleveringstidspunkt tidligere enn den valgte vil bli ignorert, dette er fordi studenter har mulighet til å levere en forbedret versjon av obligen før tidsfristen har gått ut. De som ikke har levert vises kun med brukernavn og er flagget som *ikke levert*. I tillegg vises kun en enkelt bindestrek som erstatter innleveringstidspunktet, og som det er mulig å se ut ifra Tabell 5.2 er dette tilfellet for studenten med brukernavnet "teask". Dette er kun dummy objekter som det ikke er mulig å endre på, så hvis status endres på noen av disse vil det ikke blir registrert. Det er derfor helt nødvendig å levere en oblig for å kunne endre statusen.

For visning av data i denne listen av innleverte obliger benyttes det en mer utradisjonell måte. Grunnen til dette var at på tidspunktet ved implementering var dette en enkel løsning og at en annen løsning var vanskelig å finne innenfor rimelig tid. Det benyttes to forskjellige lister, hvor begge listene er helt identiske, mens den ene listen benyttes kun til visning av data mens den andre er en liste i command-objektet som benyttes til å ta imot nye endringer. Den sistnevnte listen brukes til å vise nåværende status for obliger. Command-objekt benyttes til kommunikasjon mellom JSP og kontroller, dette ble forklart i Seksjon 2.5.1 som omhandlet kommunikasjonen mellom bruker og applikasjon. Å "skrive ut" data fra dette command-objektet viste seg vanskelig og derfor benyttes den andre listen til å lettere skrive ut all annen data enn det som er relatert til statusfeltet som innleveringstidspunkt og brukernavn. Det finnes nok en bedre måte å gjøre dette på, men det ville kun være kosmetiske endringer i kildekoden og har ingenting å si for applikasjonens funksjonalitet. Det er derfor ikke blitt gjort noe videre med det. De genererte listene med obliger ble lagret på server-siden av visse årsaker. Dette viste seg å skape problemer når flere innloggede brukeregrupper aksesserte og benyttet seg av samme side samtidig, men dette skal vi komme tilbake til litt senere.

Det er herfra mulig å sette ny status for innleverte obliger for å så oppdatere disse i databasen. Som det kommer fram av Figur 5.3 vil det vises *Oppdatert* ovenfor listen når brukeren trykker på Oppdater-knappen. Den vil ikke ta hensyn til om det er gjort endringer i databasen, men kun om Oppdater-knappen har blitt trykket. Grunnen til dette er at det kan være vanskelig å holde oversikt over alle statusene og hvis gruppelærerne ikke husker helt hvilke obliger som har



Figur 5.3: *Visning av obliger etter oppdatering*

blitt rettet, men velger å trykke oppdatere for å være sikker på at oblig-statusen er oppdatert så er det praktisk å få en tilbakemelding på dette.

Det er viktig at gruppelærerne ikke retter obliger før innleveringsfristen på de obligatoriske oppgavene ettersom status er koblet opp mot den innleverte obligen. Ettersom studenter har mulighet til å levere inn forbedret versjon fram mot innleveringsfristen, noe det for øvrig er stor sannsynlighet at de gjør. Dette kan derfor medføre ekstra arbeid for gruppelærerne siden tidligere rettinger av eldre innleverte obliger vil ikke vises i denne oversikten siden det eksisterer en nyere innlevering. Et annet aspekt ved denne nye funksjonaliteten og opprettelse av statusfeltet i databasen gjør at det er mulig å føre statistikk ettersom Joly brukes år etter år. Dette vil kunne gi konkrete tall på frafall på f.eks. hver obligatoriske

oppgave eller totalt ved slutten av semesteret. Selvfølgelig krever dette at gruppelærerne oppdaterer slik de skal eller så vil dette medføre at dataene ikke blir helt korrekte.

5.2.1 Designendringer

Under utviklingsprosessen var det nødvendig å gjøre endringer i tidligere design idé som spesifiserte kun fire statusfelt, i motsetning til fem statusfelt som ble sluttresultatet. Feltet som opprinnelig ikke skulle være med var *ikke levert* fordi det var tiltenkt at studenter som ikke hadde levert ville ikke vises i listen over innleverte obliger. Dette ble så endret til at de som ikke hadde levert ville i stedet vises med brukernavn og at statusen er satt til *ikke levert* slik det spesifisert i forrige seksjon. Grunnen til denne endringen var for at gruppelærerne skulle enklere kunne se hvilke studenter som ikke har levert obligen sin.

5.2.2 Vesentlig problemer

Det oppstod noen problem under utvikling av funksjonaliteten *oversikten over innleverte obliger*. Noe gjenstår å bli rettet opp, men vil ikke være et stort problem for selve funksjonaliteten.

Tegnsett

Det viste seg å være et problem når det gjaldt nedlastning av filer. Filer som inneholdt de norske bokstavene æ, ø og å ville det forekomme at disse er erstattet med et spørsmålstegn som er noe uheldig. Dette gjelder kun innleverte obliger som hentes fra databasen, mens som vedlegg i mailen som sendes til gruppelærer ved hver innlevering eksisterer ikke dette problemet. Fra tidligere virker det som dette aldri var et problem, for Joly hadde allerede funksjonaliteten for å se på innleverte obliger, men selv disse viser spørsmålstegn. Så hvis dette er et nylig problem kan det være stor mulighet for at det kan være forårsaket av (et av de) oppdaterte rammeverkene. For å være sikker ble det sjekket hva slags tegnsett databasen benytter, som viste seg å være utf8, et tegnsett som skal støtte de norske bokstavene som får oss til å tro at feilen ikke ligger der. I javakoden ble det forsøkt å hente ut data fra databasen for å så skrive disse obligene til fil ved å benytte tegnsettene; utf8 og ISO-8859-1, men uten hell. Det ble ikke funnet noen løsning på dette så det vil være en av få mangler som kan ha blitt introdusert

ved videreutvikling av Joly. Må påpeke at dette også er tilfellet for godkjenning av eldre obliger, funksjonalitet som blir dekket i Seksjon 5.3.

Annotasjoner

Med nye rammeverk for Spring hadde dette innflytelse på hvordan kontrollere skulle programmeres. Dette betydde at annoterte kontrollere var å anbefale siden dette ble innført i Spring 2.5, og at tidligere superklasser av kontrollere i springversjonen som ble brukt i Joly 1.6 er utdaterte. Det var usikkert hvordan det var å gå over til annotasjoner i Joly, siden det var nødvendig at tidligere kode fungerte sammen med annotasjoner. Men det viste seg å være helt uproblematisk, som vi nå skal se nærmere på. Joly har nå to typer implementasjoner av kontrollere, de tidligere eksisterende som benytter utdaterte superklasser og ny funksjonalitet implementert med annotasjoner. For at Spring skal kunne benytte seg av disse annoterte-kontrollerne er det nødvendig å konfigurere Spring til å gjøre dette, noe som kommer fram av Listing 5.1.

Listing 5.1: Konfigurasjon for annotasjon

```
1 <context:component-scan base-package="uio.ifi.joly.web"/>
2 <mvc:annotation-driven/>
```

Disse to setningene forteller Spring til å lete etter annoterte kontrollere i pakken ”uio.ifi.joly.web“ med annotasjonen ”@Controller“. Når det gjelder ”<mvc:annotation-driven/>“ så vil vi komme tilbake til det i neste avsnitt ettersom det er relevant der. Det å programmere kontrollere med annotasjoner krevde annoterte deklarende setninger i selve kontrolleren slik at den defineres som en kontroller slik det vises i Listing 5.2. Her definerer annotasjonen *Controller* klassen som en kontroller.

