UNIVERSITETET I OSLO
Institutt for informatikk

Automatisert datainnsamling og kvalitetssikring av prosess og produktdata for utvikling og bruk av feilprediksjonsmodeller for et Telecom Java Legacy System

Masteroppgave
30 studiepoeng

Andreas Georg Gjersøe

23. mai 2007

Sammendrag

En av de store utfordringene systemutviklingsorganisasjoner møter i dag er å redusere antall feil i egne softwareprodukter. Tilfredsstillende kvalitet er vanskelig å oppnå ettersom kostnadene med verifikasjonsarbeid må holdes på et nivå som gjør produktet konkurransedyktig, samt at produktet skal nå markedet til rett tid. En teknikk som kan benyttes for å oppnå bedre kvalitet er å fokusere testingen på utvalgte deler av systemet på grunnlag av feilprediksjonsmodeller. QA personellet kan dermed bruke sine begrensede ressurser på en mer effektiv måte ved å foreta utvidet testing på de mest kritiske delene av systemet.

Denne oppgaven tar utgangspunkt i utvikling og anvendelse av feilprediksjonsmodeller i forbindelse med et omfattende Telecom Java legacysystem. For å kunne bygge og oppdatere feilprediksjonsmodeller for dette systemet er det nødvendig å trekke ut store mengder data fra et konfigurasjonsstyringssystem. Oppgaven beskriver et arbeid som er utført for å automatisere og kvalitetssikre datainnsamlingsprosessen, samt at det presenteres en løsning som sørger for automatisert preparering av datasett for bygging og anvendelse av feilprediksjonsmodellene.

Arbeidet som er gjort har hatt to hovedformål. Som et første punkt har det vært viktig å få til automatiske kvalitetskontroller av innsamlede data for å sikre et optimalt datagrunnlag, noe som igjen kan øke muligheten for å lage mest mulig presise feilprediksjonsmodeller. Det andre formålet har vært å legge til rette for utvidet anvendelsesområde for modellene, ved at utviklerne daglig skal kunne benytte oppdaterte prediksjoner til bruk for fokuserte kvalitetsforbedringstiltak som for eksempel restrukturering, inspeksjoner og fokusert enhetstesting av klasser med høy feilsannsynlighet. Dette krever en automatisert og effektiv datainnsamlingsprosess, kvalitetssikring, modellbygging og formidling av prediksjonsresultater til utviklerne som er ment å kunne dra nytte av dem.

Løsningen som er utviklet for å møte disse kravene er et Pythonscript som blant annet benytter MySQL Server som en ”back end” løsning, og et grafisk brukergrensesnitt som ”front end” løsning. Løsningen er ikke ment som å være en helautomatisert prosess, men snarere en modulbasert prosess som enkelt kan utføres.

Verktøyet er foreløpig ikke tatt i bruk i selve forskningsprosjektet, men en rekke testkjøringer er foretatt for å verifisere at det gir korrekte verdier. Testkjøringene viser blant annet at det preparerte datasettet er mer konsistent enn det man klarte å oppnå ved den manuelle prosessen, ettersom den automatiserte prosessen foretar en rekke konsistenssjekker og datafiltreringer som avdekker og fjerner feil i datagrunnlaget. Prosessen som tidligere tok bort i mot en uke å utføre tar nå få minutter ved hjelp av det nyutviklede verktøyet.

For at forskningsprosjektet skal ta et skritt videre i retning av et kommersielt produkt, vil det være behov for å gjøre ytterligere automatiseringsarbeid for å oppnå en helautomatisert prosess. Det vil også være avgjørende å utvikle et presentasjonsverktøy som gjør det enkelt for utviklerne å anvende prediksjonene i det daglige arbeidet.

Forord

Denne oppgaven er utført som en avsluttende del av Masterstudiet i Informatikk ved Universitetet i Oslo. Oppgaven er definert som en kort masteroppgave på 30 studiepoeng.

Jeg vil gjerne rette en stor takk til min veileder Erik Arisholm for å ha gitt meg muligheten til å delta i et av Simulas spennende forskningsprosjekter. Hans fremragende rolle som motivator og faglig ressurs har vært et betydelig og uunnværlig bidrag underveis i mitt arbeid. Jeg vil også takke mine medstudenter ved Simula for hyggelig og viktig samarbeid underveis i prosjektarbeidet.

Fornebu, 23.mai 2007

Andreas Gjersøe

Forklaring av ord og uttrykk

Legacy system	Eldre og omfattende softwaresystem som stadig utvides og vedlikeholdes
COS	Customer Order Server. Mellomvaresystem som benyttes av Telenor Mobil
XRadar	Verktøy som er utviklet av Telenor til bruk for å analysere kildekode i COS-systemet
JHawk	Verktøy som benyttes for å trekke ut strukturelle målinger (metrikker) av Javakode
MKS	Konfigurasjonsstyringssystem som benyttes hos Telenor
Sandbox	Kopi av kildekode. Koden kan dermed behandles uten å påvirke systemet.
COS Data Manager	Egenutviklet verktøy som benyttes for import og håndtering av MKS-data
COS-database	Relasjonsdatabase som fungerer som ”back end” løsning i COS Data Manager
Weka	Verktøy som inneholder en rekke maskinlæringsalgoritmer til bruk for oppgaver innen ”data mining”
RMI	Remote Method Invocation. Muliggjør fjernkommunikasjon mellom ulike Javaprogrammer.
CORBA	Common Object Request Broker Architecture. Standard som gjør det mulig for objekter å kommunisere med hverandre, selv om de ikke er en del av samme program. Objektene kan kjøre på forskjellige maskiner og/eller de kan være laget i ulike programmeringsspråk.

Innholdsfortegnelse

1 Innledning

Dette kapittelet gir en kort introduksjon til bakgrunnen for arbeidet som er gjort i denne oppgaven. Hovedmålsetninger og viktigste bidrag i arbeidet som er gjort er også beskrevet.

1.1 Utfordringer ved å sikre god kvalitet i store softwaresystemer

En av de store utfordringene systemutviklingsorganisasjoner møter i dag er å redusere antall feil i egne softwareprodukter. Dersom et produkt slippes på markedet med for mye feil, vil dette medføre misfornøyde kunder, og i neste omgang tap av kunder. På en annen side, dersom systemutviklingen innebærer unormalt høy ressursbruk innen testing og kvalitetssikring, vil kostnadene bli så store at produktet ikke vil være konkurransedyktig i et utsatt marked. Det er derfor viktig å finne en balanse der man bruker tilstrekkelig men ikke unødvendig mye ressurser for å sikre at produktene har tilfredsstillende kvalitet. Stadig flere bedrifter er opptatt av å sette i gang prosessforbedringstiltak innen testing og kvalitetssikring. I stedet for å bruke mer tid og ressurser på testing, ønsker bedriftene snarere å effektivisere arbeidet. På denne måten kan kostnader og antall feil holdes nede, samtidig som at produktet når markedet til planlagt tid.

Det er ofte slik at softwaresystemer blir vanskeligere å vedlikeholde etter hvert som funksjonaliteten utvides. Hver ”bugfix” eller implementasjon av ny funksjonalitet blir mer og mer kompleks etter hvert som systemet øker i størrelse. Hver endring/tillegg får som regel konsekvenser for øvrige deler av koden i systemet. Arbeidet med å teste og kvalitetsikre produktet blir derfor tidkrevende og kostbart. Resultatet av dette blir gjerne at nye releaser blir sluppet på markedet uten at testing er utført i tilstrekkelig grad.

