RAPPORT

-

Applikasjonens tekniske sider



Gruppemedlemmer:

Ø yvind Gautestad

G jermund Vedvik

L inda Fermann

A ndrea Abrahamsen

A nders Borg Larsen

Fag:

IS-OBJ2000 Objektorientert programmering, Høgskolen i Vestfold, Høst 2013

**Innholdsfortegnelse**

Innledning 3

Forutsetninger 3

Applikasjonsarkitektur 4

Hvordan har vi implementert dette i vår løsning 4

Sammenhengen mellom de forskjellige elementene 5

Lyttere til brukerkomponenter 6

Erfaringer med implementeringen 6

Chain of Responsibility 7

Eksempel på implementasjon - ZipFileData 7

Erfaringer med implementeringen 9

XMLDecoder 10

Hvordan vi har implementert dette i vår løsning 10

En beskrivelse av våre erfaringer med implementeringen. 10

Strømming av data og zipteknologi 11

Hvordan vi har implementert dette i vår løsning 11

Introduksjon 11

Lese data fra URL 12

Pakke ut innhold i zip 13

Lagre zipfil til disk 16

Lagre enkeltfiler til disk 16

Erfaringer med implemementering 17

Swingworker og tråder 17

Brukergrensesnitt 18

JTree 18

Generelt om JTree 18

Hvordan vi har implementert det i vår løsning 19

Erfaringer med implementering av JTree 19

Dialog med brukeren 19

Save 19

Erfaringer med impelmentering av Save 19

Avslutning 20

Litteraturliste 21

# Innledning

Denne rapporten skal gi et innblikk i de tekniske aspekter ved vår applikasjon. De tekniske aspekter, og særlig detaljer, er i sin største forstand uendelig mange.

Vi har ved utarbeidelse av denne dokumentasjonen forsøk å være både restriktive og utfyllende. Det er i seg selv en selvmotsigelse, men en balansegang her er nok nødvendig, da vi skal vise at vi har skjønt de tekniske sider i vår løsning uten at vi forklarer alle små detaljer. Den vil derfor ikke være fullstendig utfyllende for alle aspekter, men ta for seg de områder som vi mener har stor relevans når det kommer til valg av tekniske løsninger og beskrivelse av teknologier vi har benyttet.

Vi starter med å beskrive applikasjonsarkitekturen vi har benyttet og hvordan vi har implementert et mønster som kalles Chain-of-Responsibility.

Videre tar vi for oss XMLDecoder og strømmeteknologi/zipteknologi. Det sistnevnte har vi gått litt i dybden av, da det er helt sentralt i store deler av applikasjonen.

Deretter beskriver vi bruk av en klasse som heter SwingWorker hvor vi benytter tråder, og til slutt noen tekniske aspekter vedrørende brukergrensesnittet i applikasjonen.

# Forutsetninger

Oppgaven gir oss ingen opplysninger om hva som befinner seg på serversiden, blandt annet validering av brukernavn og passord. Det er derfor ikke lagt inn noe form for validering på karakterer som er brukt i innloggingsfeltene. Dette kunne enkelt blitt gjort ved å endre på modellen til plaindocument i disse feltene.

# Applikasjonsarkitektur

I applikasjoner, slik som den vi har laget i denne hjemmeeksamen, så er det behov for å skille mellom elementer som nødvendigvis ikke har noe med hverandre å gjøre. Applikasjonen i seg selv, gjennom sin kompleksitet, er av en slik størrelse at det nødvendiggjør en splittelse av de ulike elementer. Dette både for å opprettholde høy kohesjon i klassene, og for å fasilitere at vi kan endre på noen elementer uten å endre på alle andre.  Applikasjoner med et grafisk brukergrensesnitt har gjerne tre viktige og ganske adskilte elementer som har ansvar for hver sin del (Holmstedt, 2007)

* presentasjon og visninger (i vår applikasjon → view-package)
* logikk og datalagring (i vår applikasjon → model-package)
* brukerimpulser og kontroll (i vår applikasjon → controller-pakcage)

Ved å følge dette mønsteret, medfører det at det i vår applikasjon blir enklere å bygge hver del, og at hver del blir enklere å forstå. Hver del har en høy kohesjon, og hvert element har både veldefinerte og avgrensede ansvarsområder.