Listing 5.2: Annotasjon for en kontroller

```
1 @Controller
2 @RequestMapping("/admin/switchGroup.html")
3 public class SwitchGroupForStudentsController {
4   ...
5 }
```

Ved aksessering av URL i Joly som krever innlogging vil Joly videresende brukeren til en ”loginInterceptor“ klasse som det er mulig å se i Listing A.9. Denne

sjekker om brukeren er innlogget og har en gyldig session, hvis ikke vil brukeren bli videresendt til innlogging-siden slik at man kan logge inn. Dette kontrolleres ved hjelp av mapping av URLer som er spesifisert i konfigurasjonsfiler. Med implementasjon av nye klasser med annotasjoner benytter denne sin egen mapping av URLer som er resultatet av setningen ”<mvc:annotation-driven/>“ som unngikk denne videresendingen til innlogging-siden hvis brukeren ikke var logget inn. Dette betydde at det var nødvendig å ”override“ denne standard mappingen slik det vises i henhold til Listing 5.3.

Listing 5.3: Konfigurasjon av logininterceptor for annotasjoner

```
1 <mvc:interceptors>
2   <mvc:interceptor>
3    <mvc:mapping path="/report/**"/>
4    <mvc:mapping path="/admin/**"/>
5    <bean class="uio.ifi.joly.web.LoginInterceptor" />
6   </mvc:interceptor>
7 </mvc:interceptors>
```

Her spesifiseres det at hvis URLen begynner med */report/* eller */admin/* etterfulgt av hva som helst vil så bli sendt til klassen som er går under variablen bean class som i dette tilfellet blir LoginInterceptor som ligger i pakken uio.ifi.joly.web.

Server-side-lagring

Da Joly ble satt i drift høsten 2012 i INF1000, ble det oppdaget en feil ved bruk av rapportfunksjonalitet. Dette ble rapportert av årets gruppelærere som benyttet seg av siden for *oversikt over innleverte obliger* samtidig som gjorde at de fikk hverandres sesjonsvariabler. Grunnen til dette var at vi nyttet server-side-lagring av variabler, noe som ødela muligheten for flere innloggede brukere å kunne bruke siden samtidig. Dette gjaldt 3 kontrollere som hadde blitt implementert med samme metode. Noe som medførte mye arbeid for å rette opp ettersom en av de store grunnene til at dette ble valgt var at det var den enkleste og på det tidspunktet eneste måten å gjøre det på. Dette var uten visshet om at det ville føre til server-side-lagring. Siden det meste av testingen var gjort av bare en bruker om gangen ble det ikke oppdaget før det ble satt i drift. En annen årsak var nok mangel på erfaring rundt bruk av JSP og kontrollere gjorde det ikke lettere ettersom det viste seg at det eksisterer kun et kontroller-objekt som håndterer forespørsler fra alle brukere, noe vi ikke var klar over. Vi hadde den forståelsen at den lagde

et kontrollerobjekt for hver sesjon (session), eller at de variablene som eksisterte i kontrolleren hvertfall var variabler bundet til en sesjon og ikke delt mellom alle.

Det ble løst ved at ved hver forespørsel til kontrolleren ville command-objektet inneholde all informasjonen den trenger for å kunne utføre beregninger på serversiden. Dette command-objektet inneholder derfor all nødvendig informasjon for å kunne hente ut data som skal bli endret i skjemaet. Når dette command-objektet sendes mellom JSPen og kontrolleren, så oppfører dette objektet seg forskjellige avhengig av hvilken vei det sendes, hvertfall på måten det brukes til her. Fra kontroller til JSP vil data lagt inn i command-objektet eksistere når det kommer til JSPen og er mulig skrive ut osv.. Men derimot andre veien, fra JSP til kontroller så vil felter som den ikke får input på bli blanke altså nullverdi. Det er et problem når vi ønsket å bevare data mellom JSP og kontrolller hele tiden. Løsningen på dette var å benytte oss av *hidden* valg for variabler, som vil medføre at den ikke fjernet visse data merket med *hidden* i command-objektet når det ble sendt fra JSP til kontroller. Listing 5.4 viser til hvilke felt som inneholder informasjon som er nødvendig å ha når command-objektet returnerer tilbake til kontrolleren.

Listing 5.4: Nødvendige skjulte felter

```
1 <form:hidden path="course"/>
2 <form:hidden path="group"/>
3 <form:hidden path="oblignr"/>
4 <form:hidden path="assignment"/>
```

Disse variablene inneholder nok informasjon til å identifisere gruppen som studentene deltar på, slik at det kan gjøres korrekte endringer når det gjelder statusen på innleverte obliger.

5.3 Godkjenne eldre obliger

Her vil det være mulig å søke på studenter å få se innleverte obliger for den studenten. Dette har to bruksområder, det skal være mulig å se at studenten har tatt kurset før og at det er mulig for gruppelærere eller foreleser å avgjøre om studenten har bestått visse obligatoriske oppgaver. Figur 5.4 viser til hva brukeren ser ved å klikke seg inn på siden. Her er det mulig å taste inn brukernavn og få fram obliger fra både INF1000 og INF1010 (Objektorientert programmering). Eksisterer det ingen bruker med dette brukernavn vil en tilbakemelding bli gitt,

dette gjelder også hvis det er ingen obliger innlevert.

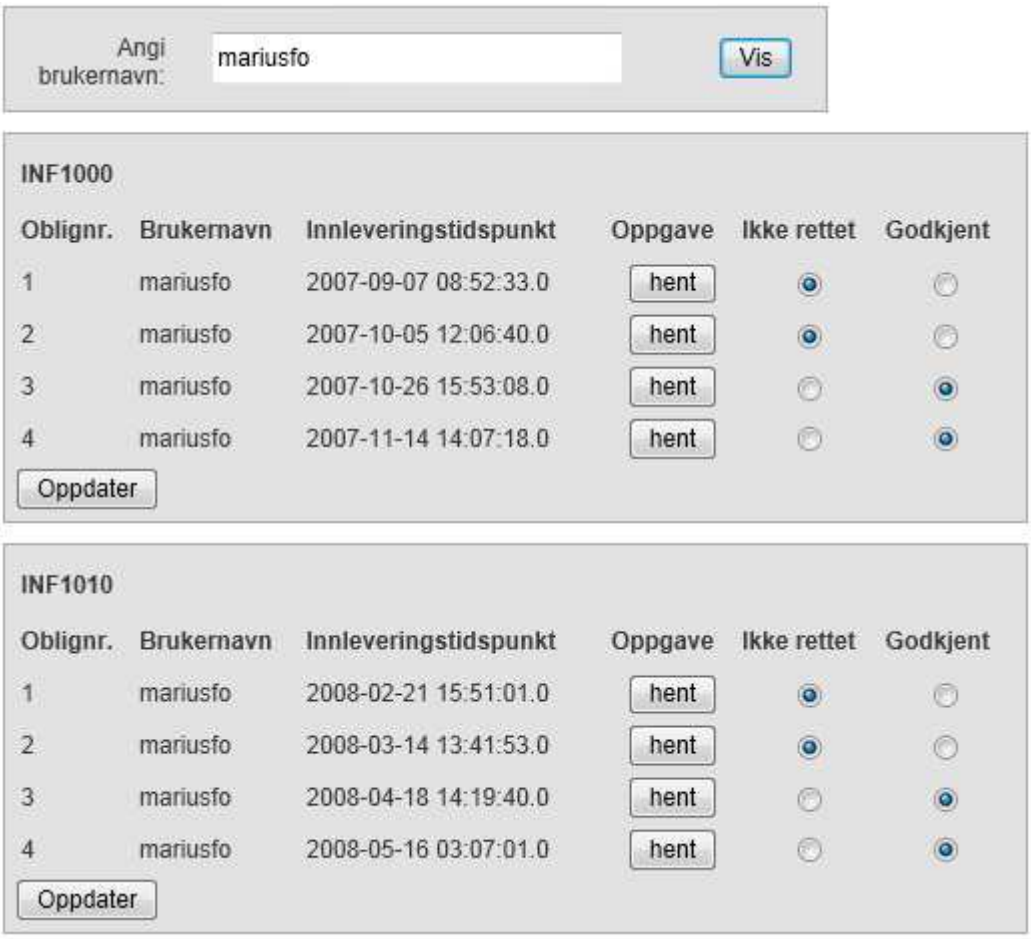
Figur 5.4: *Hovedsiden for visning og oppdatering av eldre obliger*

Figur 5.5 viser tilfellet hvor en student har innlevert obliger i både INF1000 og INF1010. Her vil listen vise siste innleverte obligen fra hver obligatoriske oppgave. Det er mulig å så velge og laste ned innleverte obligen for å kunne evaluere om den burde godkjennes eller ikke. Statusen det er mulig å velge mellom er *ikke rettet* og *godkjent*. Grunnen til kun disse to valgene er på bakgrunn av at alle eldre obliger fikk statusen satt til standardverdien *ikke rettet* etter at det nye feltet *status* ble opprettet i tabellen studentsolution. Men det oppstod et problem, noe vi nå skal se nærmere på.