Det er gjort flere omfattende forskningsarbeider der det søkes å effektivisere ressursbruken innenfor testing. Et av de større arbeidene er gjort av Simula Research Laboratory, der de i samarbeid med Telenor har utviklet feilprediksjonsmodeller for testing av et system som heter COS. Målet med dette er å hjelpe QA personellet til å bruke sine begrensede ressurser på en mer effektiv måte ved å fokusere innsatsen på deler av systemet som mest sannsynlig vil inneholde feil. Tanken er at man ved bruk av en slik prediksjonsmodell kan bruke mindre tid på å teste deler av systemet der det er liten sannsynlighet for at det vil oppstå feil, og dermed bruke desto mer tid på kritiske deler av systemet.

1.2 Utvikling og anvendelse av feilprediksjonsmodeller

For å kunne bygge og oppdatere en prediksjonsmodell for COS-systemet er det nødvendig å trekke ut store mengder data fra konfigurasjonsstyringssystemet hos Telenor. Dette er en omfattende prosess der man ønsker å sikre konsistens i innsamlede data. Dette er avgjørende for å komme frem til en mest mulig presis prediksjonsmodell, og for at det skal være enkelt å anvende modellen.

Det er også viktig at datainnsamlingsprosessen kan foregå så enkelt og raskt som mulig, ettersom man på sikt ønsker å utvide bruksområdene for prediksjonsmodellene. Det er for eksempel ønskelig at COS-utviklerne daglig kan ha oppdaterte prediksjoner, til bruk for restrukturering, inspeksjoner og lignende i tillegg til fokusert testing. En mulighet er at utviklerne kan varsles ved hjelp av et presentasjonsverktøy når det jobbes på javaklasser med høy feilrisiko. For å få til dette er det nødvendig at oppdaterte grunnlagsdata og prediksjoner er tilgjengelig til enhver tid.

Utgangspunktet for denne masteroppgaven er Simulas COS forskningsprosjekt. Oppgaven inngår som en del av prosjektet der det søkes å få til en mer strømlinjeformet innsamlings- og håndteringsprosess av MKS-data.

Hovedmålet for oppgaven er å konstruere en relasjonsdatabase som sikrer effektiv preparering av konsistente datasett basert på grunnlagsdata som samles inn fra MKS-systemet, samt en automatisert og forbedret prosess som sørger for å hente grunnlagsdataene til databasen. I dette ligger det blant annet at det bør være en robust løsning som håndterer og gir informasjon dersom inkonsistente data forekommer, slik at best mulig datagrunnlag kan sikres. I målsetningen ligger det også at det skal utvikles et verktøy som gjør det enkelt å foreta håndtering av data. Dette innebærer blant annet at løsningen skal tilby funksjoner for å generere ulike datasett som blant annet inneholder strukturelle egenskaper samt historiske endrings- og feildata for hver klasse og som benyttes videre for bygging og anvendelse av feilprediksjonsmodeller.

1.3 Bidraget i arbeidet som er gjort

Denne masteroppgaven har først og fremst vært et bidrag i retning av å få til mer automatisert bygging og anvendelse av prediksjonsmodeller i forbindelse med COS forskningsprosjektet. Arbeidet har lagt et viktig grunnlag når man i neste omgang skal utvikle en løsning der metodikken kan anvendes til ulike formål av COS-utviklerne i det daglige utviklingsarbeidet.

Tidligere masterstudent Valery Buzungu utviklet i sin masteroppgave et Perlscript som benyttes for å hente ut data fra MKS hos Telenor. Senere har scriptet blitt endret noe i forbindelse med masterarbeidet til Magnus J. Fuglerud. Dette har ført til at MKS-data ikke har blitt samlet inn på nøyaktig samme måte for alle releaser. Tidligere innsamlede MKS-data inneholdt også noe overlapp mellom datasettene for ulike releaser.

For å oppnå enklere og ryddigere datainnsamling, unngå overlapp i datasett, samt å utføre identisk datainnsamling for alle releaser er det eksisterende Perlscriptet modifisert noe, og data for alle releaser er samlet inn på nytt. Scriptet er blant annet endret slik at alle releaser kan samles inn enkelt og raskt, samt at scriptet trekker ut noe mer informasjon enn før.

For å automatisere (effektivisere) håndtering av innhentet MKS-data er det utviklet et omfattende Pythonscript som blant annet benytter MySQL Server som en ”back end” løsning, og et grafisk brukergrensesnitt som ”front end” løsning. I tillegg til funksjoner for datahåndtering, inneholder scriptet blant annet en detaljert rapporteringsfunksjon for importprosessen. Scriptet består av ca 2200 kodelinjer.

Som et siste hovedelement i arbeidet, har kvalitetskontroll av innsamlet MKS-data stått sentralt. Fokusering på å avdekke ulike typer inkonsistens i dataene som samles inn har vært viktig. Dette har blant annet ført til at det er blitt avdekket inkonsistens i Telenors MKS-system, og det er laget en rapporteringsfunksjon som kan hjelpe Telenor med å rette opp ukorrekte MKS-data. For forskningsprosjektet sin del har arbeidet ført til en bedre forståelse av årsakene til inkonsistens i dataene, noe som i sin tur kan bidra til å lage mer nøyaktige prediksjonsmodeller.

1.4 Oppgavestruktur

Den resterende delen av denne masteroppgaven er delt inn som følger:

- Kapittel 2 gir en introduksjon til forskningsprosjektet som denne oppgaven er basert på, samt noe relatert arbeid.

- Kapittel 3 beskriver en detaljert prosess for hvordan data samles og deretter blir håndtert.

- Kapittel 4 gir en teknisk beskrivelse av implementasjon som er utført i forbindelse med prosessen som er beskrevet i kapittel 3.

- Kapittel 5 konkluderer og presenterer noen ideer til hva som skal til for å ta forskningsprosjektet enda et skritt videre.

2 Utvikling og bruk av feilprediksjonsmodeller i legacy systemer

Dette kapittelet beskriver et forskningsprosjekt som er utført av Simula Research Laboratory i samarbeid med Telenor. Prosjektet har hittil arbeidet for å utvikle mest mulig presise feilprediksjonsmodeller til bruk for fokusert testing. Det er ikke gjort forsøk på å beskrive forskningsarbeidet i detalj, men snarere å sette denne masteroppgaven inn i en større kontekst og motivere masteroppgaven på bakgrunn av arbeidet som er gjort i forskningsprosjektets ulike faser.

2.1 COS forskningsprosjektet

COS (Customer Order Server) er et mellomvaresystem som utvikles og vedlikeholdes av Telenor. Systemet server mer enn 40 klientsystemer innen mobildivisjonen i Telenor, og er utviklet gradvis gjennom 22 store releaser i løpet av de siste 8 årene. Mellom 30 og 60 utviklere har til enhver tid vært involvert i utviklingen av COS-systemet, der hovedfunksjonaliteten i systemet består av rundt 2600 javaklasser som til sammen utgjør omlag 150 000 kildekodelinjer (SLOC).