Vår oppdeling av elementer gjenspeiles på pakkenivå, men den kan også foregå på komponentnivå, som for eksempel i Swing-komponenter.

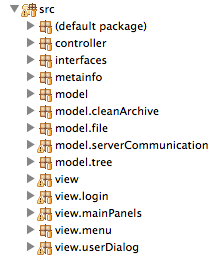
Ved å benytte en slikt designtilnærming, unngår vi å lage en applikasjon med et monolittisk design. Det vil si at logikken ikke er delt opp i spesialiserte klasser, eller at presentasjon inneholder mer enn bare presentasjon.

## Hvordan har vi implementert dette i vår løsning

Ved oppstart av prosjektet designet vi systemet etter prinsippene beskrevet ovenfor.

Vi bestemte oss for å ha et definert presentasjons-element (view), et kontroller-element og et model-element. Dette prinsippet ble fulgt etterhvert som vi utvidet applikasjonen gjennom iterasjoner, og underveis forgrenet vi ytterligere der hvor det var naturlig  innenfor hvert element basert på domene.

Oversikt over pakker i applikasjonen:



### Sammenhengen mellom de forskjellige elementene

I vår implementering så har presentasjoner en kontroller. Kontrolleren har også en model (eller flere) knyttet til seg, og fungerer i så måte som et bindeledd mellom brukergrensesnittet og model (logikk/data).

For å illustrere dette kan vi se på det som skjer ved start av applikasjonen:

Først instansieres det et LoginView:

LoginView loginView = **new** LoginView();

så en ServerCommunicatorModel:

ServerCommunicatorModel serverCommunicatorModel = **new**

      ServerCommunicatorModel();

og deretter sendes pekere til disse som parametre ved instansieringen av LoginController:

**new** LoginController(loginView, serverCommunicatorModel);

### Lyttere til brukerkomponenter

Her følger et eksempel på hvordan vi har implementert lyttere til brukerkomponenter i vår applikasjon, og de andre deler av vår applikasjon følger samme mønster.

LoginController har en indre klasse som implementerer actionlistener, og denne blir så lagt på brukerkomponenter i loginView. Den indre klassen i loginController ser slik ut:

**class** LoginListener **implements** ActionListener{

@Override

**public** **void** actionPerformed(ActionEvent e) {

...kode er implementert her

}

}  
I loginView blir så brukerkomponentene koblet til denne lytteren:

**public** **void** addListener(ActionListener listener) {

loginButton.addActionListener(listener);

closeButton.addActionListener(listener);

}

Kontrolleren lytter til brukerkomponenter, og oppdaterer presentasjon basert på hva brukerens handling var. Oppdateringen som skal gjøres hentes fra angitt model-klasse. På denne måten har vi separert presentasjon og logikk/data, og disse to elementene vet ikke om hverandre.

## Erfaringer med implementeringen

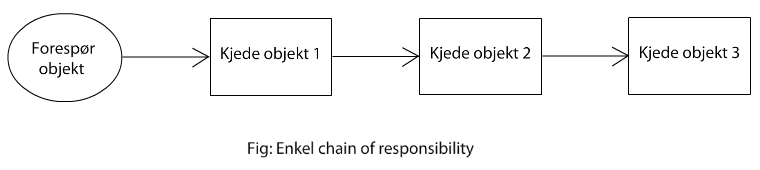
Vår læringskurve har vært bratt når det kommer til implementering av dette mønsteret i vår applikasjon. Det er ikke noe vi har kjennskap til fra før, men på tross av dette har vi tro på at vi har klart å gjennomføre en god implementering av dette. Nå er det ikke sikkert at vår implementering til fulle følger dette mønsteret i sin rette forstand, men vi har gjort et ærlig forsøk. Og som noen kloke hoder har uttrykt det: “Så lenge man prøver, finner det interessant og utforsker det, så har man kommet et langt stykke på veien”. Vi håper at vi er på rett vei.

Vi har opplevd stor nytteverdi av det, når det kommer til prosjektarbeidet. De forskjellige elementer er klart adskilt, noe som gjør at vi har kunnet jobbe i forskjellige deler av applikasjonen uten å komme i konflikt med hverandre.

