HiOA

TDK

Ingeniørfag – data

Programutvikling

Eva Hadler Vihovde

Prosjektoppgaven 2015

- Produktdokumentasjon -

Alternativ 1

- Forsikring -

Gruppe #14

Studentnavn	Studentnummer
Marius Alexander Skjolden	s114143
Hans Christian Nenseth	s236334
Hans Petter Osvold	s929913

Innholdsfortegnelse

Introduksjon

Dette dokumentet er skrevet i forbindelse med en prosjektoppgave i programutviklingkurset DATS1600 ved Høgskolen i Oslo og Akershus. Dokumentet er en dokumentasjon av produktet vi har utviklet. Vi ønsker å starte med å takke faglærer Eva Hadler Vihovde for all den gode undervisningen vi har hatt i Java kursene de to foregående semestre. Utover dette ønsker vi å takke studentassistentene Daniel Reinholdt og Lukas David Larsed, for at dere alltid hadde tid til oss. Programmet er en Java desktopp applikasjon utviklet i JavaFX rammeverket og skal simulere et kunde-relasjon-system.

Oppgavebeskrivelse

> Sakset fra fagets hjemmeside

Dere har fått i oppdrag å lage et java-program for et forsikringsselskapet. Programmet skal kunne registrere selskapets kunder, opprette ulike typer forsikringer, registrere skader og utbetale erstatninger. Systemet skal kunne generere ulik type statistikk og historikk, i tillegg til at det skal være enkelt å hente ut forskjellig informasjon. De ulike forsikringstypene selskapet i dag tilbyr er bilforsikring, båtforsikring, hus- og innboforsikring, forsikring av fritidsbolig og reiseforsikring. De har imidlertid planer om å utvide sitt forsikringstilbud og programmet må derfor bygges opp på en måte som gjør at det enkelt å bygges det ut med flere forsikringstyper etterhvert.

Forsikringsselskapet tilbyr 10% i rabatt på den årlige forsikringspremien for forsikringskunder som defineres som "totalkunde". For å bli totalkunde må kunden ha minst tre ulike typer forsikringer.

Med hensyn til bilforsikring følger selskapet det vanlig bonus-systemet som gjelder i Norge, der kunder som kjører skadefritt opparbeider seg bonus og øker derved sin rabatt på bilforsikringen med en gitt prosent for hvert år. Kunden kan pr idag ikke opparbeide høyere bonus enn 75%. Hvis vedkommende skader bilen, vil bonusen minske og bilforsikringspremien øke. Bonussystemet er imidlertid til revidering og systemet skal derfor lett kunne oppdateres og tilpasses de bonusregler som til enhver tid måtte gjelde. (Sjekk reglene som gjelder i andre forsikringsselskap.)

Programmets hovedoppgave

Det programmet vi har utviklet er et utsnitt av et større program for behandling av forsikrings selskapets kunder, forsikringer og skademeldinger. Et slikt datasystem vil kunne ha mange moduler som for eksempel en kundeportal på web / mobil, en desktopp applikasjon for ansatte i firmaet, et statistikk verktøy for administrasjonen.

En av våre største avgjørelser var å la programmet kun fokusere på forsikringsagenten, som gjerne sitter i et kundesenter og kommuniserer med kunder over telefon eller epost. Det vi har utviklet kan derfor kun sees på som en bit i et større puslespill. Vi henviser til Prosessdokumentasjonen.

Programmet lar brukeren ta hånd administrasjon av kunde, forsikring og skademelding objekter. Det er det blitt implementert et *googlesque* fulltekst søk, som lar brukeren søke igjennom hele modellen ved et enkelt søk. Det kan med et enkelt tastetrykk genereres både statistikk og rapporter både ut ifra søkeresultater og fra modellen som helhet.

Utover dette valgte vi å implementere et sentralisert strengbibliotek. En fin sideeffekt av streng biblioteket er at vi fikk utvidet programmet til å inneholde støtte for tre ulike språk: norsk bokmål, engelsk, italiensk.