Det store og omfattende COS-systemet har hele tiden vokst i størrelse og kompleksitet, samt at kontinuerlig vedlikehold foregår. QA personellet opplever hele tiden at det er for lite tid og ressurser til å utføre verifikasjon av systemet på en tilfredsstillende måte.

Simula ønsket i samarbeid med Telenor å komme frem til en teknikk der det er mulig å fokusere testingen på de mest kritiske delene av systemet. Simula har dermed arbeidet med å utvikle best mulig feilprediksjonsmodeller som baserer seg på historiske data om feilrettinger, endringer og strukturelle egenskaper ved javaklasser, og som kan anvendes som et hjelpemiddel for å fokusere testarbeidet riktig. Dette arbeidet har nå pågått siden høsten 2005, og er heretter betegnet som COS forskningsprosjektet.

COS forskningsprosjektet har gått igjennom flere faser for å komme frem til der prosjektet står i dag. I første fase av prosjektet ble feilprediksjonsmodellene bygget ut fra en hypotese om at en klasses feiltendens i all hovedsak er avhengig av strukturelle egenskaper (f.eks. kobling), antall ganger klassen er blitt endret for å komme frem til en gitt release, samt kodekvalitet (f.eks. kodestil og grad av redundans i kode). Ved hjelp av logistisk regresjon ble feilprediksjonsmodellene bygget på grunnlag av data som ble samlet inn fra XRadar [8]. Denne fasen av prosjektet kan betraktes som et pilotprosjekt, der resultatet indikerte at det var et godt potensial for å lage gode og kostnadseffektive feilprediksjonsmodeller. Arbeidet er beskrevet ytterligere i [2]. En svakhet som ble identifisert i forbindelse med arbeidet som er beskrevet i [2] var at grunnlagsdataene som ble samlet inn fra XRadar var for upresise.

Forskningsprosjektet gikk dermed videre til fase 2, der det blant annet ble testet ut og gjort sammenlikninger mellom ulike modelleringsteknikker for å bygge best mulig feilprediksjonsmodeller. Algoritmene som ble vurdert opp mot hverandre har sitt utspring i områder som ”data mining”, maskinlæring og neurale nettverk [5]. En av de mest relevante kategoriene innenfor maskinlæring fokuserer på bruk av beslutningstrær, der et datasett blir kategorisert rekursivt. Den mest anerkjente algoritmen innenfor dette området er C4.5 [6]. Bruk av denne algoritmen viste seg å gi spesielt god nøyaktighet og forventet kostnadseffektivitet for anvendelser mot fokusert testing i COS-systemet.

Hypotesen som lå til grunn for utvikling av modellene på dette stadiet var nokså lik som i fase 1, men en vesentlig forskjell var at datagrunnlaget som ble brukt for å bygge modellene nå var atskillig større enn tidligere ved at dataene ble hentet direkte fra konfigurasjonsstyringssystemet MKS [7] hos Telenor framfor via XRadar (som kun hadde relativt grovkornede data fra et svært begrenset sett av releaser). Kodeparseren JHawk [9] ble benyttet for å trekke ut strukturelle egenskaper for hver enkelt javaklasse, som deretter ble koblet sammen med historiske feil- og endringsdata fra MKS. Evaluering av de ulike modellene som ble sammenliknet indikerte blant annet at en feilprediksjonsmodell utviklet på grunnlag av C4.5 algoritmen, kan identifisere 60 % av feilene i systemet innenfor 10 % av det total systemet. Arbeidet som er gjort i fase 2 er beskrevet nærmere i [1].

Fase 3 i COS forskningsprosjektet innebærer en omfattende evaluering av den praktiske nytten ved å anvende feilprediksjonsmodellene. Evalueringen foregikk ved at en feilprediksjonsmodell ble brukt til å rangere klasser i neste release på grunnlag av feiltendensen som modellen viste for hver enkelt klasse. De klassene som viste høyest feiltendens ble tildelt ekstra ressurser for testing, og ved å gjennomføre fokusert testing på denne delen av koden ble det identifisert og rettet en relativt stor andel feil. Tiden som ble brukt til ekstra testing på disse klassene ble registrert, og deretter ble det gjennomført intervjuer av utvalgte COS-utviklere for å samle inn ekspertestimater om potensiell besparelse ved å utføre den fokuserte testingen. Evalueringen indikerte at det ble oppnådd en besparelse på 91.7 timer ved at feilene ble funnet og rettet på det gitte tidspunktet fremfor i senere faser. Antall timer brukt for å gjennomføre testingen var 49.5 timer, noe som anslagsvis vil tilsvare en kostnadsreduksjon på 50 %. Denne delen av prosjektet er beskrevet i detalj i et masterarbeid utført av Magnus J. Fuglerud.

Hittil i forskningsprosjektet har det stort sett blitt fokusert på å benytte prediksjonsmodellene til fokusert testing av utvalgte klasser mot slutten av en release, hvor datainnsamling, bygging og anvendelse av prediksjonsmodellen har vært en manuell prosess som har krevd en betydelig innsats fra forskerne som har vært involvert. Videre i forskningsprosjektet ønsker man å øke omfang på bruksområder, for eksempel ved at utviklerne *daglig* kan ha oppdaterte prediksjoner, til bruk for fokuserte kvalitetforbedringsaktiviteter som for eksempel restrukturering, inspeksjoner, og enhetstesting av de klasser som til enhver tid blir endret. Fase 4 av COS forskningsprosjektet, som denne masteroppgaven er en del av, innebærer derfor et behov for å automatisere og kvalitetsikre datainnsamlingsprosessen. Dette vil være viktig for å kunne lage mest mulig nøyaktige prediksjonsmodeller i fremtiden, samt at det er nødvendig for å kunne utvide anvendelsesområdet for prediksjonsmodellene.

2.2 Behov for automatisering av datainnsamlingsprosessen

Det finnes mye arbeid som beskriver utvikling av feilprediksjonsmodeller til bruk for fokusert testing, som blant er annet beskrevet videre i [1], men relativt lite arbeid som har forsøkt å anvende slike modeller i praksis – med de implikasjoner det har i forhold til krav til automatisering av datainnsamlingsprosess, kvalitetssikring, modellbygging og formidling av prediksjonsresultater til utviklerne som er ment å kunne ha nytte av dem.

Et unntak er [4], som beskriver et forskningsprosjekt med den hensikt å utvikle et prediksjonsverktøy som sier noe om risikoen for at det vil oppstå feil som følge av en endring gjort i et softwaresystem. Dersom man ønsker å gjøre en endring, og prediksjonsverktøyet viser at det er stor sannsynlighet for at det vil oppstå nye feil som følge av en slik endring, kan man f. eks allokere mer ressurser til testing, eller man velger kanskje å utsette en releasedato.

Hensikten er at man skal vite hvilke konsekvenser det får dersom man gjør en gitt endring i systemet, og at man da kan håndtere dette på en best mulig måte.

Målet for dette forskningsprosjektet var å utvikle et prediksjonsverktøy som kunne være et nyttig hjelpemiddel i kommersielle softwareprosjekter. For å teste ut om dette var mulig ble det utviklet en prediksjonsmodell for et stort telefonsvitsjesystem (5ESS). Modellen ble utviklet med en hypotese om at egenskapene til en softwareendring (som størrelse, varighet, spredning og type) har betydelig påvirkning på sannsynlighetsgraden for at det vil oppstå nye feil som følge av endringen. Med endring mener man i denne sammenheng ”bugfix” eller implementasjon av ny funksjonalitet. I tillegg til endringens egenskaper mente man at programmererens erfaring har mye å si for risikoen ved å utføre en gitt endring.