5.3.1 Problemer

Mye av funksjonaliteten rundt å godkjenne eldre obliger var at etter innføringen av status-feltet i tabellen studentsolution ble alle eksisterende obliger i databasen satt til *ikke rettet* som dermed ville være standardverdien for en innlevert oblig. Som tidligere nevnt vises kun to statusfelter noe som viste seg å bli et problem ved semesterstart hvor alle tidligere innleveringer sin status ble satt til godkjent.

Et annet problem som ble klart i etterkant var at hvis en student har tatt kurset tidligere og har godkjente obliger for noen av de obligatoriske oppgavene; og er tatt opp som student for et nytt semester vil ikke tidligere innleverte obliger være godkjent for de nye semesteret. Dette betyr at hvis studenten har innlevert disse obligene tidligere er det nødvendig for den ansvarlige for gruppen å sjekke om en oblig er innlevert og avgjøre om den blir godkjent. For så å legge den obligen inn for studenten for det nye semesteret. Dette er høyst nødvendig ettersom funksjonalitetene *oversikt over innleverte obliger* og *godkjentlisten* bruker data fra valgt semester. Dermed vil godkjenne eldre obliger kun fungere som en måte å sjekke om studenten har tatt kurset før og kanskje innlevert obliger.

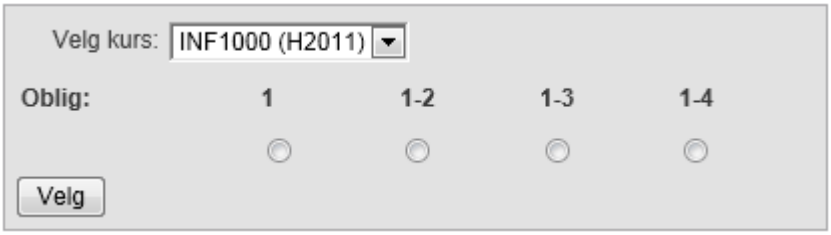


Figur 5.5: *Visning av obliger for student*

5.4 Godkjentliste

Denne funksjonaliteten er ment for å gjøre det enklere for foreleser av kurset og fakultetet på slutten av semesteret å få en ferdig generert liste over de som har bestått alle obligatoriske oppgavene i kurset. Funksjonaliteten er ment for kun foreleser, men vi så allikevel ikke det nødvendig med restriksjoner for å benytte denne funksjonaliteten, som betyr at alle innloggede brukergruppene har tilgang til denne. Det har for så vidt ikke noe å si siden dette er informasjon som de egentlig ikke får gjort noe med. Hovedsiden vises i Figur 5.6, hvor det velges hvilket kurs de ønskes å generere godkjentliste for. Brukeren må velge hvilke obligatoriske oppgaver den skal inkludere. Den ene grunnen til at det er mulig å velge hvilken obligatorisk oppgave er at det er ønskelig å kunne se fremgang

underveis i semesteret å se hvordan det går. Den andre grunnen er at det åpner muligheten for at hvis forelesere for INF1000 velger å ha f.eks. tre obligatoriske oppgaver i stedet for fire som er normal praksis, så vil det da være mulig å velge obligatoriske oppgave 1-3 i stedet for at programmet bare sjekker obligatoriske oppgave 1-4 som ville vært tilfellet hvis vi ikke hadde vært oppmerksomme på dette. Selve formatet på godkjentliste-filen som blir generert vil vi komme tilbake til snart.



Figur 5.6: *Hovedsiden for godkjentliste*

Det å kunne tilby å velge kurs fra hvilket som helst år vil i første omgang ikke ha noen fordeler siden databasen inneholder innleverte obliger uten riktig satt status. Men det vil i de senere årene, hvor forhåpentligvis Joly fortsatt er i drift, kunne fungere som et verktøy for å hente ut interessant statistikk. Som kan brukes til å se om f.eks. frafallet av studenter er stort ved oblig 3, eller kanskje oblig 4. Som kanskje vil kunne ha effekt på vanskelighetsgraden på obligatoriske oppgaver for senere år eller at det kun vil bli brukt som informasjon uten at det trenger å medføre noen analytisk del.

Når det gjelder implementasjonen, ved valgt kurs og hvilke obligatoriske oppgaver den skal ta hensyn til. Vil Joly ta to tallverdier, kurs og oblignummer for å kunne hente ut alle brukernavn fra innleverte obliger med status godkjent, som da har tilhørighet til kurset og opp til det valgte oblignummeret. Etter uthenting av godkjente studenter fra obligatorisk oppgave 1, vil godkjente innleveringer fra de andre obligatoriske oppgavene bli hentet ut og sammenlignet med den tidligere listen for å finne ut om studenten har bestått tidligere obligatoriske oppgaver. Til slutt vil det være en liste som inneholder alle studenter som har bestått et x antall obligatoriske oppgaver avhengig av hvilke oblignummer som ble valgt. Denne listen med brukernavn vil så bli skrevet til en fil som vil være mulig å laste ned for foreleser. Formatet på denne filen er det mulig å se i henhold til Listing 5.5.

Listing 5.5: Eksempel på godkjentliste fra GodkjenteStudenter.txt

```
1 Godkjente studenter for oblig 1-4:
2 V2010, INF1000, Grunnkurs i objektorientert programmering
3 haavstr,
4 hennihor,
5 catod,
6 henrste,
7 krisbrox,
8 hanieha,
9 eirikkst,
10 polinapr,
```

Denne filen vil dermed inneholde nyttig informasjon som er nødvendig for å vite hvilket kurs den er relatert til. Hvilke obligatoriske oppgaver det gjelder, hvilket semester, kurskode og navn på kurset. Navnene vil bli separert ved bruk av komma slik at det enkelt kan importeres inn i excel.

5.4.1 Problemer

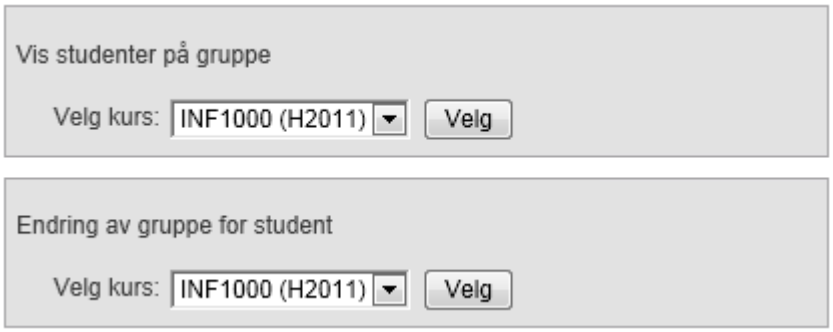
Ved semesterstart hvor denne nye funksjonaliteten skulle tas i bruk var det et ønske om at alle eksisterende obliger i databasen skulle settes til godkjent. Uansett om noen mest sannsynlig ikke var godkjent selv om de ble levert, men at det teknisk sett ville medføre mer riktig enn galt. Dette førte til at den tidligere implementasjonen av oppretting av godkjentlisten ikke fungerte helt som den skulle. Tidligere implementasjon tok høyde for at det kun ville være en innlevering av obligatorisk oppgave 1 per student som var godkjent, men siden flere innleveringer fra samme student hadde blitt satt til godkjent så endte det opp med at disse ble med i listen over godkjente student og skapte duplikater. Dette ble løst ved at denne listen med duplikater ble filtrert for duplikater og returnert.