Videre har vi fått kjennskap til utfordringer når det kommer til å ha tilgang til riktige objekter til riktig tid på rett sted. Dette medførte at vi på enkelte deler av applikasjonen implementerte noe som vi mener ligner et mønster som kalles chain-of-responsibility (se neste kapittel).

# Chain of Responsibility

Innen objektorientering er *chain of responibility* et designmønster. Mønsteret går ut på kjede sammen objekter, for så å gå langs kjeden for finne ut av hvilket objekt som kan håndtere ønskede forespørsler når de måtte oppstå.  Utvalgte objekter i vår applikasjon har en metode for å sette hvilket objekt som er den neste i kjeden etter den selv, og minst en metode for å ta for seg forespørsler objektet kan ta seg av.

Hvis objektet får en forespørsel den ikke kan ta seg av, så sendes den samme forespørselen til det neste objektet i kjeden. Dette gjentas helt til slutten av kjeden har blitt nådd hvis ingen av objektene i kjeden tar på seg ansvaret for forespørselen.

Et av hovedpoengene til dette mønsteret er å skape løse koblinger, det vil si at objektene har lite eller ingen kunnskap om, eller gjør liten bruk av, andre separate objekter. Dette er det som regnes som best praksis.

Det ble valgt å implementere *chain of responibility* mot slutten av applikasjonsutviklingen basert på faglærers innspill. Av den grunn er den ikke brukt fullstendig i strukturen, men har blitt brukt på noen objekter, som for eksempel ZipfileData.

## Eksempel på implementasjon - ZipFileData

ZipFileData-objektet, ble før implementeringen av *chain of responsibility*, sendt mye rundt for å være tilgjengelig til en hver tid, men blir nå bare forespurt via kjeden. Det er også brukt på MainView, der det viste seg at objekter som trengte den ikke fikk tilgang på en lett måte.

Først ble en ChainController konstruert. Den vet til en hver tid hvilket objekt som er den første og siste i kjeden. Den har to statiske metoder.

*addToChain* legger objekter til på kjeden.

Hvis objektet er den første i kjeden så blir den satt som første og siste:

**public** **static** **void** addToChain(ChainInterface latestInChain){

**if**(*firstInChain* == **null**){

*firstInChain* = latestInChain;

**if**(*lastInChain* == **null**){

*lastInChain* = *firstInChain*;

                }

         }

**else**{

*lastInChain*.setNextChain(latestInChain);

*lastInChain* = latestInChain;

         }

   }

Den andre metoden er *get*, som sender forespørselen *objectName* til den første i kjeden:

**public** **static** Object get(String objectName) {

**return** *firstInChain*.get(objectName);

   }

Det neste som ble konstruert var *ChainInterface*. Dette er et interface med to abstrakte metoder; *setNextChain* og *get*. Meninger er at klasser som implementerer dette interface skal sette *setNextChain* til å sette det neste objektet i kjeden basert på det objektet som kommer inn som parameter. Disse klassene skal også sette *get* metoden til å returnere dem selv eller den neste i kjeden vis det er ikke dem selv som er forespurt.

Eksempel fra ZipFileData:

      @Override

**public** **void** setNextChain(ChainInterface nextChain) {

**this**.nextInChain = nextChain;

   }

      @Override

**public** Object get(String objectName) {

**if**(objectName.equals("ZipFileData")){

**return** **this**;

         } **else** {

**return** nextInChain.get(objectName);

         }

   }

Når et objekt legge til i kjeden benyttes som sagt *ChainController*, for eksempel:

ChainController.*addToChain*(zipFileData);

Her legge en instanse av *ZipFileData* i kjeden. Dette gjøres som regel rett etter objektet har blitt instansiert.

Når dette objektet trengs senere i koden kan man bruke *ChainController* til å forespørre det slik:

ZipFileData fileData = (ZipFileData) ChainController.*get*("ZipFileData");

## Erfaringer med implementeringen

Ved å benytte *chain of responibility* ble koden enklere, da mange objektoverføringer kunne fjernes. Dette var en veldig positiv opplevelse for gruppen, da dette var noen av utfordringene vi støtte på underveis.

