Evil Insurance Co

Gruppe 36

Audun Brustad s236341

Rudi Yu s231776

Introduksjon

- Om rapporten
- Oppgavevalg
- Oppgavetolkning
- Tekniske detaljer

Prosessdokumentasjon

- Arkitektur
- Arv og polymorfisme
    - Swing
    - Forsikringer o.l
- Arbeidsfordeling
- Utfordringer

Produktdokumentasjon

MVC

- Datastruktur (Model)
- Kontroller

GUI

Søking

Custom Swing-komponenter

Serialisering og lagring

Packages

Kommentering

Mangler/nedprioritering

Introduksjon

Om rapporten

Rapport for java-gruppe nummer 36, bestående av Audun Brustad og Rudi Yu. Rapporten er delt opp i tre deler: introduksjon, prosessdokumentasjon og produktdokumentasjon. Introduksjonen består av forutsetninger, tekniske detaljer, vår tolkning av oppgaven og valg av oppgave. Prosessdokumentasjonen beskriver hvordan vi jobbet, hvilke valg vi kom fram til, fremgangsmåter og utfordringer vi støtte på på veien. Produktdokumentasjonen er en kort, teknisk beskrivelse av programmet vi har laget. Det er brukt skisser og kodeeksempler for å på en best mulig måte gi utenforstående forståelse for programmet.

Oppgavevalg

Vi valgte oppgave 1 som gikk ut på å lage et datasystem for et forsikringsselskap. Programmet skulle kunne registrere nye kunder, opprette forskjellige typer forsikringer koblet til disse kundene, registrere skader og utbetale erstatninger. Systemets skulle genere statistikk fra dataene, og ta vare på alle opprettelser, selv om kundene blir inaktive.

Oppgavetolkning

For å kunne ha et funksjonelt og brukbart program til innleveringsfristen har vi gjort noen valg og prioriteringer. Oppgaven gav oss heller stor frihet til implementering av funksjoner, og vi har valgt å heller satset på ting vi mente var viktig. Vårt mål har vært å sitte igjen med et så robust og oppgraderbart program som mulig. Det vi mener med det er at det skal være mulig å endre, legge til og fjerne klasser, metoder osv. uten å risikere at programmet blir gjort ubrukelig. Dette var en tidkrevende prosess og noe som ikke nødvendigvis kan sees ved bruk av det kjørbare programmet, men som vi følte var viktig. På grunn av denne prioriteringen er det dessverre noen funksjoner som har blitt nedprioritert. Den funksjonen som kanskje har lidd mest, men som vi hadde store planer for er statistikk, som selv om den er tilstedeværende og panelet vises når du trykker i menyen, har veldig lite back-end.
Vi bestemte oss også tidlig for ikke å ha en logg inn-funksjon, og heller ikke opprettelse av forsikringsselskapets ansatte.

Mål

Ved prosjektstart satte vi opp følgende mål for oppgaven:

MVC

Det skal brukes MVC-arkitektur (Model View Controller) i programmet, slik at programlogikken og brukergrensesnittet er skilt fra hverandre. All utveksling av informasjon mellom disse skal håndteres av en kontroller.

Generell kode

Programmet skal ha «generell kode», altså at ting som blir brukt flere steder, i stedet kan kalle opp generell metode. Vi har også brukt ferdig deklarerte konstanter som kan brukes i alle klassene. Dette er gjort slik at det er enklere å utvide eller endre programmet senere, i tillegg til at det sparer oss for arbeid.

Enkelt brukergrensesnitt

Programmet skal ha et enkelt og intuitivt brukergrensesnitt slik at nye brukeren enkelt kommer inn i systemet uten en bratt læringskurve. Det skal brukes minst mulig undermenyer og det skal brukes popup-vinduer ved registreringer og visning i stedet for at hovedvinduet oppdateres.

Tekniske detaljer

Utviklermiljø

Ved utvikling av programmet er det brukt NetBeans IDE versjon 8.0.2. Vi valgte NetBeans på grunn av støtte for GIT, som vi tidlig bestemte oss for å bruke, samt flere andre nyttige funksjoner. Logo/ikoner og annen grafikk er laget i Adobe PhotoShop, og er tegnet fra bunnen av for hånd. UTF-8 er brukt som tegnoppsett og vi har unngått å bruke norske bokstaver og spesialtegn i koden og kommenteringen.