5.5 Endring av gruppe for student

Det var ikke planlagt å inkludere denne funksjonaliteten til og begynne med, men dette var noe som ble forespurt av veileder Arne Maus å implementere. Den skulle i all enkelhet bare bytte gruppe for studenten, en funksjonalitet som merkelig nok ikke allerede eksisterte i Joly. Å kunne bytte gruppe er greit å ha muligheten til, ettersom studenter har muligheter til å bytte gruppe ved semesterstart og som kan var opptil 4 uker framover. Mengden studenter som velger å bytte gruppe

er stor nok til at en slik funksjonalitet gjør det lettere for de som skal administrere kurset og at listene over gruppene reflekterer studenter som faktisk er på gruppen, noe Joly ikke har gjort tidligere. Hvis studenter hadde ønske om å bytte gruppe før, var ikke dette mulig og det endte med at i realiteten fikk de lov til å bytte ved at de gikk på en annen gruppe, men dette ville ikke vises i Joly. I Joly ville de fortsatt stå på den gamle gruppen.

Figur 5.7 viser til hva brukeren vil se ved å klikke seg inn på siden. Her er det lagt til muligheten for enkelt å kunne se hvem som er på hvilken gruppe hvis det er tvil eller det er usikkerhet om studenten er på gruppen eller ikke. Det andre som er mulig å gjøre er å endre hvilken gruppe studenten går på. I begge tilfellene har brukerne frihet til å velge hvilket som helst kurs, selv om det i de fleste tilfeller vil bare være aktuelt å velge kurset for gjeldende semester.



Figur 5.7: *Hovedsiden for bytting av gruppe*

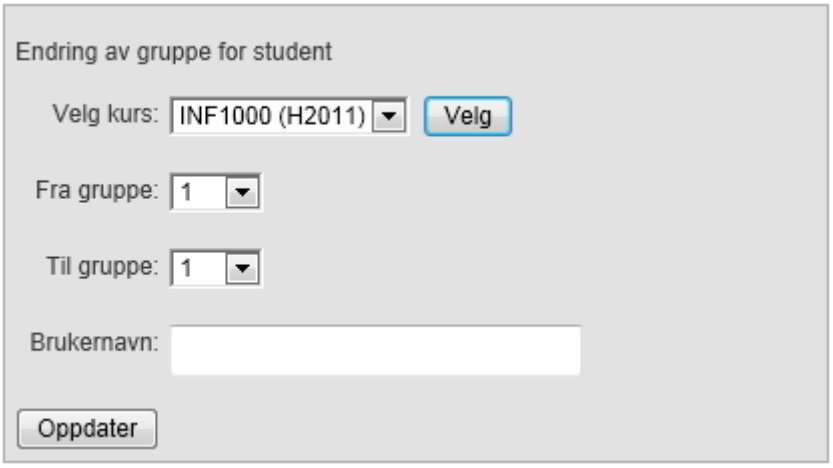
Figur 5.8 viser listen over studenter på gruppen som blir hentet ut ved valg av kurs og gruppe. Denne funksjonaliteten var ikke noe som var nødvendig å ha der, men den ble implementert ettersom det er en enkel egenskap å implementere. Den kan gjøre det enklere hvis det er ønskelig å kunne raskt sjekke at studenten er på den gruppen, i tilfellet brukeren har mottatt feilmelding ved forsøk på bytting av grupper. En annen måte å finne ut om studenten er på gruppen er å benytte seg av *oversikten over innleverte obliger* tidligere beskrevet i Seksjon 5.2, men at det er upraktisk å måtte navigere seg bort fra siden hvor selve funksjonaliten man ønsker å benytte er.

Ved valg av kurs vil vi få slik det er vist på Figur 5.9, hvor valgene for hvil-

Figur 5.8: *Liste over studenter på gruppe*

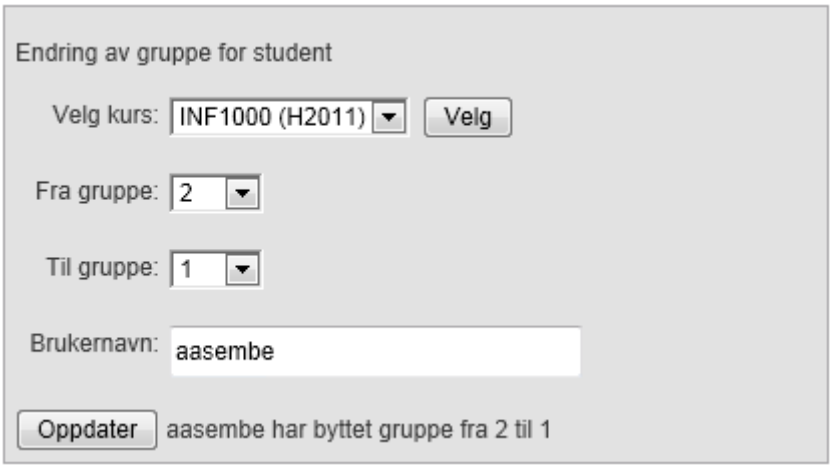
ken gruppe studenten er på og hvilken gruppe studenten skal overføres til bli tilgjengelig, samt tekstfelt for brukernavnet til studenten. Her velges så gruppen studenten allerede går på i *Fra gruppe*, så gruppen studenten skal overføres til *Til gruppe* og dermed inntasting av brukernavnet til studenten. Ved å trykke på oppdater-knappen vil studenten bli overført, hvis informasjonen om studenten stemmer i forhold til brukernavn og hvilken gruppe studenten går på. Korrekt overføring vil resultere i en tilbakemelding om at studenten har byttet gruppe slik det er vist på Figur 5.10.

Ved feil inntastet brukernavn eller valg av gruppe vil ingenting skje. Men for



Figur 5.9: *Meny for bytting av gruppe*

å kunne gi brukeren av Joly en anelse om hva som er feil returneres det en feilmelding. Denne feilmeldingen vil ikke gjøre forskjell på om studenten eksisterer i databasen eller om studenten ikke er på gruppen. Så hvis studenten ikke eksisterer i det hele tatt i databasen vil det fortsatt kun vises en feilmelding om at studenten ikke er på den og den gruppen. Tidligere funksjonalitet som det er vist til i Seksjon 5.3 om *godkjenning av eldre obliger* er det mulig å gjøre et enkelt søk hvis det er nødvendig å finne ut av om studenten er i databasen eller ikke.



Figur 5.10: *Suksessfull endring av gruppe for student*

Det er mulig å velge tidligere kurs av to årsaker; Det var lettere implementa-

sjonsmessig og at det er ikke nødvendig med slik restriksjon ettersom det vil ha ingen effekt hvis noen skulle endre grupper på tidligere studenter og at det er forbehold om at gruppelærere for kurset vet hva de har lov til. Men vi skal nok ikke se bort ifra at det ville vært en bedre løsning å kun la de gjøre endringer på kurs de er gruppelærere for, mens f.eks. foreleser har alle muligheten til å velge alle kurs.

5.6 Øvrige rettinger og nødvendige endringer

Under denne seksjonen skal vi se på diverse problemer som oppstod under utvikling og testing av Joly. Det ble utført kontinuerlig testingen av Joly for å forsikre oss om at det fungerte slik det skulle til semesterstart høsten 2012, hvertfall i så stor grad som overhodet mulig. De neste seksjonene skal derfor beskrive noen av de større feilene som ble funnet og endringer som måtte gjøres. Vi gjør oppmerksom på at noen endringer som ble gjort er ikke optimale, og alle disse det gjelder vil bli forklart nedenfor.