Og igjen, det er ikke sikkert at vår implementering til fulle følger dette mønsteret i sin rette forstand, men vi har gjort et ærlig forsøk. Og som noen kloke hoder har uttrykt det: “Så lenge man prøver, finner det interessant og utforsker det, så har man kommet et langt stykke på veien”. Vi håper at vi er på rett vei.

Ved å forespørre objektet i seg selv, ser vi at koblingen mellom objektene ikke blir fullt så løst som kanskje ønskelig. Dette er en forenkling vi har gjort, og som er et alternativ til å lage mer kompleks testing på ansvar og måter å håndtere forespørsler på. Likevel er det et objekt som er på kjeden som har en løs kobling når det blir forespurt, nemlig MainPanelsController. Når den blir forespurt setter den treet på plass WestPanel og bare returnerer null istedenfor å returnere seg selv, for dette er hva objektet skal gjøre når zipfilen fra server er ferdig lastet ned. Objektet trenger ikke å være direkte tilgjengelig for andre objekter da den ikke har noe annet andsvar enn å være kontrolløren til hovedpanelene.

Gruppen ser på det som en introduksjon til området og positiv erfaring, og vi ser fram til å lære mer om dette mønstret senere i studiet.

# XMLDecoder

Vi mottar XML via url som er oppgitt i oppgaven, og derfor må vi benytte en teknologi som er  i stand til å motta dette på en god måte - hvilket vi gjør ved hjelp av XMLDecoder. Denne leser XML data og disse blir satt inn i en eksisterende klasse.

## Hvordan vi har implementert dette i vår løsning

Vi har to klasser som bruker denne teknologien: BackupRequester og LoginRequester. Det som skjer i koden i disse klassene er kort beskrevet nedenfor.

Url konstruktøren mottar en link og det returneres en instans av klassen URL. Denne inneholder en peker til en ressurs på internett.

URL url = **new** URL(requestBackup);

URL’en brukes til å åpne en strøm til denne ressursen og dekodes med XML-dekoderen. Med en peker til instansen av denne klassen kan vi få tak i egenskapene ved denne ressursen:

XMLDecoder xmlDecoder = **new** XMLDecoder(url.openStream());

Her bruker vi XML-dekoderen til å lese dette objektet. Da brukes en av metodene til klassen får å få en instans av denne ressursen som et objekt.

Object object = xmlDecoder.readObject();

Det blir testet om objektet er en instans av en annen klasse. Hvis dette stemmer, blir objektetet typekonvertert til klassen CachingResultSetTableModel ( i LoginRequester brukes Poperties).

**if**(object **instanceof** CachingResultSetTableModel){

   caching = (CachingResultSetTableModel) object;

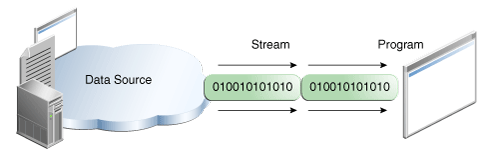
## En beskrivelse av våre erfaringer med implementeringen.

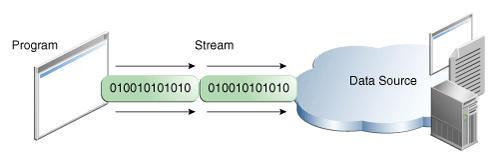
Implementeringen av denne teknologien var enkel og uproblematisk så lenge klassen det skulle typekonverteres til var kjent (CachingResultSetTableModel)*.* Vi brukte det vi tidligere hadde lært i forelesningen ved Viggo Holmstedt, og bygget videre på dette.

# Strømming av data og zipteknologi

En datastrøm kan beskrives ved at det er strømmer (flyt) av data som går fra en kilde til en destinasjon. Type kilder og destinasjoner kan variere, for eksempel kan både en kilde og en destinasjon være en fil, harddisk eller et annet program. En datastrøm kan overføre flere typer data, for eksempel bytes, primitive datatyper, karakterer og objekter.

En strøm kan overføre data rått og i andre tilfeller manipulere data, som for eksempel ZipOutputStream som både kan ta komprimerte og ukomprimerte data og skrive dette til en fil i Zip format.

Leser informasjon inn i et program ved hjelp av en input-stream:

Programmet skriver data til en destinasjon ved hjelp av en output-stream:

Oracle. (u.å.)