Programmets oppbygging

Programmet er organisert inn under syv grunn pakker, og følger prinsipper om model view controller. Det vil si at programmet har en egen pakke for data (model), brukergrensesnitt (view) og arbeids-logikk (controller). Det finnes en pakke som tar hånd om å lese / lagre brukerens tidligere preferanser til fil. Ut over dette finnes det pakker som inneholder statiske klasser som config, localization, og validator.

Config klassen inneholder en rekke konstanter som kan brukes fritt andre steder i programmet. Localization er et sentralisert strengbibliotek som lar programmets tekst bli lagret et sentralt sted slik at disse lett kan endres ved behov. En hendig fordel ved Localization implementasjonen er at brukeren ved et enkelt tastetrykk kan endre applikasjonens språk.

Programmet er en JavaFX applikasjon. Når man starter programmet for første gang vil man først få opp et lite introduksjonsvindu hvor brukeren kan åpne eller opprette et nytt prosjekt. Det her også her brukeren velger sitt prefererte språk. Brukerens valg vil bli lagret i en preferanse fil, slik at ved neste oppstart så vil ikke nødvendigvis dette introduksjonsvinduet blir vist.

Etter at hoved-applikasjonen har startet, så vil all kommunikasjon mellom brukergrensesnittet og

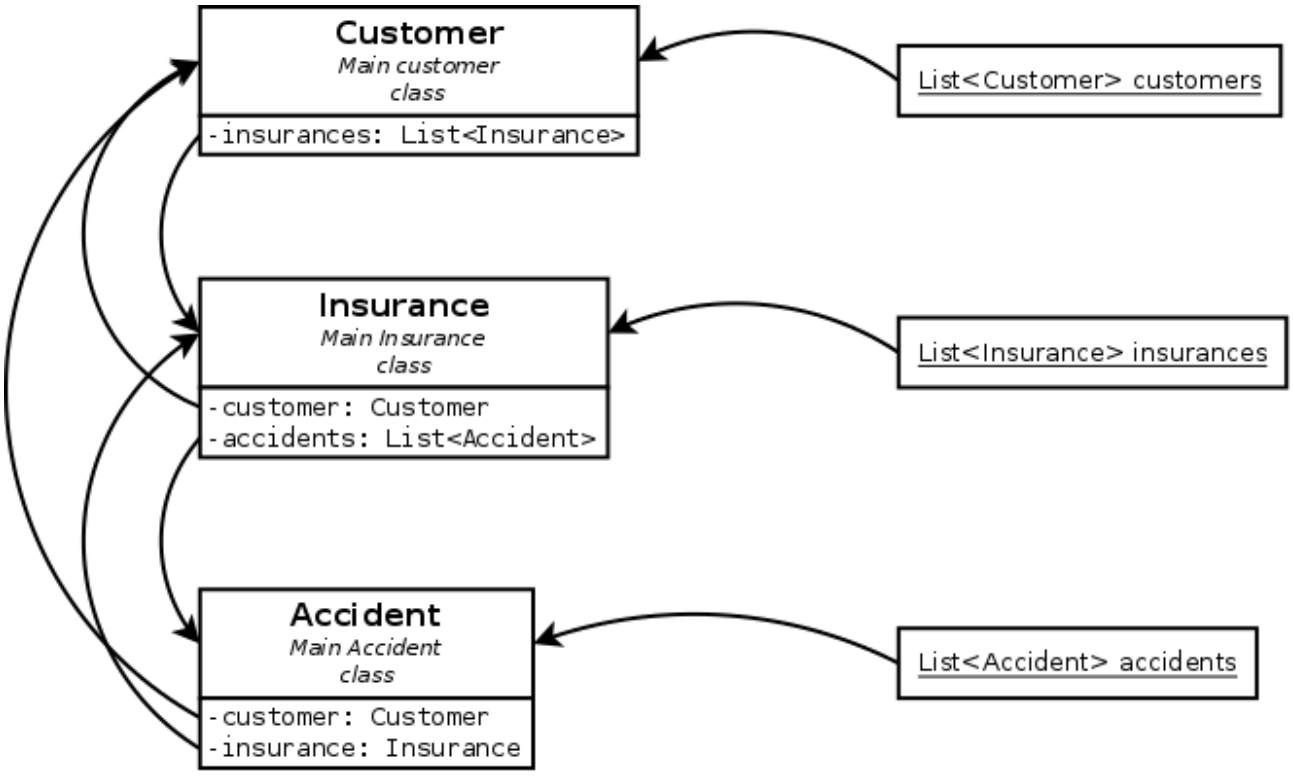
modellen rutes innom en Mediator klasse som tar i mot et Signal-objekt og et Payload-objekt. Den faktiske kommunikasjonen i form av knappelyttere er stor forbruker av Lambda uttrykk.