5.6.1 Oversikt over gruppe

Av tidligere funksjonalitet er det mulig i Joly å få en liste over innleverte obliger av studenter, der er det mulig å se likhetstatus i henhold til andre innleverte obliger. Det oppstod en feilmeldingen når brukeren valgte gruppe, som hadde med en HQL-spørring å gjøre. Den skulle hente ut de siste innleverte obligene fra hver student på gruppen. For å finne feilen ble det gjort flere forsøk på å forstå denne overkompliserte HQL-spørringen som er lagt ved i Listing 5.6, men det ble tydelig at dette kunne ta lang tid å rette opp.

Listing 5.6: Overkomplisert HQL spørring

```
1 "SELECT new Studentsolution(MAX(s.studentsolutionID) as studentsolutionID, s
      .assignment, s.student, s.deliverytime, s.text)\n"
2     + "FROM   Studentsolution s\n"
3     + "WHERE  s.assignment.assignmentID=?\n"
4     + "AND s.student.username IN(SELECT s.username FROM Student s WHERE s.
          coursegroups.coursegroupID=?)"
5     + "AND s.studentsolutionID = (SELECT MAX(st.studentsolutionID) FROM
          Studentsolution st WHERE st.student.username=s.student.username and
          st.assignment.assignmentID=?)"
6     + "GROUP BY s.studentsolutionID, s.assignment.assignmentID, s.student.
          username, s.deliverytime, s.text order by s.student.username";
```

Løsningen ble derfor å unngå å bruke denne kompliserte spørringen og heller dele problemet opp i flere deler. Så da ble den reimplementert med javakode, noe som tidsmessig tok mindre tid enn tiden som ble brukt på å forsøke og forstå HQL-spørringen. Det oppstod et mindre problem som var feil casting fra long til int. Dette var mer overraskende for det betyr at i løpet av årene som har gått har det tydeligvis skjedd en endring i hvordan det er lov å caste long til int, har derfor inkludert endringen som vises i henhold til Listing 5.7.

Listing 5.7: Retting av casting fra Long til int

```
1 //Produserte casting feil:
2 ss.setReportcaseCount(((Integer)(o[2])).intValue());
3
4 //Rettet opp til:
5 ss.setReportcaseCount((int) ((long)o[2]));
```

For å komme tilbake til dette med HQL-spørringen stiller vi oss usikker til hvorfor denne feilen oppstod nå. Av det som er kjent har det ikke vært noen klager på dette tidligere så spørsmålet er om dette kan ha noe med de oppdaterte rammeverkene å gjøre. Hibernate (hvor HQL er fra) var tross alt veldig utdatert før oppdatering slik det ble påvist i Seksjon 4.2.3 om oppdateringsplanen. Allikevel skal vi ikke utelukke muligheten for at det kan ha blitt utført noen feil i tidligere seksjoner, hvor endringer var nødvendig for at Joly skulle kjøre med oppdaterte rammeverk. Om det ikke er det siste er det interessant å se hvordan disse åpen kildekode rammeverkene kan endre seg over tid. Er man ikke er oppmerksom på disse endringene vil man kunne sitte igjen med en applikasjon som ikke fungere helt som den skal.

5.6.2 Oppdatering av ansatt

Det ble oppdaget feil ved oppdatering av ansatt. Ut ifra brukergrensesnittet som blir presentert skulle det tilsi at det var mulig å endre privilegium og fjerne ansatt. Ingen av disse fungerte som det skal. Ved valg av privilegium så ble det benyttet en valg-meny hvor det var mulig å velge mellom brukergruppene; gruppelærer, foreleser og administrator. Ved en innsending benytter Joly seg av en *egenskaps-editor* som brukes til å få tak i ansattobjekt slik at det kan oppdateres med nye verdier som måtte være i skjemaet. Problemet var at dette gjorde den ikke riktig,

parameteren som ble mottatt i denne *egenskapseditoren* var brukernavnet og var av typen String. Slik det var programmert forventet den EmployeeID av typen int og ikke brukernavn av typen String, noe vi kan se ut ifra Listing 5.8. Løsningen var at i stedet for å kalle på metoden for å få tak i ansattobjektet med EmployeeID som argument, ble det benyttet en annen metode som tok brukernavnet som argument i stedet slik det kommer fram av Listing 5.9.

Listing 5.8: Uthenting av ansatt med Id

```
1 public void setAsText(String text) throws NumberFormatException {
2   int id = Integer.parseInt(text);
3   Employee employee = dao.findEmployeeByID(id);
4   setValue(employee);
5 }
```

Listing 5.9: Uthenting av ansatt med brukernavn

```
1 public void setAsText(String text) throws NumberFormatException {
2   Employee employee = dao.findEmployee(text);
3   setValue(employee);
4 }
```

Det å finne denne feilen var vanskelig, siden det oppstod ingen synlige feilmeldinger eller lignende som resultat av kjøring. Allikevel var mulig å trekke konklusjonen om at funksjonaliteten ikke fungerte som den skulle ettersom det hverken var mulig å oppdatere privilegium eller fjerne ansatt. Til slutt ble det funnet en feilmelding som var en variabel som ble videresendt i metodekall slik at det mest sannsynlig er mulig å teste på dette. Det var dermed mulig å printe ut denne feilmeldingen slik at det var mulig å finne ut hva som var feil. På en måte er det bra at en applikasjon ikke kræsjer eller viser tegn til at noe har gått galt, men å forsøke å debugge noe sånt med skjulte feilmeldinger gjør saken mye vanskeligere enn det trenger å være.

5.6.3 Retting tidligere medførte følgefeil

Under utvikling av ny funksjonalitet ble det oppdaget at det var problemer med å legge til studenter. Det ble stilt tvil rundt om denne feilen kunne ha eksistert tidligere, men at det var noe tvilsomt, så det betydde at det utelukkende var nye endringer som kunne ha forsårsaket dette. Det sto mellom endringer i oppdaterte rammeverk som kunne ha betydning eller at tidligere rettinger kunne ha skylden.

Her viste det seg å være det siste alternativet, hvor endringer i annotasjoner i mappingfilene skapte problemer. De fleste databasetabeller som Joly benytter seg har et Id-felt, og i forbindelse med dette så er det nødvendig å fortelle Hibernate at ved oppretting av ny rad skal Id-feltet automatisk økes til det neste. Dette var ikke tilfellet for den ene tabellen *Student* som ikke inneholder noe Id-felt, men kun et brukernavn. Den hadde blitt programmert slik at den trodde at den hadde et Id-felt som det er mulig å se ut ifra Listing 5.10 hvor annotasjonen *GeneratedValue* betyr at ved en ny rad så vil dette feltet økes som et Id-felt. Etter at årsaken ble funnet ble det rettet opp som det er vist i Listing 5.11 til at den nå ikke ser på kolonnen username som et Id-felt.

Listing 5.10: Implementasjon før endring

```
1 @Id
2 @GeneratedValue
3 @Column(name="username", nullable=false, length=15)
4 public String getUsername() {
5   return username;
6 }
```

Listing 5.11: Implementasjon etter endring

```
1 @Id
2 @Column(name="username", nullable=false, length=15)
3 public String getUsername() {
4   return username;
5 }
```

5.6.4 Mail

Det var usikkerhet rundt hvorvidt mail-funksjonaliteten fungerte som den skulle. I tillegg til dette kom det ønske fra årets gruppelærere om å kunne inkludere mailadressen til studenten i mailen som blir sendt til gruppelærer når studenten leverer sin besvarelse. I dette tilfellet viste det seg at selve mail implementasjonen hadde kun blitt kommentert ut og erstattet med en dummy implementasjon som mest sannsynlig brukes under utvikling. Endring av dette var relativt enkelt siden Joly benytter seg av beans fra Spring, så det var kun å bytte til ønskelig implementasjon og rekompilere. Videre var det arbeidet med å legge til mailadressen til studentene i mailen som blir generert ved innlevering. Her var det snakk

om minimale endringer i form av en ekstra linje lagt til i mailen til gruppelæreren. Noe viktig å merke seg var at siden INF1000 ikke kun er forbeholdt ifi-studenter var det nødvendig å ta i bruk mailadresser som fungerer på et globalt nivå på universitetet. Hvor vanligvis for ifi-studenter brukes *brukernavn@ifi.uio.no* måtte det benyttes *brukernavn@ulrik.uio.no* hvor den sistnevnte vil rute mailen til riktig mailadresse.