For å gjøre prediksjonsverktøyet mer effektivt, ble det benyttet såkalte ”risk-flags” som fungerer som ulike indikatorer. Disse gir melding til bruker om hvorfor det er stor sannsynlighet for at feil vil oppstå ved en gitt endring. Eksempler kan være risk-flag med verdien ”many sub systems touched” eller ”low developer experience”. For å komme frem til en modell som gir gode prediksjoner viste det seg at det er avgjørende at det finnes historiske data som gjør det mulig å definere endringer som har høy risiko og hvilke som har lav risiko.

Metodikken brukes i dag som et verktøy i 5ESS softwareprosjektet, og har påvist gode resultater. Alle data som verktøyet benytter seg av blir automatisk hentet ut av konfigurasjonsstyringssystemet brukt i 5ESS softwareprosjektet. Det hevdes også at verktøyet kan benyttes i andre systemprosjekter, dersom nødvendige historiske data er tilgjengelig.

Fellestrekk for 5ESS og COS forskningsprosjektene er blant annet at det benyttes historiske data for å komme frem til prediksjonsmodellene, og at dataene hentes ut fra eksisterende konfigurasjonsstyringssystemer.

Det er også en del klare forskjeller. Teknikken som benyttes i COS forskningsprosjektet fokuserer på klassers feilsannsynlighet for å kunne fokusere testingen, mens 5ESS-prosjektet fokuserer på risikoen ved å utføre en gitt endring. I [4] poengteres det også at det er viktig å være klar over hvorfor en gitt endring har stor risiko, noe som ikke i like stor grad er tatt stilling til i COS-prosjektet.

En annen viktig forskjell er at COS-prosjektet er unikt på den måten at det er det første prosjektet som tar stilling til bruk av en feilprediksjonsmodell for fokusert testing i et legacysystem. Dessuten er det utført en ny type kost-nytte analyse som sier noe om hvilke ressursbesparelser man kan vente seg ved å bruke prediksjonsmodellen

Arbeidet i [4] er kommet noe lenger i retning av et kommersielt produkt enn det som er presentert i [1]. Her har man lykkes i å strømlinjeforme bruk av verktøyet ved at data hentes automatisk fra konfigurasjonsstyringssystemet. Det er også lagt vekt på å gjøre verktøyet mer effektivt og forståelig for en bruker. Dette er blant annet gjort ved å bruke såkalte ”risk-flags”. Verktøyet er dessuten gjort webbasert slik at informasjonen er enkelt tilgjengelig.

COS forskningsprosjektet har fremdeles et stykke igjen for å kunne oppnå en strømlinjeformet datainnsamlingsprosess. Dessuten har bygging av prediksjonsmodellene til nå vært en meget tidkrevende og omstendelig prosess. Datainnsamlingsprosessen har foregått ved at et Perlscript trekker ut nødvendig data fra konfigurasjonsstyringssystemet hos Telenor. Dataene må deretter organiseres og formateres før de importeres i Excel. Videre er det behov for ytterligere endringer av dataene. For å lette jobben med dette er det laget en håndbok som

beskriver alle endringer som skal foretas. Excelfiler blir deretter importert til statistikkverktøyet SAS som brukes til å generere et resultatdatasett som igjen danner grunnlag for prediksjonsmodellen.

Hele denne prosessen er som nevnt omfattende og tidkrevende, og inneholder lite automatikk. Når COS-utviklerne er ferdig med implementasjon av en ny release, må prosessen gjøres manuelt på kort tid. Hvis COS-utviklerne skal ha nytte av prediksjonsmodellen er det viktig at prediksjonsmodellen er tilgjengelig før testfasen starter (som typisk er maks 1 uke etter at implementasjon er avsluttet). Det har vist seg at dette er vanskelig å få til i praksis, ettersom bygging/oppdatering av prediksjonsmodellen er tidkrevende. Et annet problem er at tidsnød kan gå ut over kvaliteten til prediksjonsmodellen, ettersom prosessen mangler tilstrekkelige kvalitets- og konsistenssjekker på dataene som samles inn.

For at COS forskningsprosjektet skal være i stand til å ta et skritt videre i retning av et kommersielt produkt, er det nødvendig å få til en mer automatisert prosses i forhold til datainnsamling, kvalitetssikring, modellbygging og formidling av prediksjonsresultater til utviklerne som er ment å kunne ha nytte av dem. I den forbindelse er arbeidet som er presentert i denne oppgaven sentralt. Kapittel 3 presenterer en nyutviklet datainnsamlings- og håndteringsprosess som tar et steg videre i retning av å få til en strømlinjeformet prosess.

3. Innsamling og håndtering av MKS-data

Dette kapittelet beskriver hvordan MKS-data samles inn og deretter håndteres i forbindelse med bygging og anvendelse av feilprediksjonsmodeller. Kapittel 3.1 introduserer et verktøy som er kalt COS Data Manager, mens kapittel 3.2 gir en overordnet beskrivelse av hele prosessen som gjøres for å komme frem til predikerte datasett som kan hjelpe COS-utviklerne til å foreta fokuserte kvalitetsforbedringstiltak. Videre gir kapittel 3.3 en fremstilling av selve datainnsamlingprosessen, og kapittel 3.4 gir en grundig beskrivelse av filtreringer som foretas for å oppnå et mest mulig optimalt datagrunnlag. Til gir kapittel 3.5 en beskrivelse av hvordan generering av historisk datasett foregår, hvordan det anvendes, samt en beskrivelse av selve datasettet.