Klassen InputStream og OutputStream er abstrakte fordi de har henholdsvis metodene read() og write(). Dette fordi det skal være mulig å streame fra og til ulike typer medium. For eksempel vil klassen ZipInputStream ha definert en read() metode som gjør det mulig å lese en fil av type Zip

Holmstedt, V. (2013)

## Hvordan vi har implementert dette i vår løsning

### Introduksjon

Da vi startet undersøkelse av URL3 i eksamensoppgaven, ble vi oppmerksomme på følgende tekst  i url: “service=getzip”. Vi antok raskt at vi hadde med strømmeteknologi å gjøre og startet å utforske ulike strategier. Da vi åpnet url direkte i nettleser så vi også at karakterene “PK” ble gjentatt (med unntak av første karakter “!” som vi kommer tilbake til lenger ned) og dette bekreftet at data vi mottok fra url var av type Zip (komprimet arkiv).

### Lese data fra URL

Selv om vi ikke har direkte innsyn i hvordan dette er implementert på serversiden, kan vi gjøre en antakelse om data vi ser i URL er resultat av en ZipInputStream.

For å hente denne strømmen brukes en spesialisering av InputStream, kalt BufferedInputStream. Denne leser først ut et definert antall bytes (1024) fra strømmen og legger det i et minneområde (internt buffer array). Bruk av buffer er mer effektivt enn å lese en og en byte direkte fra en ekstern kilde, reduserer overhead. Etter hvert som det blir lest fra minneområdet blir det fylt opp igjen fra strømmen.

InputStream in = **new** BufferedInputStream(url.openStream(), 1024);

Det benyttes også en BufferedOutputStream som er tilsvarende, men utføres i motsatt rekkefølge:

OutputStream out = **new** BufferedOutputStream(**new** FileOutputStream(zip));

Videre brukes input- og outstream som input parameter til metoden copyInputStream(in,out) i klassen ZipExtractor for å kopiere fra input- til outputstream. Her brukes det også et eget buffer som brukes i kopieringen:

**byte**[] buffer = **new** **byte**[1024];

**int** len = in.read(buffer);

**while** (len >= 0) {

           out.write(buffer, 0, len);

           len = in.read(buffer);

      }

BufferedOutputStream bruker FileOutputStream i konstruktøren, slik at data leses inn i en fil med navn arc.zip som lagres i mappen runtimeArchive (TARGET\_DIR).

File zip = **new** File(*TARGET\_DIR* + File.*separator* + "arc.zip");

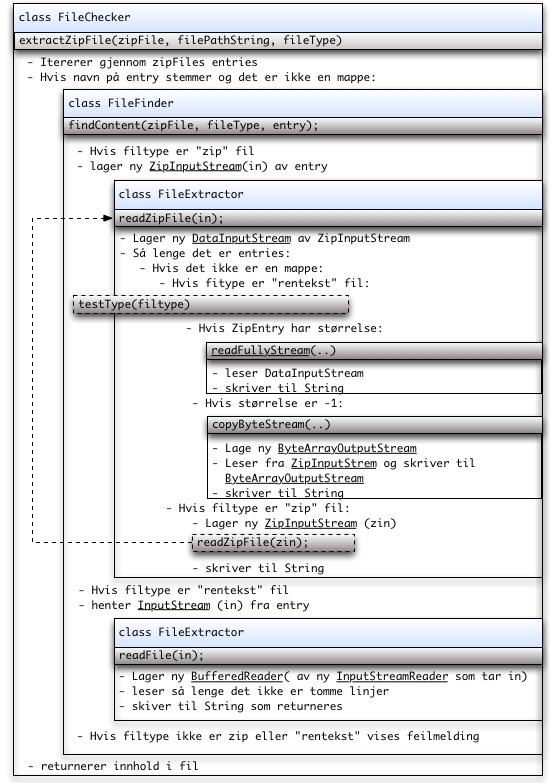
Merk at den første byten (“!”) leses, men blir ikke skrevet til BufferedOutputStream:

in.read(**new** **byte**[1]);

For å tilgjengeliggjøre zipFildata flere steder i applikasjonen, instansieres klassen ZipFileData som tar i mot zip som argument. Her larges zip i et klassefelt samt at det lages en ZipFile som gjør det enkelt å hente ut entries i zipfilen.