5.6.5 Nødvendig med flere filer ved innlevering

Slik innleveringssystemet Joly fungerer er at ved innlevering av en oblig legges løsningen fra studenten ved først, så med muligheten for å legge ved øvrige filer. Disse øvrige filene kan f.eks. være en tekstfil som programmet leser eller informasjon til gruppelæreren. Det er kun den første filen som legges til som vil bli registrert i databasen og som er den innleverte oppgaven. Gruppelærere vil bli tilsendt en mail ved innlevering fra en student inneholdende selve løsningen og alle ekstra filer lagt til i øvrige-filfeltet som vedlegg. Så når det ble opplyst at første obligatoriske oppgave høsten 2012 i INF1000 skulle ha studentene til å løse to oppgaver som skulle være i hver sin fil var det nødvendig å utføre noen endringer. Siden denne informasjon kom veldig nærme første obligatoriske oppgave krevde det en rask løsning. Noe som resulterte i at Joly vil konkatenere den første filen levert og den andre filen levert i øvrige-filfeltet, som da vil bli lagt til i databasen.

Dette vil gjøre øvrige-filfeltet ubrukelig når det ikke leveres mer enn en fil, samt at det vil heller ikke fungere som et øvrige-filfelt. Tvert imot vil det kunne skade selve fuskesjekken Joly bruker siden hvis en student velger å legge ved en fil i øvrige-filfeltet vil denne konkateneres med selve løsningen. Med mer tid ville en bedre løsning være å kunne ha muligheten til å legge til enten en java-fil eller zip-fil i det første feltet. Slik at hvis det var kun en java-fil gjorde den som vanlig, men hvis det var en zip-fil ville den åpne denne og konkatenere alle filene inneholdt i zip-filen for å så lagre dette i databasen.

5.6.6 Tall i brukernavn til studenter

Ved semesterstart viste det seg at det var en student med tall i brukernavnet, noe som viste seg å være et problem ettersom Joly tydeligvis ikke aksepterte tall i brukernavn ved registrering. Dette var forårsaket av tester som sjekket at alle tegnene var innenfor spekteret a til z ved hjelp av regexp mønster. Mest sann-

synlig så kan dette ha vært en måte å unngå at det ble forsøkt registrering av studenter med spesialtegn, selv om det virker litt rart og siden brukernavn ikke vanligvis inneholder dette. Dette ble løst ved å fjerne disse testene for å se om bokstavene var mellom a til z. Legge til student siden måtte endres og den siden hvor den tar en fil med studenter. I tillegg måtte det gjøres en endring ved innlevering av obliger ettersom den sjekket at brukernavnet var riktig skrevet i forhold til reglene gitt som var da a til z.

Med mer tid ville det nok kunne være bedre å beholde en form for sjekk slik at det unngås spesialtegn og brukernavn med store bokstaver hvor disse mest sannsynlig vil opptre som separate brukernavn. Så det er viktig å holde styr på dette når det skal legges inn studenter i fremtiden inntil en ordentlig sjekk er implementert.

5.6.7 Visuelle og øvrige endringer

Det ble også gjort små endringer her og der, disse vil ikke bli dekket i stor detalj, men bare nevnt kort. Det ble gjort en del visuelle endringer i tidligere eksisterende JSPer, dette inkluderte som f.eks tekst på knapper og posisjon av elementer som knapper og option-felt. Gruppelærerne fikk nye restriksjoner på rettighetene hvor de nå ikke har mulighet til å opprette kurs eller legge til studenter i batch.

5.7 Tidligere forsøk på intergrering i kompilert kode

Ved våren 2012 hadde vi enda ikke kildekoden til Joly, og i denne forbindelsen vurderte vi mulighetene for å implementere ny funksjonalitet i den kompilerte versjonen av Joly som vi hadde tilgang til. Denne kompilerte utgaven av Joly var versjon 1.5, men før vi kom til stadiet som omhandlet oppdatering av rammeverk så fikk vi tilgang til kildekoden til Joly etter arbeid av Siamek Darisiro[5]. Denne kildekoden var resultatet av dekompilerte filer og gikk derfor under versjonsnummeret 1.6. Det ble allikevel utført noen enkle tester for å se hvorvidt det var overhodet mulig å videreutvikle Joly med kun kompilert kildekode. Dette innebærte et forsøk hvor det ble lagt til nye filer og at noen filer ble erstattet med en oppdatert versjon. Blant disse filene var det endringer i konfigurasjonsfilen, ny kontrollerklasse, ny JSP og noen andre små endringer. Av disse var det kun

kontrollerklassen som måtte kompileres på forhånd før den ble lagt til i filstrukturen til Joly. Det viste seg å være mulig å legge til ny kontrollerklasse på denne måten, men dette var kun en veldig enkel klasse og derfor var det uvisst hvordan det ville fungere på mer kompliserte klasser.

I etterkant av endringene som ble gjort med oppdatering av rammeverkene og de nødvendige rettingene der, ser vi at det hadde blitt en mye større jobb å videreutvikle Joly med kun kompilerte kildekode. Men ettersom vi fikk tak i kildekoden før arbeidet med å oppdatere rammeverkene ble satt i verk, så ble det ikke utforsket noe videre med endringer til kompilert kildekode.

5.8 Oppsummering

Vi har sett på hvordan det er å komme med et tillegg til Joly, ved å videreutvikle med bruk av annoterte kontrollerklasser selv om eksisterende deler av Joly ikke benytter seg av dette. Så Joly har nå fått implementert fire nye funksjonaliteter, samt rettinger av en rekke feil og små visuelle oppussinger og en endring i databasen. Dette betyr at Joly kan nå benyttes som godkjentlistesystem, hvor gruppelærerne har oversikt over alle innleveringene til studentene og kan endre status på dem. Vi fikk også erfare at det kan forekomme endringer i design for funksjonaliteter under implementasjonsdelen. Samtidig fikk vi oppleve hvordan det er å måtte rette opp feil i en applikasjon som var satt i drift og hvordan dette ble løst. Dermed skal vi gå over til siste kapittel hvor vi ser på hvilke konklusjoner det er mulig å trekke på bakgrunn av arbeidet som er utført med Joly.

Kapittel 6

Konklusjoner og videre arbeid

Vi skal i dette avsluttende kapitlet se på hva vi har oppnådd gjennom denne oppgaven og hvilke slutninger vi kan trekke ut ifra arbeidet og problematikken vi møtte underveis. Joly kan nå benyttes som et godkjentlistesystem og ble derfor satt i drift høsten 2012. For at dette skulle la seg gjøre fikk Joly oppdatert rammeverk og kildekode, og implementert de nødvendige funksjonalitetene. Gruppelærere kan godkjenne obliger fra studenter og det kan genereres en godkjentliste over studenter som har bestått visse obligatoriske oppgaver.

Vi innså at vi måtte prioritere visse planlagte funksjonaliteter. Dette medførte at oversikt av status for studenter måtte vike p.g.a. av at Joly allerede gir en god nok beskjed om når en innlevering er blitt gjort; ettersom det ved innlevering gis en kvittering om suksessfull innlevering. Studentene får også vite når gruppelærerne har rettet obligen via mail med informasjon om de har fått godkjent, nytt forsøk eller ikke godkjent. Det var også planlagt en rekke nye funksjonaliteter som var tiltenkt foreleser som var statistikkbaserte rapporter, men som det heller ikke var tid til. Men dette er det fullt mulig å utvide i fremtiden ettersom det vil være endringen av Joly over til et godkjentlistesystem som vil gjøre det mulig å trekke ut statistikk etter endt semester.