Datastruktur

Selve grunnmuren til applikasjonens datastruktur er Java Collections rammeverket. På bunnen av datastrukturen blir det benyttet en HashMap av generisk type med Streng som nøkkel og en generisk List som verdi. Tanken her er at hver hovedtype (Person, Skade, Forsikring) blir definert her med sine lister. Samtlige hovedtyper og internrelasjoner mellom konkrete objekter benytter LinkedList som datastruktur.

Datastrukturen blir lagret i en datafil på brukerens datamaskin. Programmet benytter seg av Storage.getInstance() for å lese og skrive til fil. Dette er et eksempel på bruk av Singleton design pattern. Det ble valgt å implementere et singleton design pattern for å sikre at det kun kan være en enkelt instans av datafila oppe til en hver tid. Samtidig kan man se for seg at ulike deler av systemet skal benytte seg av denne datafilen samtidig.

Klasserelasjoner



Modellen

Pakken som inneholder datamodellen består av en ulike pakker som representerer ulike entiteter som for eksempel forsikring, skademelding og person. Hver entitet implementerer grensesnittene *Model, FullTextSearch* og *Serializable.* Hensikten med grensesnittet Model er å enkelt og raskt kunne identifisere modellens type. De ulike modellene er agnostiske og vil fungere uten hverandre, dermed vil det ikke være noe problem å utvide denne med flere typer.

Ettersom samtlige entiteter er komplekse, har vi implementert builder-pattern. Fordelen med dette er at det å opprette objekter blir både enklere og samtidig mer elegant. En annen fordel er modellen nå enkelt kan utvides med flere datafelter uten at hele arkitekturen til entiteten må vurderes endret.

Søking

I modellen er det implementert et fulltekstlignende søk. Det betyr at vi har laget et søk som strekker seg over de fleste felter til samtlige modeller. Det har gitt oss muligheten til å implementere et googlesque søk, hvor et nøkkelord er nok til å treffe på data over hele datastrukturen (Person, Forsikring, Skademelding).

FullTextSearch.java er implementert som et «interface» med diverse metodesignaturer definert. Den viktigste metoden her er den boolske «query». Klasser som implementerer dette interfacet blir da oppfordret til å teste alle sine private datafelter og returnere en boolsk verdi som resultat av dette. I alle de konkrete klassene har vi valgt å bruke denne innebygde Stringmetoden «contains» og «toLowerString» for å støtte «case insensitive søk».

På denne måten kan vi gjøre enkle Stream API søk som «liste.stream().filter(i -> i.query(«søkeord»))..». Hva listen består av i eksempelet over er da irrelevant sålenge den implementerer FullTextSearch.

Forslag til søkeord:

Eva, Oslo, 10000, Bil/Car/Auto alt etter som hvilket språk som er aktivert.