Versjonhåndtering

Vi har brukt GIT til versjonhåndtering, via GitHub, terminal og støtte via NetBeans IDE. Lagring av prosjektet er gjort sentralt i en repository på GitHub.

Programvarekrav

Programmet er kompilert med javac 1.8.0.25 og testet med samme versjon. Programmet er kodet og testet på både Microsoft Windows 7 og 8.1, samt Mac OS X for se om oppførsel og utseende endret seg nevneverdig, og eventuelt endre noe med hensyn til dette.

Prosessdokumentasjon

Arkitektur

Til programmets arkitektur er det brukt designmønsteret MVC, kort for Model View Controller. Det vil si at programmet er delt i model, som er data-delen av programmet og view, som er programmets brukergrensesnitt. Med dette mønsteret er dataene og brukergrensesnittet helt separert, og en endring i brukergrensesnittet vil ikke ha noen virkning på datahåndtering, og omvendt. Som bindeledd mellom disse to brukes det en kontroller som har tilgang til og kommuniserer med begge partene. Vi har separert de forskjellige modulene i sine egne pakker – kontrollere i controllers-pakken, brukergrensesnitt i views og views.registrations-pakkene og data i models-pakken. Til forskjell fra annen arkitektur er den største forskjellen at brukergrensesnitt-klassene trenger get-metoder for at kontrollerne kan kommunisere med komponentene.

I tillegg til å skille datahåndtering fra brukergrensesnitt

deles koden som ellers måtte ha vært i samme klasse, opp i flere klasser. Uten denne arkitekturen kunne fort ha blitt noen tusen linjer kode i samme klasse, som ville gjort den svært vanskelig å finne fram i og programmet svært vanskelig å endre eller oppgradere ved en senere anledning.

Arv og polymorfisme

Vi la tidlig en plan om å i stor grad benytte oss av arv og polymorfisme for å kunne gjenbruke så mye kode som mulig. For å gjøre koden så gjenbrukbar og enkel som mulig, følte vi at dette var en nødvendighet.

Forsikring

Forsikringsregistreringen er delt opp i et klassehierarki. Dette er gjort så vi slipper å skrive svært mye av kodene mer enn én gang, i tillegg til at det er god praksis. På toppen av Forsikring-hierarkiet er et «Insurance»-objekt som tar imot alle data som er felles for alle forsikringstypene. Deretter er det delt i tre undergrupper: «PropertyInsurance» som tar imot data for en forsikring til en eiendom, «VehicleInsurance» som tar imot data for en forsikring til et kjøretøy og «TravelInsurance» for reiseforsikring. «PropertyInsurance» er igjen delt i to deler: «HouseInsurance» som er for hus- og innboforsikringer, og «LeisureHouseInsurance» som er for fritidshusforsikringer. «VehicleInsurance» er også delt i to undergrupper, «BoatInsurance» for registrering av båtforsikringer og «CarInsurance» til registrering av bilforsikringer.

Swing

Med Swing-komponentene gir arv svært god avkastning. De Swing-komponentene vi visste kom til å bli mest brukt ble redefinert i form av en «personalisert» komponent. Eksempler

på slike tilfeller er klassene «CustomButton» og «CustomTextField». Ved å la disse klassene arve de opprinnelige komponentene fra super-klassen sin i Swing endret vi utseende på alle av den gjeldende komponenten, i stedet for å måtte «refactore» hver eneste instanse av komponenten. På grunn av dette er det ikke mange steder brukt komponenter direkte fra Swing og ikke gjennom en egen versjon, men likevel er noen det. Eksempler på en komponent brukt rett fra Swing er JTabbedPane, som trengs så lite endring i forhold til standard utseende at vi valgt å ikke prioritere en personalisert versjon.

Arbeidsfordeling

Koding

Vi bestemte oss tidlig for å dele oppgaven i to deler: første delen er kort sagt «back end», bestående av lagring og objekter osv.. Andre delen er GUI (brukergrensesnitt), forms, vinduer og ellers annet utseende. Siden vi valgte å satse på MVC og Rudi fra tidligere hadde noe erfaring med det, fant vi raskt ut at det var lurt han satset på «back end»-delen, og derfor Audun på GUI. Selv om vi delte oppgaven opp på denne måten har vi jobbet tett sammen i hele perioden, og jobbet «synkront» på samme funksjon fra hver vår side. Vi har heller ikke systematisk jobbet etter denne inndelingen, og begge har vært innom alt. Denne oppgavefordelingen gjorde også et såkalte merge-feil i GIT skjedde svært sjelden, da vi stort sett jobbet i forskjellige klasser.