6.1 Konklusjon

Det viste seg å være en større prosess enn antatt med å implementere ny funksjonalitet i Joly med tanke på at det ikke har vært noe vedlikehold de siste to

årene. I første omgang var det nødvendig å oppdatere eldre rammeverk fordi det hadde gått relativt lang tid siden dette sist ble gjort. Grunnen til at det var ønskelig å oppdatere var fordi når det kommer nye versjoner av rammeverkene er dette for å adressere problem ved rammeverket som kan være bl.a. rettinger av funksjonalitet og sikkerhetshull for å nevne noen. Det er viktig å presisere at selv om rammeverkene har blitt oppdaterte, så betyr ikke det nødvendigvis at Joly tar i bruk denne funksjonaliteten. Ved endring av rammeverkene medførte det at det var nødvendig å rette eksisterende kildekode i Joly siden det tydeligvis hadde skjedd endringer i hvordan ting blir implementert. For å kunne gå videre med implementasjonen av ny funksjonalitet var det nødvendig å lære seg hvordan rammeverkene skulle brukes, hvertfall de mest nødvendige. Læringskurven for å lære seg å bruke disse rammeverkene var bratt og medførte nok til dårligere implementert kildekode enn ønsket.

Dermed står vi mellom to mulige årsaker til at dette var vanskeligere enn antatt. Først, at det har vært lite vedlikehold tidligere som kan ha forårsaket at vedlikeholdet nå tok lang tid. Det andre er endringer i rammeverkene som har foregått over tid. Her vil jeg påstå endringer til disse åpen kildekode rammeverkene vil gjøre det vanskeligere å vedlikeholde en applikasjon om man ønsker å oppdatere rammeverkene ofte. Dette så vi tidligere ved oppdatering av rammeverk i oppgaven, hvor endringene medførte at vi måtte endre kildekoden til Joly til ny syntaks.

Det var et ønske om å gå over innloggingssystemet Weblogin for bedre sikkerhet slik at alle kunne logge inn på Joly inkludert studenter. Det ble derfor brukt en god del tid tidligere i prosjektet på å se om dette var mulig og hvordan det skulle gjøres. USIT benyttet seg tidligere av en gammel måte som var usikker og utsatt for “man-in-the-middle attack” og gikk derfor over til en ny måte som var Weblogin. Det ble brukt noen uker på å se hvorvidt det var mulig å få Joly til å bruke Weblogin, men det kom ikke så mye ut av det siden USIT var i en endringsfase og det var lite veiledning å få på dette området siden programmeringsspråket vi benyttet var java. Dermed var det ikke hensiktsmessig å gå noe videre med dette ettersom det ville ta altfor lang tid med tanke på at det var annen viktig funksjonalitetet som måtte prioriteres først.

Dermed vil jeg avsluttende komme innpå spørsmålet om det er verdt å bruke

disse åpen kildekode rammeverkene med tanke på det som vi tidligere har sett i oppgaven. På dette punktet er jeg tildels splittet, men at med tanke på hvor lite vedlikehold som tidligere hadde blitt gjort med Joly for å så oppdatere de fleste rammeverkene vil jeg si at vi kom godt ut av det. Dette er kun i lys av rettinger som var nødvendige for at Joly skulle kunne kjøre og være operativt. Skal disse type rammeverkene tas i bruk så burde det utføres regelmessig vedlikehold for å unngå at det må gjøres flere større rettinger samtidig. For hvis ikke kan det være vanskelig å kartlegge hvilke endringer som blir gjort i forskjellige rammeverk, som kan medføre at det kan bli vanskelig å vite hva som må rettes og hvordan det skal gjøres.

6.2 Erfaringer

Det å implementere ny funksjonalitet viste seg å være lettere etter hvert, men at det å lære seg rammeverk og virkemåten deres har en bratt læringskurve. Dette gjelder spesielt når det er ønskelig å gjøre noe litt utenom det de enkleste eksemplene i dokumentasjonen til disse rammeverkene viser til. Dette er nok det største problemet jeg har med disse åpen kildekode rammeverkene så langt. Er man uerfaren og skal forsøke å gjøre noe litt mer avansert enn de enkleste eksemplene som de har illustrert i dokumentasjonen så krever det en del forståelse og bearbeiding av dokumentasjonen. Dette har nok medført at ikke alle løsninger implementert er gjort etter beste praksis, men at funksjonaliteten fortsatt er til stede. For å kunne utføre endringer i kildekoden til Joly var det nødvendig å forstå rammeverkene og samspillet mellom de, noe som var vanskelig.

Ved å videreutvikle et system som Joly som skulle tas i bruk relativt tidlig i forhold til når jeg startet med implementeringen var tøft. Ved semesterstart sto jeg som både videreutvikler av de nye funksjonalitetene og som systemansvarlig for Joly. Hvis det oppstod noen feil med Joly ble jeg kontaktet slik at det var mulig å finne årsaken for så å rette det opp. Dette var tilfellet ved obligatorisk oppgave 1 for høsten 2012. Hvor det oppstod problemer som krevde en løsning snarest, noe som ble ordnet opp og ga oss et operativt system.

Under utvikling av den nye funksjonaliteten fikk jeg erfare hvor nødvendig det var å iterere og evaluere nåværende implementasjon, for å så komme med nye endringer om nødvendig. Dette gjaldt endringer i selve brukergrensesnittet slik

at det var mer intuitivt. Med tanke på at Joly skulle tas i bruk høsten 2012 ble det fokusert mye på å få til at all funksjonalitet fungerte som den skulle innenfor tidsplanen. Dette gikk på bekostning av kildekoden som kunne vært noe bedre, men at filstrukturen på selve prosjektet er bevart.

6.3 Videre arbeid

Gjennom denne er oppgaven er vi blitt kjent med innleveringssystemet Joly, og det ble planlagt tilleggsfunksjonalitet som det tidsmessig ikke var mulig å fullføre. Dette gjelder en del rapporter spesifisert i Kapittel 3, men som viste seg å ikke være nødvendige med det første og ble derfor lavere prioritert. I tillegg ble det ingen tid til å se mye på Weblogin for innloggingsmuligheter av studenter eller HTTPS slik at forbindelsen mellom bruker og applikasjoner krypteres. Jeg vil påpeke at etter å ha jobbet med Joly vil den eneste grunnen til å implementere bruk av Weblogin være for å gi studentene den ene planlagte rapporten nevnt tidligere. Selve arbeidet som da må legges inn i dette, er nok ikke verdt det.

Videre var det arbeidet med oppdatering av rammeverk som ble dekket i Kapittel 4. Her fikk vi ikke mulighet til å oppdatere alt, så det som gjenstår er å oppdatere Jetty ettersom denne er veldig utdatert, noe som gjør det mer vanskelig. I tillegg til oppdatering av rammeverkene ble det ikke oppdatert tidligere kode som benytter seg av utdaterte superklasser, noe som gjør det aktuelt å reimplementere disse ved hjelp av annoterte kontrollere. Det gjenstår også migrering fra Hibernate 3 til Hibernate 4.

Så er det avsluttende videre arbeid som kommer fram av Kapittel 5. Her er det mer små endringer som inkluderer feil tegnsett som gjør at de norske bokstavene æ, ø og å opptrer som spørsmålstegn i innleveringer som hentes fra databasen. Det ble tidligere gjort endringer til øvrige-filfelt og mulighet for tall i brukernavn, disse implementasjonene var gjort så raskt som mulig og burde reimplementeres skikkelig for robusthet.

Bibliografi

[1] Emmanuel Bernard. *Hibernate Annotations, Reference Guide*. Online. 2010. URL: http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/.

[2] John Blackmore. *GeneratorType*. Online. 2006. URL: https://community.jboss.org/wiki/GeneratorType.

[3] Joseph Caristi. *Jetty 6 Architecture*. Online. 2009. URL: http://docs.codehaus.org/display/JETTY/Architecture.

[4] OW2 Consortium. *ASM*. Online. 2012. URL: http://asm.ow2.org/.

[5] Siamek Darisiro. "The Recovery of an Old Lost System". Masteropp. Universitet i Oslo, 2012.