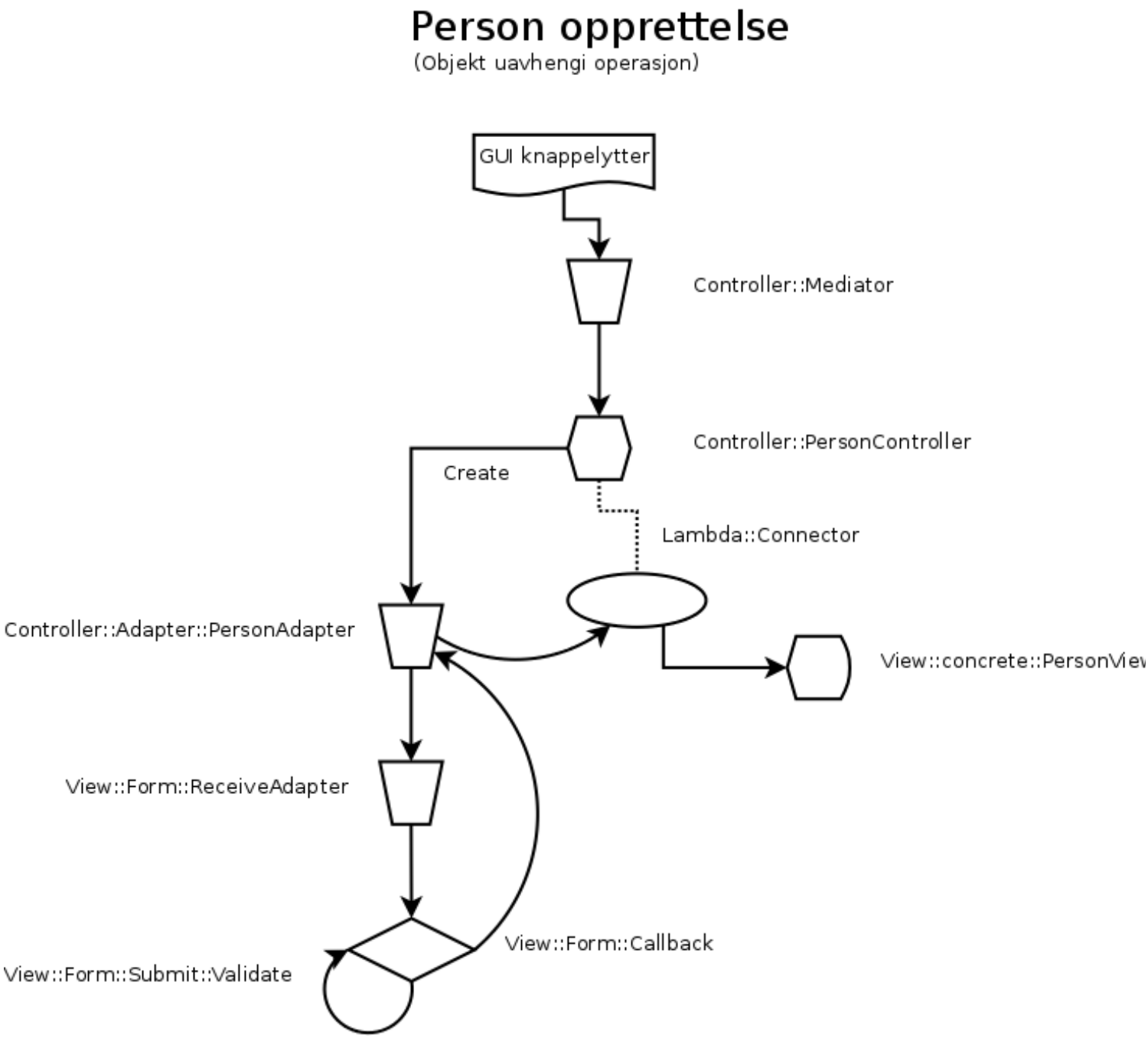
Bruk av design patterns

Programmets hovedarkitektur er sterkt inspirert av MVC pattern. Det har en agnostisk, «loose coupling» modell som inneholder all data. View pakken inneholder konkret brukergrensesnitt. Mens controller pakken inneholder controllerne samt de ulike form adapterne.