ZipFileData zipFileData = **new** ZipFileData(zip);

### Pakke ut innhold i zip

Vi ønsker å pakke ut inneholdet i forespurt fil og vise det i brukergrensesnittet. Dette initieres med metoden **extractZipFile**(zipFile, filePathString, fileType) i klassen **FileCheker**, som igjen bruker metoder i to andre klasser; **FileFinder** og **FileExtractor**

**extractZipFile**(zipFile, filePathString, fileType) i klassen **FileChecker**

Metoden returnerer til slutt content som er en String som brukes for å vise innhold zipFile  i brukergrensesnittet:

**return** content;

Det lages først en enumeration av type ZipEntry. Denne itereres så lenge det er entries:

**for** (Enumeration<? **extends** ZipEntry> entries = zipFile.entries(); entries.hasMoreElements(); ) {

For hver entry sjekker vi om filepath er lik forespurt fil og at den ikke er en mappe.

**try** { **if** ( !entry.isDirectory() && filePathString.equals( entry.getName())) {

Hvis dette validerer “true”, kjøres metoden **FindContent**(zipFile, fileType, entry) i klassen **FileFinder**

**FindContent**(zipFile, fileType, entry) i klassen **FileFinder**

Hvis filtypen som sendes inn i metoden er av type zip, så lages en ny ZipInputStream av entry:

ZipInputStream zin = **new** ZipInputStream( zipFile.getInputStream (entry));

Denne sendes som input-parameter i metoden readZipFile(in).

content = *readZipFile*(zin);

**readZipFile**(in) i klassen **FileExtractor**

Først lages en DataInputStream. Denne skal brukes til å lese “rentekst” filer i entries som returnerer en filstørrelse:

DataInputStream dataReader = **new** DataInputStream(in);

Alle entries i ZipInputStream leses så lenge de ikke er tomme:

**while** ((z = in.getNextEntry()) != **null**) {

Hvis filtype er “ren tekst”, gjøres først en sjekk av størrelse på entry: *Bakgrunn for dette er at zip-entries som er laget på OS-X platform returnerer -1, selv om de har innhold. Denne kontrollen gjøres for å unngå at lesing fra DataInputStream feiler.*

**if** (z.getSize() >= 0) {

Hvis getSize returnerer større eller lik 0 kjøres metoden **readFullyStream()** som bruker DataInputStream for å lese innholdet:

**byte**[] b = **new** **byte**[(**int**) z.getSize()];

dataReader.readFully(b);

line = line + temp + **new** String(b);

Hvis størrelsen på entry er mindre enn 0, kjøres metoden **copyByteStream()** som bruker en alternativ strategi for å lese innholdet. Her brukes en ByteArrayOutputStream som skriver til fra ZipInputStream:

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

**byte**[] buffer = **new** **byte**[1024];

**int** len = in.read(buffer);

**while** (len >= 0) {

baos.write(buffer, 0, len);

len = in.read(buffer);

..

}

Hvis filtypen er “zip”, lages en ny ZipInputStream som på nytt sendes til metoden **readZipFile** (seg selv).  Denne prosessen gjentas så lenge det finnes en zip fil inne i en zip fil, slik at alle filer til slutt blir pakket ut.

ZipInputStream zin = **new** ZipInputStream(in);

line = line + *readZipFile*(zin);

tilbake til **FindContent**(zipFile, fileType, entry) i klassen **FileFinder**:

Hvis filtypen er er av typen “rentekst” filtype hentes en InputStream fra entry:

InputStream in = zipFile.getInputStream(entry);

Denne sendes til metoden **readFile**(in) i klassen **FileExtractor**

content = *readFile*(in);

**readFile**(in) i klassen **FileExtractor**:

Her lages en ny BufferedReader av InptStreamReader som mottar InputStream:

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(in));

Denne leser så lenge det er linjer og skriver til String

**while** ((temp = br.readLine()) != **null**) {

line = line + temp + "\n";

tilbake til **FindContent**(zipFile, fileType, entry) i klassen **FileFinder**:

Hvis filtypen hverken er zip eller en av de definerte “rentekst” filtypene sendes en feilmelding til brukergrensesnitt:

content = AppInfoResourses.*WRONG\_FILE\_TYPE*;

### Lagre zipfil til disk

Nedlastning av hele zipfil til disk gjøres fra actionPerformed(..) i klassen **MenuController**.