[6] DocForge. *Framework*. Online. 2011. URL: http://docforge.com/wiki/Framework.

[7] Marty Hall og Larry Brown. "Core Servlets and JavaServer Pages". I: Prentice Hall PTR, 2004.

[8] JBoss. *About Hibernate*. Online. 29. okt. 2012. URL: http://www.hibernate.org/about.html.

[9] JBoss. *What is Object/Relational Mapping?* Online. 29. okt. 2012. URL: http://www.hibernate.org/about/orm.html.

[10] Rod Johnson, Juergen Hoeller og Keith Donald. *Introduction to Spring Framework JDBC*. Online. 2012. URL: http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/jdbc.html#jdbc-introduction.

[11] Rod Johnson, Juergen Hoeller og Keith Donald. *Spring Framework Reference Documentation*. Online. 2012. URL: http://static.springsource.org/spring/docs/current/spring-framework-reference/html/overview.html.

[12] Christian Kringstad Kielland. "Metoder for likhetsvurdering av innleverte obligatoriske oppgaver i java". Masteropp. Universitet i Oslo, 2006.

[13] Gavin King bl. a. *Hibernate Reference Documentation*. Online. 2004. URL: http://docs.jboss.org/hibernate/stable/core/manual/en-US/html/architecture.html.

[14] Meyyappan Muthuraman. *Spring MVC*. Online. 2009. URL: http://www.vaannila.com/spring/spring-mvc-tutorial-1.html.

[15] Oracle. *Java API, Interface Servlet*. Online. 2003. URL: http://docs.oracle.com/javaee/1.4/api/javax/servlet/Servlet.html.

[16] Universitetet i Oslo. *weblogin*. Online. 2. mar. 2012. URL: http://www.uio.no/tjenester/it/brukernavn-passord/weblogin/mer-om/.

[17] George Reese. *JNDI and the new JDBC 2.0 Standard Extension*. Online. 1999. URL: http://www.developer.com/java/data/article.php/610901/JNDI-and-the-new-JDBC-20-Standard-Extension.htm.

[18] Springsource. *Spring Framework*. Online. 2012. URL: http://www.springsource.org/spring-framework.

[19] Springsource. *Spring, Javadoc-api 2.5*. Online. 2004. URL: http://static.springsource.org/spring/docs/2.5.x/api/index.html.

[20] Springsource. *Spring, Javadoc-api 3.1*. Online. 2004. URL: http://static.springsource.org/spring/docs/3.1.x/javadoc-api/.

[21] Therese Steensen og Hanne Vibekk. "DHIS and Joly: two distributed systems under development: design and technology". Masteropp. Universitet i Oslo, 2006.

[22] Team Lead Stefan Hinz bl. a. *What is MySQL?* Online. 2012. URL: http://dev.mysql.com/doc/refman/5.1/en/what-is-mysql.html.

[23] Wikipedia. *HTTP Secure*. Online. 29. okt. 2012. URL: http://en.wikipedia.org/wiki/HTTP_Secure.

[24] Wikipedia. *Inversion of Control*. Online. 29. okt. 2012. URL: http://en.wikipedia.org/wiki/Inversion_of_control.

[25] Wikipedia. *Model-view-controller*. Online. 29. okt. 2012. URL: http://en.wikipedia.org/wiki/Model-view-controller.

[26] Wikipedia. *Multitier architecture*. Online. 29. okt. 2012. URL: http://en.wikipedia.org/wiki/Multitier_architecture.

[27] Wikipedia. *Open-source software*. Online. 1. nov. 2012. URL: http://en.wikipedia.org/wiki/Open-source_software.

[28] Wikipedia. *Spring Security*. Online. 20. mar. 2012. URL: http://en.wikipedia.org/wiki/Spring_Security.

Tillegg A

A - Kodesnutter

Listing A.1: Joly 1.6 Entity annotasjon

```
1 @Entity(access = AccessType.PROPERTY)
2 @Table(name="Course", uniqueConstraints
3     = {@UniqueConstraint(columnNames={"semester", "courseCode"})})
4 public class Course {
5  ...
6 }
```

Listing A.2: Joly 1.7 Entity annotasjon

```
1 @Entity
2 @Access(AccessType.PROPERTY)
3 @Table(name="Course", uniqueConstraints
4     = {@UniqueConstraint(columnNames={"semester", "courseCode"})})
5 public class Course {
6  ...
7 }
```

Listing A.3: Joly 1.6 Id annotasjon

```
1 @Id (generate = GeneratorType.AUTO)
2 @Column(name="courseID", nullable=false, updatable=false)
3 public int getCourseID() {
4   return courseID;
5 }
```

Listing A.4: Joly 1.7 Id annotasjon

```
@Id
@GeneratedValue
@Column(name="courseID", nullable=false, updatable=false)
public int getCourseID() {
  return courseID;
}
```

Listing A.5: Joly 1.6 JoinTable annotasjon

```
@ManyToMany(targetEntity=Employee.class, fetch=FetchType.EAGER)
@JoinTable(table=@Table(name="Employee_Course"),
  joinColumns={@JoinColumn(name="courseID")},
  inverseJoinColumns={@JoinColumn(name="username")})
public Collection<Employee> getEmployees(){
  return employees;
}
```

Listing A.6: Joly 1.7 JoinTable annotasjon

```
@ManyToMany(targetEntity=Employee.class, fetch=FetchType.EAGER)
@JoinTable(name="Employee_Course",
  joinColumns={@JoinColumn(name="courseID")},
  inverseJoinColumns={@JoinColumn(name="username")})
public Collection<Employee> getEmployees(){
  return employees;
}
```

Listing A.7: Joly 1.6 TemporalType annotasjon

```
@Basic(temporalType = TemporalType.TIMESTAMP)
@Column(name="deliverytime", nullable=false)
public Timestamp getDeliverytime() {
  return deliverytime;
}
```

Listing A.8: Joly 1.7 TemporalType annotasjon

```
@Temporal(TemporalType.TIMESTAMP)
@Column(name="deliverytime", nullable=false)
public Timestamp getDeliverytime() {
  return deliverytime;
}
```

Listing A.9: LoginInterceptor.java

```
1 public class LoginInterceptor extends HandlerInterceptorAdapter{
2   public boolean preHandle(HttpServletRequest request, HttpServletResponse
        response, Object handler) throws Exception {
3
4     UserSession userSession = (UserSession) WebUtils.getSessionAttribute(
          request, "userSession");
5     if (userSession == null) {
6       response.sendRedirect("/loginform.html");
7       return false;
8     } else {
9       return true;
10    }
11  }
12 }
```

Tillegg B

B - User guide

Her er det inkludert en user guide for å hente kildekoden. For å kompilere, sette opp Joly og endre databaseoppsett ligger det en utvidet user guide sammen med Joly 1.7 på SVN. Grunnen til at det er delt opp er av sikkerhetsmessige årsaker.

B.1 Hente kildekode

All kildekoden til Joly finner man på adressen vist i Listing B.1, herfra er det mulig å velge hvilken versjon av kildekode som er ønskelig å laste ned. I skrivende stund er Joly 1.7 nyeste og fungerende versjon. I Joly 1.7 ligger har all kildekode, rammeverk og Jetty. Det er viktig å merke seg at i jetty/webapps mappen ligger det allerede en ferdig kompilert versjon av Joly klar for å kjøres, men denne er nok ikke siste versjon av kildekoden. Så derfor er det viktig å kompilere før man skal sette opp Joly. Det er kun mulig å få lastet ned fra repository om man er bli gitt tilgang til dette av eieren av repository.

Listing B.1: SVN repository

```
1 https://svn.ifi.uio.no/repos/users/arnem-Joly2011/arnem-Joly2011/
```

For å laste ned Joly 1.7 benyttes kommandoen i Listing B.2.

Listing B.2: Kommando for å laste ned Joly1.7

```
1 svn checkout https://svn.ifi.uio.no/repos/users/arnem-Joly2011/arnem-
      Joly2011/trunk/Joly1.7/
```