Form adapterne fungerer som et bindeledd mellom modell og brukergrensesnitt.

Det blir brukt et mediator pattern for å lese og sende signaler til ulike deler av applikasjonen. Mediatoren er logikkens ambassadør for uavhengige operasjoner.

Storage klassen, som tar hånd om lesing og lagring til fil, samt Localization klassen, bruker singleton pattern slik at vi er sikre på at det kun er en åpen instans av de ulike filene.



Personopprettelse er en av de «objekt uavhengig operasjonene» i systemet. Søking er et annet eksempel på en objekt uavhengig operasjon. Dette betyr at et nytt objekt blir skapt i applikasjonen uten å være i direkte relasjon med andre. På toppen av type hierarkiet sitter Person typen som er den eneste av de tre entitet-typene (Person, Skade, og Forsikring) som kan leve uten pekere.

Opprettelseprosessen starter ved at en av knappelytterne i viewet sender et *Signal* og en *Payload* til Mediatoren. Signal-objektet skal identifisere signal typen, og dermed hvilke kontroller som skal

benyttes. Hvis det skal sendes data til kontrolleren sendes dette igjennom Payload-objektet. Dette er en generisk prosess for all kommunikasjon til Mediatoren. Andre prosesser kan dra bedre nytte av Payload-objektene.

Mediatoren fungerer dermed som et bindeledd mellom knappelytterne i det grafiske brukergrensesnittet og kontrollerne. Grunnen til at dette er hensiktsmessig er at kontrollerne og all dens «arbeids logikk» både blir skjult og begrenset for brukergrensesnittet.