Det lages en ny FileInputStream fra en metode i Klassen **ZipFileData** (peker til objekt hentes fra Chain). Denne sennes til igjen SaveFileDialog.

ZipFileData fileData = (ZipFileData) ChainController.*get*("ZipFileData");

InputStream in = **new** FileInputStream(fileData.getTheFile());

**new** SaveFileDialog().SaveFile(mainView, "zip", in, "backup.zip");

### Lagre enkeltfiler til disk

Det er også mulig å laste ned enkeltfiler i zipfil. Dette gjøres fra actionPerformed(..) i klassen **CenterPanelController.**

Først hentes peker til objektet fileData fra klassen **ZipFileData**  (fra Chain).

ZipFileData fileData = (ZipFileData)

ChainController.*get*("ZipFileData");

Fra en metode i denne klassen hentes en ZipFile:

ZipFile zipFile = fileData.getTheZipFile();

Her hentes også siste entry som ble satt sist gang metoden extractZipFile() i klassen **ZipExtracor** ble kjørt (det er den filen som sist ble trykket på i Jtree)

ZipEntry entry = fileData.getLastChoosenEntry();

Det lages så en InputStream fra entry som sammen med en del andre parametre som sendes til SaveFileDialog

InputStream zin = zipFile.getInputStream(entry);

## Erfaringer med implemementering

Vi bestemte oss tidlig for at vi ønsket vise innholdet i zipfilen i applikasjonen vår. Vi tok derfor utgangspunkt i et eksempel som ble vist i en tidligere forelesning, hvor innholdet i en zipfil blir lest ut til en String. Her leser vi ZipInputStream inn i en DataInputStream. Denne bruker metoden readFully som tar i mot et byte array hvor størrelsen på entry leses inn:

DataInputStream dataReader = **new** DataInputStream(in);

**..**

**byte**[] b = **new** **byte**[(**int**) z.getSize()];

dataReader.readFully(b);

Dette fungerte fint på filene som var tilknyttet brukeren “test”. Men da vi skulle teste applikasjonen på filer knyttet til en bruker som har lastet opp filer fra en MacBookPro med OS-X, opplevde vi problemer. Vi fikk feilmeldinger i konsoll knyttet til DataInputStream. Vi debugget da applikasjonen ved å skrive ut verdier i variabler og oppdaget at getSize returnerte verdien “-1” for de aktuelle entries.

Vi brukte lang til på å finne ulike løsninger og tok også opp dette i veiledningssamtale. Til slutt kom vi tilfeldigvis over en løsning hvor vi i stede for å lage en DataInputStream leser vi ZipInputStream og skriver ut i en ByteArrayOutputStream som vi igjen leser ut til String. På denne måten omgår vi problemet med å lese ut filstørrelse.

# SwingWorker og tråder

Vi så tidlig i prosessen at det ville være et behov for å implementere tråder i applikasjonen. Dette fordi det ved inlogging lastes ned store mengder med data fra server, og som medførte at brukergrensesnittet frøs. Vi prøvde først å løse dette med bruk av klassen Threads, men dette løste ikke problemet for oss. Vi tok derfor i bruk klassen SwingWorker.

C:\Users\Anders\Pictures\lasteinn.PNGSwingWorker er en klasse som gjør at man kan gjøre flere prosesser samtidig. Nedlasting fra serveren kjøres i bakgrunnen, og samtidig tegnes brukergrensesnittet og er fullt operativt. Dette kan ses ved at brukeren får melding om at data lastes inn. Når denne prosessen er ferdig vises det i fil treet. Det gjøres samtidig oppmerksom på at menyvalget “Last ned alt” ikke blir tilgjengliggjort før operasjonen er ferdig.