GIT

GIT-oppsett og vedlikehold har i hovedsak vært tatt hånd om av Rudi, mye på grunn av at han har gjort det ved tidligere tilfeller og visste bedre hvordan det gjøres.

Rapport

Det er i hovedsak Audun som har skrevet rapporten, da vi støtte på noen utfordringer «backend» mot slutten av prosjektperioden det i hovedsak var Rudi som tok seg av.

Utfordringer

Programstruktur

MVC

Layout-managere

En av utfordringene vi støtte på i startfasen av prosjektet når brukergrensesnittet skulle utvikles var hvilken av Layout-managerene i AWT som skulle brukes. Dette var også et felt vi var heller ferske på, da vi ikke har hatt mye undervisning i dette, annet enn «FlowLayout». Med prøving og feiling har vi nesten prøvd ut alle de forskjellige Layout-managerene, og kommet fram til en miks av «GridBagLayout» og «BorderLayout». «GridBagLayout» er i hovedsak brukt i skjema-panelene, som for eksempel ved registrering av en ny kunde. «BorderLayout» i hovedsak brukt på hovedvinduet, der det bare legges til et panel og en tabell.

Konflikter med GIT

Vi hadde ikke spesielt mye erfaring med GIT noen av oss, men vi hadde vært borti det med et tidligere prosjekt. Den første tiden ble derfor brukt til å repetere GIT og å lage til et ordentlig system med forskjellige «branches» vi lagret til. Selv om vi delte oppgaven i to separate deler vi jobbet hvert til vårt med, fikk vi fortsatt «merge»-konflikter, som kort sagt er at vi har jobbet på samme dokument og GIT gjør feil når den prøver å sette disse sammen. Årsaken til disse konfliktene er at vi bor langt unna hverandre, og har derfor jobbet mye hjemmefra uten at vi har vært i samtale. Disse konfliktene har ført til noen set-backs og skapt litt ekstra jobb, men i det store og hele har det gått veldig bra.

Swing-komponenter

Ved å personalisere nesten alle Swing-komponentene har vi støtt på noen utfordringer. Blant annet opplevde vi at tekstfelter og tekstområder kollapset til en helt annen størrelse uten

noen åpenbar grunn til å gjøre det. Med JTabbedPane opplevde vi også at faner noen gang ikke ble vist, og endret rekkefølge tilfelle. Noen av disse utfordringene ble så store at vi så oss nødt til å bruke komponenten rett fra Swing, i stedet for å lage en «custom» versjon.

Produktdokumentasjon

MVC

Vi bestemte oss, som tidligere nevnt i rapporten, tidlig om å bruke designmønsteret MVC i prosjektet. Dette er kunnskap vi har lest til oss på fritiden før prosjektstart og ikke hatt noen undervisning ved skolen i. Dersom MVC-arkitekturen ikke hadde vært implementert, ville alle metoder kommuniserende mellom data-behandlingsdelen og brukergrensesnitten vært liggende i brukergrensesnitt-klassene, noe som i verste fall kunne gjort brukergrensesnitt-klassene flere tusen kodelinjer lange. Det er også ingen mulighet for gjenbruk av kode dersom det brukes forskjellige paneler/views, siden alle metoder ville vært eksklusive for sin klasse. Dette gjør det også vanskelig å utbygge, endre og vedlikeholde koden, som vi i starten satte oss et mål om at det ikke skulle være.

Datastruktur (Model)

- Valg av datastruktur(hashset?)

- Datatyper

- Kunde-objekt

- Forsikrings-objekt

- Skademeldings-objekt

Kontroller

- Skriv om hver av de forskjellige typene, hva de gjør

- Oversikt over hvilken kontrollere som hører til hvilken GUI-klasse

- Jtable

- Sorting ved å trykke header
- Hendelse ved dobbeltklikk

GUI

Som tidligere nevnt i rapporten satte vi oss tidlig et mål om et enkelt brukergrensesnitt. Skissene vi kom fram til i kravspesifikasjonen ble raskt forkastet da vi innså hvor innviklet ting kom til å bli. For eksempel det å opprette en skademelding skal kun gjøres på en gitt forsikring til en gitt kunde, og da fant vi ut at det var lettere å først åpne en kunde, deretter åpne en forsikring der man finner en «Opprett skademelding»-knapp. Med dette slipper brukeren å gå igjennom flere valgfaser med forskjellige søk, og søker heller opp kunden i det første vinduet og tar det derfra.