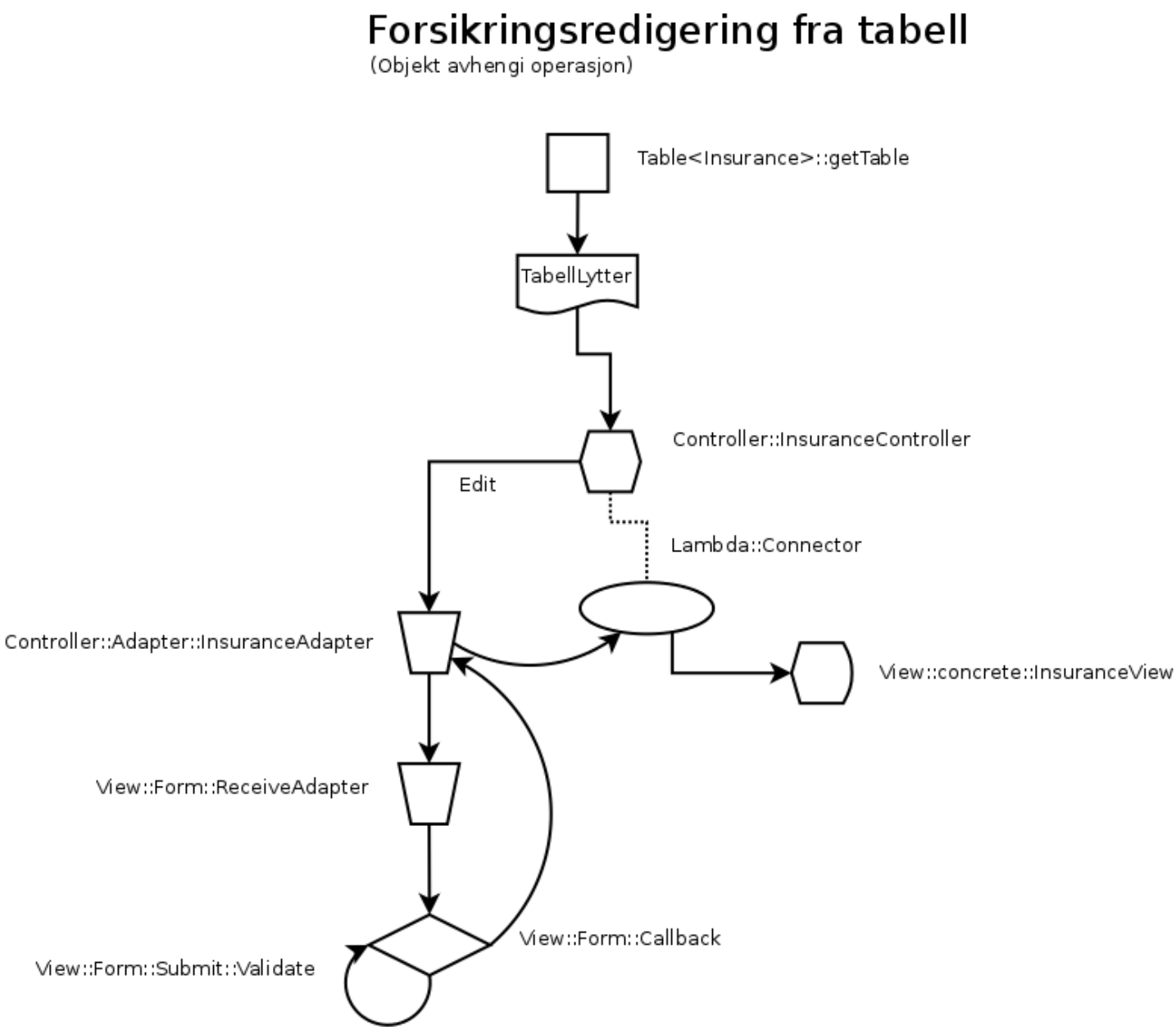
Det har blitt tatt hensyn for å unngå at Mediatoren blir et enormt «super-objekt» i kontroller pakken, blant annet hvordan kontrollerne kan snakke med hverandre. Les mere om dette i kapitelet om «Objekt avhengige operasjoner». Etter at Mediatoren har prosessert signalet og sendt Payload-objektet videre til riktig kontroller, så tar nevnte kontroller over prosessen. Mediatoren har ingen videre involvering fra dette punktet.

Person-kontrolleren instansierer et nytt objekt av typen PersonAdapter som inneholder blant annet regler og definisjoner for hvordan en Person skal valideres og se ut. De ulike form adapterne er et ledd i kjeden som frigjør modellene, og lar de være helt agnostiske. Adapterne fungerer som et lim mellom modell og visning, samtidig som de tar hånd om validering og sjekk av data.

Etter at Person adapteret har blitt initialisert blir det instansiert et Form-objekt. Dette objektet kommuniserer med adapteret, og setter opp den konkrete visningen av adapteret. Form-objektets hovedfunksjon er å lese reglene og beskrivelsene til adapteret og skape visning ut ifra dette.

Formet har nå tegnet et skjema for brukeren. Når formets knappelytter blir fyrt av looper form-objektet igjennom regelsettet opprettet i kommunikasjon med adapteret. Hvis dette regelsettet feiler, vil valideringsmetoden vise feilmelding(er), og midlertidig stanse prosessen. Denne løkken fortsetter helt til alle feltene er fylt inn korrekt etter de reglene adapteret har instruert. Først når valideringen er feilfri blir et signal sendt tilbake til adapteret som nå overtar prosessen.

Person adapteret vil i dette tilfellet instansiere en ny Person. Neste steg er at den kontakten kontrolleren opprettet med adapteret bestemmer hva person adapteret skal gjøre videre med person-objektet. I alle opprettelse prosesser er neste steg nemlig å vise det ny opprettede objektet.