# Brukergrensesnitt

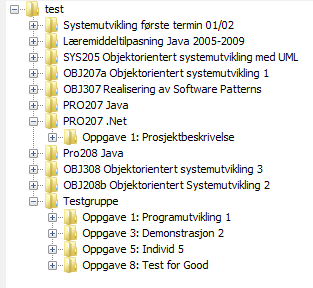
Vi vil i det følgende gå inn på spesiell tekniske aspekter når det kommer til implementering av brukergrensesnitt. Vi har i det store og hele brukt swing-biblioteket for å konstruere det visuelle brukergrensesnittet, med ett unntak som er nevnt under (save). Vi vil her ikke ta for oss de generelle swing-komponentene som vi har brukt i vår applikasjon.

## JTree

Når vi skulle avgjøre hvordan vi skulle framstille de mottatte data for brukeren, fant vi at det å presentere det i en kjent struktur for brukeren var ønskelig. I dette henseende fant vi at JTree tilfredsstilte våre krav og ønsker. Dette er en god måte å fremstille filstrukturer, og kunne gjøre filstrukturer og annen hierarkisk data oversiktlig.

### Generelt om JTree

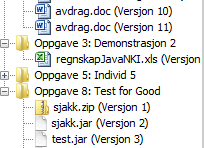
JTree er en klasse i javax.swing biblioteket . Det består av komponenter som vises som et hierarki av data i en trestruktur.



Første oppføring i et JTree blir kalt en rotnode (root node). Alle elementene som er festet til roten blir kalt grener (branch nodes).  Bladene er det ytterste punktet i treet og består av filer.

### Hvordan vi har implementert det i vår løsning

Vi har valgt å kun åpne den første grenen under roten ved oppstart. Dette fordi vi ønsker at brukeren skal å få en god oversikt over de kurs som det er levert oppgaver i. Deretter kan brukeren velge hvilke grener som skal åpnes videre.



Selve treet konstrueres basert på innhold hentet fra informasjonen som blir mottat av URL 2 i oppgaven. Navnet på roten bli satt til det samme som brukernavnet som er logget inn. Hyppige og velkjente filtyper blir gitt vist med ikoner. .

Lytteren blir lagt til på samme måte som loginView.( Se kapittelet om applikasjonsarkitektur.) Når en fil (som er et blad i treet) blir trykket på, så vises innholdet i denne filen på et annet sted i vinduet. Dette er mulig fordi at “bladene” er instanser av en egen konstruert klasse. Når det blir trykket på en node, blir det testet på om noden er et “blad” eller en gren.

### Erfaringer med implementering av JTree

Vi fant at JTree var en nyttig komponent å ta i bruk i vårt prosjekt. Det var en god teknologi å jobbe med og viste seg å være konfigurerbar for den bruken vi ønsket.

## Dialog med brukeren

### Save

For å lagre filene bruker vi FileDialog. Den gir oss muligheten til å velge en filen. Når man åpner dialogvinduet er det umulig å fortsette bruken av applikasjonen inntil man lukker dialog vinduet.

#### Erfaringer med impelmentering av Save

Bakgrunnen for at vi valgte å bruke awt.FileDialog framfor JFileChooser i swing-biblioteket, var at det for brukeren oppleves mer naturlig å få operativsystemets fildialog opp. Nå er det muligheter til å bruke en metode for å sette det visuelle uttrykket likt operativsystemet man jobber på. Dette prøvde vi ut, men det ga ikke ønsket resultat. Derfor landet vi på awt.FileDialog.

# Avslutning

Vi har i denne rapporten beskrevet de ulike tekniske sider ved vår applikasjon. Noen aspekter har vi gått litt mer i dybden på enn andre, og noen er utelatt i sin helhet da de ikke innebærer noen tekniske utfordringer etter vår vurdering.

I vår beskrivelse har vi tatt for oss noen generelle betraktninger, vi har sett på hvordan vi har implementert det i vår applikasjon, og vi har kommet med noen erfaringer vi har gjort oss underveis i implementeringen.

# Litteraturliste

Holmstedt, V. (2007) *Mønstre og tekniske applikasjoner* Bergen: Fagbokforlaget

Holmstedt, V. (2013, 2.opplag) *Objektorientert programmering med Java*

Bergen: Fagbokforlaget

Oracle. (u.å.). *I/O Streams*. Hentet 9. desember, 2013, fra <http://docs.oracle.com/javase/tutorial/essential/io/streams.html>