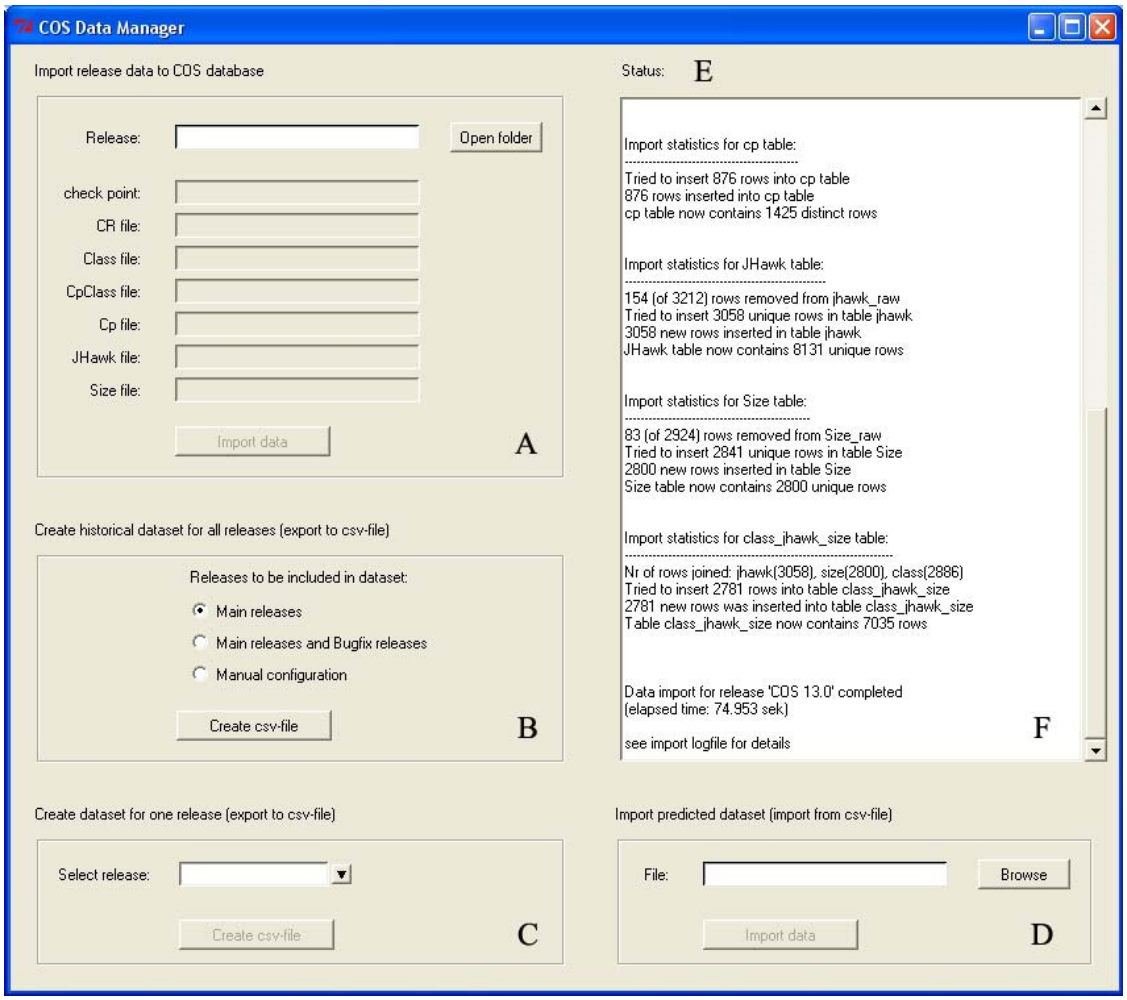
3.1 Beskrivelse av COS Data Manager

COS Data Manager er et verktøy som er utviklet for å lette arbeidet med å importere og håndtere MKS-data. Verktøyet er laget i Python og benytter blant annet en relasjonsdatabase som ”back end” løsning og et enkelt grafisk brukergrensesnitt som ”front end” løsning. Teknisk informasjon om hvordan COS Data Manager er bygget opp er beskrevet i kapittel 4. I dette kapittelet vil det fokuseres på anvendelse av verktøyet, der hver enkelt funksjon vil bli omtalt.

Det viktigste og mest omfattende COS Data Manager inneholder er en funksjon som sørger for rask import og lagring av MKS-data. Funksjonen utføres ved hjelp av felt A i Figur 1. Etter at MKS-data (rådata) er samlet inn ved hjelp av Perlscriptet, vil alle disse dataene bli liggende i en releasemappe som inneholder 6 ulike datasett (flatfiler). Denne mappen kan lastes direkte inn i COS Data Manager ved hjelp av en fildialog. Funksjonen søker igjennom innholdet i den valgte mappen for å kontrollere at alle nødvendige filer eksisterer. Dersom dette ikke er tilfelle, vil det ikke være mulig å foreta selve importfunksjonen (importknapp ”disabled”). Hvis derimot alle nødvendige filer eksisterer i den valgte mappen, vil import knappen ”enables” slik at selve importprosessen kan starte.

Importfunksjonen foretar en rekke formateringer, konsistenssjekker og filtreringer for å gjøre rådata om til grunnlagsdata, som deretter lagres i relasjonsdatabasen. Denne prosessen tar 1-2 minutter ved hjelp av importfunksjonen, noe som typisk tok flere dager å utføre manuelt tidligere. Funksjonen er laget slik at en release kan importeres flere ganger uten at duplikater forekommer. Hvis en gitt release allerede er lagret i databasen, og man importerer denne releasen på nytt, så vil de gamle verdiene bli skrevet over. Dette vil blant annet være nyttig på et senere tidspunkt når man ønsker å få til daglige oppdaterte prediksjoner. Databasen vil i denne sammenheng kun inneholde den siste oppdateringen av en release, noe som gjør at databasen ikke blir unødvendig stor.

For å kontrollere at importprosessen foregår korrekt, inneholder brukegrensesnittet et statusfelt (felt E i Figur 1), samt et resultatfelt (felt F i Figur 1) som gir informasjon om importprosessen. Det genereres også en importlogfil som i tillegg gir mer detaljert informasjon om filtreringer som gjøres.



Figur 1: Grafisk brukergrensesnitt i COS Data Manager

En annen sentral del av COS Data Manager er en funksjon som benyttes for å generere historiske datasett (csv-fil) som benyttes som grunnlag når feilprediksjonsmodellene skal bygges. Denne funksjonen betjenes ved hjelp av felt B i Figur 1. Her er det mulig å velge hvilke releaser som skal inngå i det historiske datasettet. Man kan her velge å ha med kun mainreleaser eller både main- og bugfixreleaser. Det er også mulighet for å velge et egendefinert utvalg av releaser, men dette må foreløpig konfigureres i selve relasjonsdatabasen. Også her vil resultatvinduet i brukergrensesnittet fungere som et hjelpemiddel for å kontrollere at prosessen foregår korrekt. Prosessen med å lage historisk datasett tok tidligere flere dager å utføre manuelt. Ved hjelp av COS Data Manager kan denne prosessen utføres i løpet av få minutter.

I tillegg til de nevnte funksjonene inneholder COS Data Manager en funksjon for å generere et datasett for siste release. Dette foregår ved hjelp av felt C i Figur 1, som sørger for å lage datasettet i form av en csv-fil. Csv-filen importeres deretter manuelt til Weka [10] der feilprediksjonsmodellen anvendes på datasettet. Datasettet får deretter lagt til predikerte verdier, før det importeres tilbake til COS databasen. Dette gjøres ved hjelp av felt D i Figur 1. På et senere tidspunkt vil det dermed være mulig å benytte de predikerte dataene til fokuserte kvalitetsforbedringstiltak som restrukturering, inspeksjoner og fokusert enhetstesting av klasser med høy feilsannsynlighet.