Tabeller er et sentralt komponent i applikasjonen, og brukes til all listevisning av objekter. Det er implementert en abstrakt generisk Tabell klasse som er modellert slik at den kan utvides med konkrete tabell definisjoner. En av disse definisjonene er sub-klassen InsuranceTable (forsikringtabell).

Alle tabeller som utvider Tabell klassen vil få tre injisering metoder for definering av Lambdauttrykk, som beskriver en hendelseprosess. Disse metodene er «rediger», «vis», og «dobbeltklikk».

Standarden er at «dobbeltklikk»-hendelsen linker direkte videre til «vis»-metoden. «rediger»-hendelsen aksesseres via et høyreklikk og valg av rediger i kontekstmenyen.

I alle tabell-initialiseringer defineres implisitt hvilken type tabellen skal vise. Hvert av objektene vil bli knyttet opp mot de ovennevnte hendelse-prosessene igjennom JavaFX TableViews selectionModel. Det muliggjør et godt gjenbruk av Tabell klassen og knappelytterne til alle hovedtypene (Person, Skade, Forsikring), og flere ved behov.

Rediger lytterne som dette kapitelet tar for seg, er i alle forsikringstabeller knyttet til sin types kontroller. I dette tilfellet forsikrings-kontrolleren. Til denne klassen sendes det konkrete forsikrings-objektet.

Når dette objektet har nådd kontrollerens statiske «endre» metode, vil den derfra rutes i riktig retning basert på metodekallet «identifiser». En metode samtlige modeller er påkrevd å ha implementert.

Kontrolleren oppretter nå et forsikringsadapter i «redigerings modus», og de neste stegene er nokså like som i forrige kapitel.

Forslag til utvidelser.

For å kunne selge dette programmet til næringslivet må det utvides med et bruker system med muligheter for å opprette, endre og slette brukere. Disse brukerne vil til en viss grad trenge ulike rettigheter og brukergrensesnitt.

En annen utvidelse vil være å utvikle en kundeportal som et nettsted eller en serie av mobilapplikasjoner.

En tredje og enklere utvidelse vil være å videreutvikle det grafiske brukergrensesnittet ved for eksempel bruk av CSS som Java FX har god støtte for.

Et konkret forslag til utvidelse av modellen.

Forsikringsselskapet ønsker å utvide porteføljen til også ta hånd om motorsykkelforsikringer.

For å få dette til trengs det en ny modell type som gjerne arver den abstrakte klassen Vehicle. Ettersom modellen både er agnostisk og bygget på prinsipper om «loose coupling» så vil det å utvide modellen være en svært lett oppgave, med få eller ingen uventende problemer.

InsuranceType som en enum må oppdateres til å inkludere type motorsykkel. Neste steg vil være å oppdatere controllerne og view til å inkludere motorsykkelforsikring. Dette betyr at for å implementere en ny forsikringstype så vil hele MVC kjeden påvirkes. Med tanke på at kodebasen vår er såpass «liten» er ikke dette det største problemet.

Men hvis dette programmet skulle ut i den virkelige verden, med virkelige kunder, budsjetter og lengre tidsfrister ville det muligens vært naturlig å tenkt mer modulært og implementert et type-plugin-system. Da ville en plugin for eksempel inneholde alle de ovennevnte komponentene, og dermed kunne plugges rett inn i systemet dynamisk.

Systemkrav

Det settes ingen krav til maskinvare utover at det bør være en datamaskin av nyere dato. Det har blitt generert en datafil i underkant av 3 Megabyte.

Java versjon

Ettersom bruk av Lambda uttrykk er en integral del logikken som gjennomsyrer hele programmet fra søking til generering av statistikk, er minimums kravet for å kjøre programmet JDK 1.8.

Det er ønskelig at JDK 1.8u40 er tilgjengelig på maskinen programmet skal kjøres på grunnen noen Java FX komponenter.

Referanser

Applikasjonen inneholder åpne ikoner fra http://famfamfam.com/ .