I starten hadde vi også store ambisjoner når det kommer til vindudesign, spesielt med tanke på ikoner, farger og lignende. Videre ut i prosjektet satte vi oss fast på langt viktigere ting, og design har derfor blitt svært nedprioritert. Vi har valgt å gå for et heller fargeløst

program, som bruker «Look and Feel»-en til datamaskinen til brukeren. Selv om dette er kjedeligere enn det vi først satte oss som mål, mener vi det er et godt kompromiss mot den svært kjedelig standard «Look and Feel»-en i Java.

Siden vi ville ha vinduet så enkelt som mulig, er det første som møter brukeren ved oppstart et stort Jtable som strekker seg over hele vindusbredden. Denne tabellen blir ved start fylt opp av alle registrerte kunder. Over tabellen er et enkelt søkepanel, kun bestående av et søkefelt, en sjekkboks og en søkeknapp. I dette panelet kan brukeren søke, enten med navn eller nummer, på kunden den er på jakt etter. Tabellen under blir da oppdatert med søkeresultatene.

Søking

Vi valgte at et søkepanel skal være det første som møter brukeren ved oppstart av programmet. Dette søket er svært lett å bruke, og det er få steder det kan gå galt. Søkefeltet er kun for å søke på kunder, og brukeren kan både søke med kundenummer og med kundenavn - programmet finner selv ut hva det søkes etter. I tillegg gis det feilmelding ved et pop-up varsel til brukeren dersom det søkes uten søkeord. Ved søk byttes tabellen i underkant av søkepanelet med en tabell av søkeresultater i stedet. Ved søk på kundenummer fåes det maks være et treff, mens ved søk på kundenavn/søkeord kan det være flere treff på samme navn. I tillegg hadde vi planer for et panel for avansert søk (det kan sees ved å klikke «Søk» og deretter «Avansert søk» i menyen i toppen av programmet), men som tidligere nevnt måtte vi nedprioritere dette da vi ikke var ferdige med viktigere ting.

Custom Swing-komponenter

For å slippe å refaktorisere alle Swing-komponentene i programmet hver gang de legges til, valgte vi i stedet å bruke spesialtilpassede komponenter som arver av en Swing-komponent. I alle disse subklassene endres fonten og dens størrelse, og i subklassene som arver Jbutton endres også mellomrommet på side av teksten. Dette er gjort fordi vi følte knappene ble for små i forhold til de andre komponentene.

Serialisering og lagring

Packages

Kommentering

For å få en grundig dokumentasjon ved slutt, begynte vi tidlig i programmet å kommentere ved bruk av JavaDocs. Det virket som en smart ide og bruke JavaDocs da det kan generere en fungerende nettside med alle klassene, deres metoder og en kort forklaring. På den måten føler vi også at mange av metodene er kort, men godt nok, beskrevet og derfor ikke trenger videre utdypning i rapporten.

Mangler/nedprioritering

Vi ønsket å få med så mye som mulig av de funksjonene som var nevnt i oppgaven, men måtte mot slutten gi opp håpet om å klare alt. Vi valgte å prioritere det vi mener er de viktigste funksjonene, der iblant å kunne opprette kunde, forsikring og skademelding mest. Siden vi har lagt mye vekt på et enkelt brukergrensesnitt, har vi derfor også lagt til et enkelt, raskt og lett å forstå søk på forsiden. Vi hadde lenge et mål om å få til alt, men mot slutten innså vi at det ikke var nok tid. Vi ble nødt til å prioritere og satte derfor til side avansert søk og statistikk, som vi ser på som mindre viktige funksjoner. I tillegg er disse enkle å legge til ved en senere anledning, da det er mye if-er og statiske tellevariabler.

En annen funksjon vi har valgt å nedprioritere er å lagre og å vise fram bilder ved opprettelse og framvisning av skademelding. Dette er også lett å gjøre ved en senere anledning, og vi føler det er en heller liten del av oppgaven.