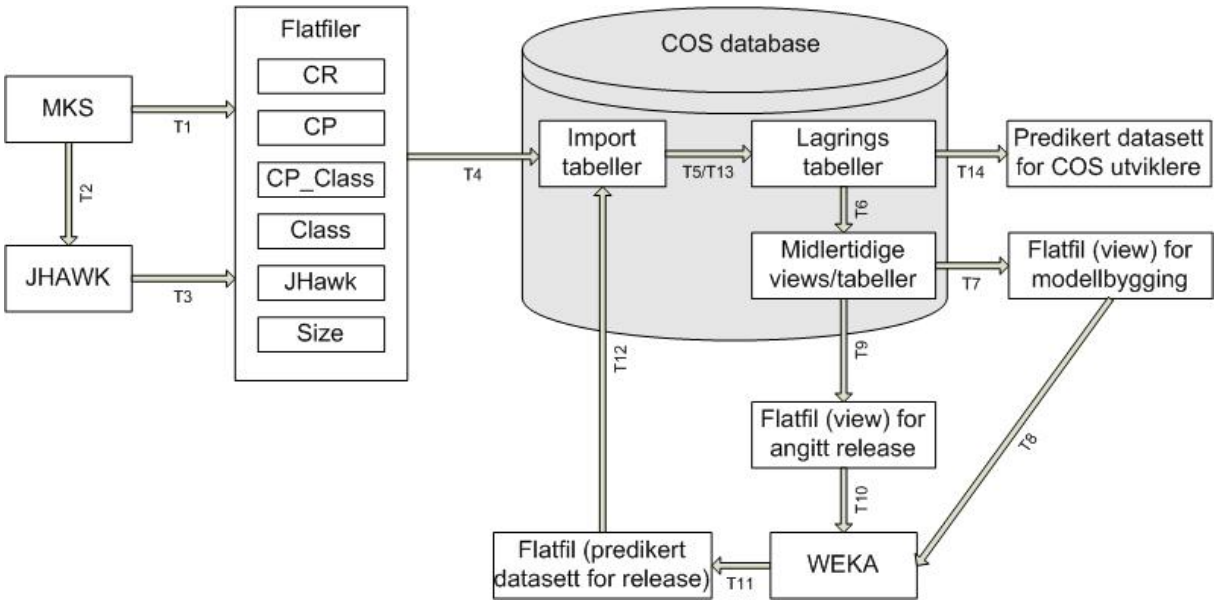
3.2 Overordnet prosess for generering av predikerte datasett

Prosessen med innsamling og håndtering av data foregår gjennom trinnene T1-T14 som er illustrert i Figur 2. T1-T3 utføres ved hjelp av et Perlscript, mens T4-T14 utføres av Pythonscriptet (med unntak av T8, T10, T11 og T14 som foreløpig utføres manuelt). Hvert enkelt trinn er beskrevet nærmere i Tabell 1.

Importtabellene sørger blant annet for rask import av data. Disse tabellene tar imot dataene slik de er presentert i flatfilene (rådata). Så snart dataene er modifisert, filtrert og overført til lagringstabellene (T5), blir dataene slettet fra importtabellene.

Lagringstabellene inneholder data for alle importerte releaser på korrekt/ønsket format. Tabellene har en rekke integritetsnøkler som sørger for konsistens, samt indekser som sørger for effektive SQL-operasjoner. Mer detaljert informasjon om dette er gitt i kapittel 4.3.

Generering av historisk view til bruk for modellbygging gjøres typisk når en release blir lagt til eller oppdatert i databasen. Det historiske viewet benyttes også som kilde når det skal genereres csv-fil for siste release (T9). Prediksjonsmodellen vil dermed anvendes på dette datasettet for å komme frem til predikerte verdier. Det predikerte datasettet blir deretter lagt til i en egen lagringstabell i databasen, slik at dataene er tilgjengelig for en fremtidig presentasjonsapplikasjon som kan benyttes av COS-utviklerne.



Figur 2: Illustrasjon av innsamling og håndtering av MKS-data

Tabell 1: Beskrivelse av ulike trinn i datahåndteringsprosessen
(trinnene utføres ikke nødvendigvis i stigende rekkefølge)

Trinn	Beskrivelse
T1-T3	For en gitt release samles datasettene CR, CP, CP_Class, Class, JHawk og Size fra MKS. For å få frem JHawk-data lages en ”sandbox” som inneholder alle aktuelle javaklasser. Disse klassene blir deretter parset av JHawk for å trekke ut en rekke strukturelle egenskaper for hver enkelt javaklasse. Size-data hentes også på grunnlag av ”sandboxen” der antall linjer og antall interne klasser for hver javaklasse beregnes. Alle de nevnte datasettene blir lagret separat i hver sin flatfil.
T4	Flatfilene for en gitt release importeres til importtabeller i COS-databasen
T5	Dataene modifiseres slik at dataene har korrekt format, samt det gjøres en rekke filtreringer for å bli kvitt uønskede data. Deretter overføres dataene til lagringstabellene i COS-databasen, og det genereres en importlogfil som inneholder detaljert informasjon om importprosessen.
T6	Dataene i lagringstabellene joines, filtreres og modifiseres for å komme frem til et historisk view for valgte releaser. Dette viewet inneholder data som benyttes for å bygge/oppdatere feilprediksjonsmodellen.
T7	Historisk view eksporteres til csv-fil.
T8	Csv-fil med historisk datasett benyttes til å bygge/oppdatere prediksjonsmodellen. Dette gjøres i et verktøy som heter Weka, og er ikke en del av pythonscriptet.
T9	Historisk view for siste release eksporteres som csv-fil (filtrerer bort øvrige releaser).
T10	Csv-fil for siste release imorteres til data mining verktøyet Weka. Denne handlingen utføres ikke av pythonscriptet.
T11	Etter at Weka har anvendt prediksjonsmodellen på datasettet for siste release, lagres det nye datasettet i en csv-fil. Dette er ikke en del av Python scriptet.
T12	Den nye csv-filen med predikert datasett importeres til en importtabell i COS-databasen.
T13	Dataene overføres fra importtabell til lagringstabell.
T14	Predikert datasett for siste release eksporteres til et presentasjonsverktøy som kan benyttes av COS-utviklere. Dette er ikke en del av denne oppgaven, men vil bli utviklet i det videre arbeidet i COS forskningsprosjektet.

3.3 Datainnsamling

Dette kapittelet gir en komplett oversikt over ulike MKS-data som samles inn til bruk for bygging og anvendelse av feilprediksjonsmodeller. Kapittel 3.3.1 og 3.3.2 gir en kort introduksjon til verktøyene MKS og JHawk ettersom disse utgjør en sentral del av datainnsamlingsprosessen, mens kapittel 3.3.3 presenterer hvordan dataene samles inn, samt at det gis en beskrivelse av dataene. Mer utfyllende informasjon om MKS, JHawk og ytterligere detaljer rundt datainnsamlingen kan sees i en masteroppgave utført av Valery Buzungu. Denne oppgaven gir en omfattende beskrivelse av arbeidet som er gjort i fase 1 og 2 av COS forskningsprosjektet.

3.3.1 MKS

MKS er et konfigurasjonsstyringssystem utviklet av selskapet MKS, som blant annet tilbyr robust håndtering av arbeidsflyt og endringer i et softwaresystem [7]. Verktøyet gir en komplett oversikt over alle endringer som er gjort i forbindelse med utviklings- og vedlikeholdsarbeid, samt støtte for samarbeid mellom ulike utviklingsgrupper i en organisasjon. MKS består av hovedkomponentene Integrity Manager og Source Integrity. Kort fortalt tilbyr Integrity Manager funksjoner som hjelper utviklerne å ha kontroll over all informasjon relatert til hver enkelt endring i systemet, mens Source Integrity fungerer som et verktøy for versjonskontroll.

I forbindelse med COS forskningsprosjektet benyttes MKS som kilde når rådata skal samles inn. Data samles både fra Integrity Manager og Source Integrity som senere kobles sammen med strukturelle egenskaper for hver enkelt klasse (generert av JHawk). Dette vil dermed utgjøre datagrunnlaget (rådata) som feilprediksjonsmodellene bygges ut ifra.

3.3.2 JHawk

JHawk er et verktøy utviklet av Virtual Machinery som gir mulighet for å generere sentrale metrikker i javakode [9]. Verktøyet er ment som et hjelpemiddel for å forbedre ytelse, gjenbruk, vedlikehold og kvalitet av et javasystem. Generering av metrikkene foregår på pakke-, klasse- og metodenivå og tilgjengeliggjøres som .xml, .html og .csv filtyper. Analyse av et system kan enten gjøres ved hjelp av et grafisk brukergrensesnitt, eller fra en kommandolinje.

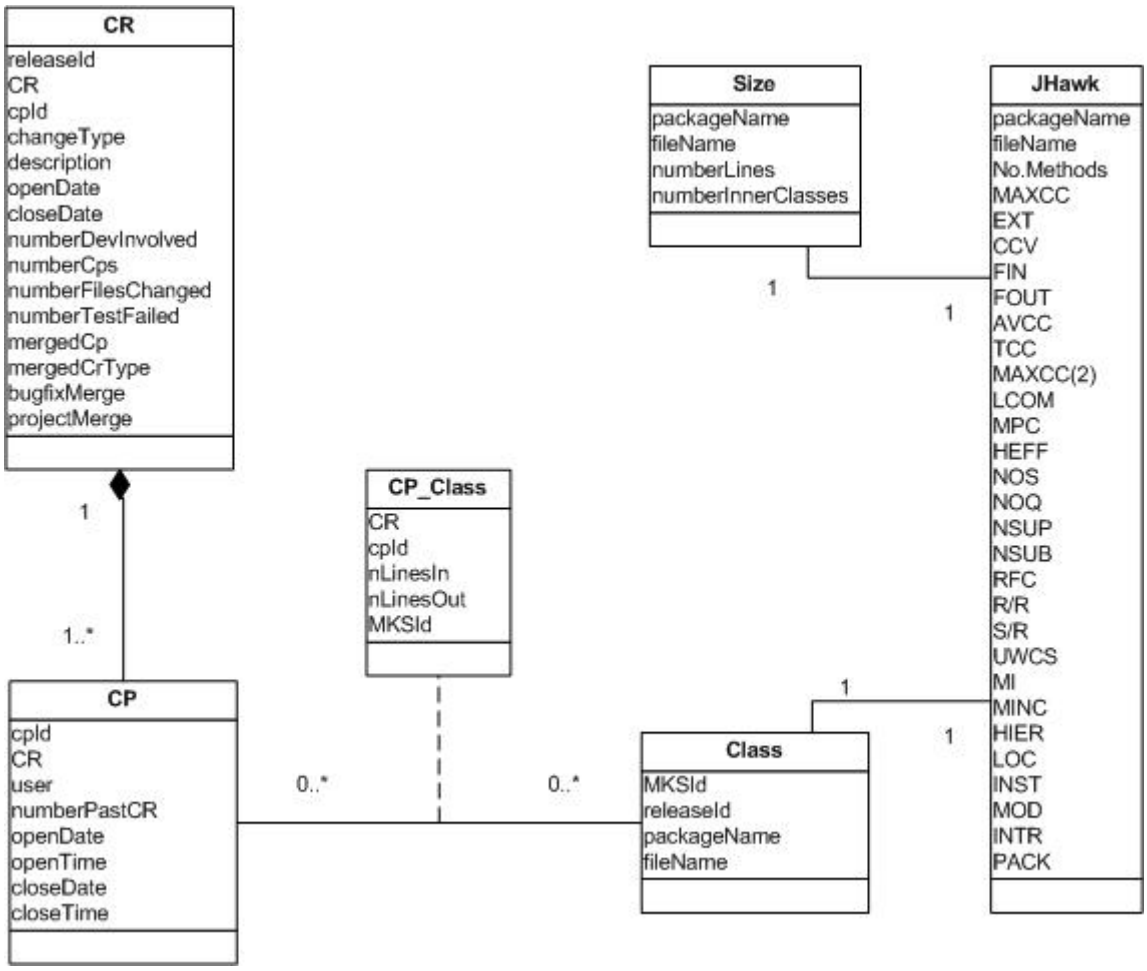
I forbindelse med COS forskningsprosjektet benyttes JHawk for å hente inn strukturelle egenskaper (for eksempel kohesjon og kobling) til javaklasser som inngår i alle COS-releaser. Disse dataene kobles mot endringsdata fra MKS på et senere tidspunkt. Kjøring av JHawk er lagt inn som en del av et Perlscript. Dette scriptet sørger for å samle alle nødvendige data for å kunne bygge og anvende feilprediksjonsmodeller.

3.3.3 Innsamling av MKS-data

Datainnsamlingsprosessen foregår ved at et Perlscript trekker ut data fra MKS på filnivå. I denne sammenheng tilsvarer hver enkelt fil en ”Java public class”, der hver endring gjort på en klasse er representert ved en CR (Change Request). Videre er en CR relatert til en gitt release og har en ”ChangeType” som definerer om endringen er en kritisk/ikke-kritisk feilretting, en liten/middels/stor funksjonalitetsendring, eller en vedlikeholdsendring. En utvikler kan jobbe på en CR gjennom en logisk enhet som kalles CP (Change Package). Her kan utviklerne sjekke ut og inn klasser som er relatert til CR’en som det arbeides med.

Videre beregnes utviklererfaring ved å se på hvor mange CR'er hver enkelt utvikler har jobbet på før den aktuelle CP'en er åpnet. For hver klasse som er endret samles data om hvor mange linjer som er lagt til og hvor mange linjer som er trukket fra. I tillegg til endrede klasser samles også et eget datasett som består av alle javaklasser som inngår i den aktuelle releasen det samles data for.

Hver klasse har en unik identifikator som kalles MKSId. Denne identifikatoren gjør det mulig å spore endringshistorikk til en klasse selv om klassen har skiftet navn, eller blitt flyttet til en annen pakke i systemet. JHawk benyttes for å trekke ut strukturelle egenskaper for hver enkelt klasse i systemet. Til slutt beregnes antall linjer og antall interne klasser i hver enkelt klasse. Dataene lagres i 6 ulike flatfiler. Figur 3 viser datamodell for datainnsamlingen, mens Tabell 2 gir en komplett oversikt og beskrivelse av attributter i hver enkelt flatfil. Hvis noe av innholdet i disse filene bryter med datamodellen presentert i Figur 3 vil dette sees på som inkonsistente data, og vil følgelig bli fjernet og rapportert underveis i importprosessen før dataene lagres i COS-databasen.



Figur 3: Datamodell for innsamling av MKS-data

Tabell 2: Beskrivelse av flatfiler med tilhørende attributter

fil	Attributt	Beskrivelse
CR	releaseId	Angir hvilken release CR'en hører til
	cpId	Change Package Id
	CR	Change Request Id
	openDate	Data for når CR ble åpnet
	closeDate	Dato for når CR ble avsluttet
	numberCps	Antall Change Packages som CR inneholder
	numberFilesChanged	Antall filer som ble endret i CR
	numberDevInvolved	Antall utviklere som var involvert i CR
	numberTestFailed	Antall systemtestfeil i forbindelse med CR
	changedType	Angir hvilken kategori CR'en tilhører
	description	Beskrivelse av CR
	mergedCp	Angir kilde-cpId dersom en endring er overført fra en "bugfix branch" eller "project branch".
	mergedCrType	Angir hvilken release endringen kommer fra dersom endring er overført fra en "bugfix branch" eller "project branch".
	bugfixMerge	Angir om en endring er overført fra en "bugfix branch"
	projectMerge	Angir om en endring er overført fra en "bugfix branch"
CP	cpId	Change Package Id
	CR	Angir hvilken CR CP'en hører til
	user	Angir hvilken utvikler som har jobbet på CP'en
	numberPastCr	Angir hvor mange CR'er utvikleren har jobbet på før denne CP'en
	openDate	Startdato for CP
	openTime	Starttidspunkt for CP
	closeDate	Avslutningsdato for CP
	closeTime	Avslutningstidspunkt for CP
CP_Class	MKSId	Unik identifikator for en klasse
	cpId	Angir CP der endringen er gjort
	CR	Angir hviken CR endringen/CP'en hører til
	nLinesOut	Angir hvor mange linjer som er lagt til i klassen i forbindelse med endringen.
	nLinesIn	Angir hvor mange linjer som er slettet i klassen i forbindelse med endringen.
Class	releaseId	Angir hvilken release klassen hører til
	MKSId	Unik identifikator for en klasse
	packageName	Angir hvilken pakke klassen ligger i
	fileName	Navn på klasse
	project	Angir hvilket prosjekt klassen hører til (brukes kun til rapportering)
	revision	Angir revisjon av klassen (brukes kun til rapportering)

JHawk	packageName	Angir hvilken pakke klassen ligger i
	fileName	Navn på klasse
	No.Methods|NOQ|NOC	Antall [implementerte | spørrings | command] metoder i en klasse
	LCOM	Mangel på kohesjon i klassen
	TCC|MAXCC|AVCC	[Total | Maks | gjennomsnittlig] syklisk kompleksitet i klasse
	NOS | UWCS	Klassens størrelse i [antall javauttrykk | antall attributter + antall metoder]
	HEFF	Halstead-arbeid for klasse
	EXT | LOC	Antall [eksterne | lokale] metoder som kalles av klasse
	HIER	Antall metoder som kalles som er i klassehierarkiet til denne klassen
	INST	Antall instansvariabler
	MOD	Antall "modifiers" for deklarering av denne klasse
	INTR	Antall implementerte grensesnitt for klasse
	PACK	Antall importerte pakker
	RFC	Total respons for klasse
	MPC	Meldingsflytkobling
	FIN	Antall unike metoder som kaller metodene i klassen
	FOUT	Antall unike ikke-arv relaterte klasser som klassen er avhengig av
	R/R | S/R	[Gjenbruks | spesialiserings] forhold for en klasse
	NSUP | NSUB	Antall [sub | super] klasser
	MI | MINC	Gjenbruksindeks for klasse [med | uten] kommentarer
Size	releaseId	Angir hvilken release klassen hører
	packageName	Angir hvilken pakke klassen ligger i
	fileName	Navn på klasse
	numberLines	Angir antall linjer klassen består av
	numberInnerClasses	Angir antall interne klasser i klassen

- CR-filen inneholder informasjon om alle CR'er som tilhører den aktuelle release det samles data for. Alle CP'er som tilhører hver enkelt CR er også listet opp her.

- CP-filen inneholder alle CP'er som hører til en gitt release, samt informasjon om utviklere som har utført arbeidet.

- CP_Class-filen inneholder data om alle klasser det er gjort endringer på for en gitt release.

- Class-filen inneholder alle javaklasser som inngår i en release (ikke bare klasser det er gjort endringer på)

- JHawk-filen lages på grunnlag av en sandbox. Perlscriptet sørger for å lage sandboxen som deretter blir parset av JHawk. JHawk trekker ut en rekke strukturelle egenskaper for hver klasse som lagres i JHawk-filen.

- Size-filen lages også på grunnlag av sandboxen. Det beregnes antall kodelinjer og antall interne klasser i alle klasser som er med i sandboxen.

3.4 Filtrering av MKS-data

Datasettene som samles inn fra MKS inneholder en del klasser som må fjernes for å få et best mulig datagrunnlag til bruk for bygging av feilprediksjonsmodeller. Målet er at databasen i størst mulig grad kun skal inneholde data som benyttes i forbindelse med prediksjonsmodellene. For å oppnå dette foretas det filtreringer både underveis i importprosessen, og når det genereres historisk view for utvalgte releaser.

For å gi en oversikt over data som importeres til databasen genereres det en importlogfil for hver release som importeres. Importlogfilen inneholder blant annet detaljert informasjon om data som blir fjernet. Dette er først og fremst klasser som ikke er unike på MKSId eller packageName + fileName. Importlogfilen kan blant annet benyttes til å rapportere tilfeller av MKS-inkonsistens til Telenor, noe som allerede har blitt gjort. Et annet formål er å se størrelsesorden på ulike typer data som blir fjernet. Hvis det for eksempel fjernes mye viktig data på grunn av at data ikke kan identifiseres, kan man vurdere om innsamlingsprosessen bør modifiseres slik at disse dataene kan inngå i datagrunnlaget for modellbygging i fremtiden. Importlogfilen er ment som å være et hjelpemiddel til å forstå datainnsamlingsprosessen og de filtreringer som forstas.

3.4.1 Filtrering av CP-data

Det finnes noen CP'er som ikke hører til noen CR. Dette er CP'er som finnes på en "project branch" og skal fjernes. Tilfellene kommer til syne ved at tupler i CP-filen ikke har noe matchende tuppel i CR-tabellen.

Noen CP'er gjør ingen endringer på klasser. Her vil det være tupler i CP-filen som ikke har matchende tuppel i CP_class-filen.

Begge disse tilfellene fjernes når CP-tabellen joines med CR- og CP_Class-tabellen, noe som skjer når historisk view genereres for utvalgte releaser.

3.4.2 Filtrering av CP_Class-data

CP_Class-filen inneholder noen klasser (MKSId'er) som ikke finnes i Class-filen. Dette betyr at det er hentet enringsinformasjon om en klasse som ikke er en del av kildekoden man ønsker å se på. Et eksempel på dette er at det kommer med endringer på f.eks xml-filer i CP_Class-filen, mens Class-filen bare inneholder java-filer. For å håndtere dette, blir bare klasser som finnes i Class-filen lagret i CP_Class tabellen i databasen.

For noen releaser forekommer også duplikater i CP_Class-filen. Duplikatene blir fjernet slik at bare unike tupler står igjen.

3.4.3 Filtrering av Class-data

Class-filen skal inneholde alle aktuelle javaklasser som inngår i en gitt release. Det er også viktig at hver enkelt fil kan identifiseres både ved hjelp av MKSId for kobling mot CP_Class-tabellen, og ved kombinasjonen packageName + fileName for kobling mot JHawk-tabellen (for en gitt release).

Det har vist seg at dette ikke alltid er tilfelle. En av årsakene er at en javafil kan ha blitt flyttet til et annet prosjekt i MKS, uten å bli slettet på opprinnelig plassering. Da får man en klasse som ligger på to forskjellige prosjekter men med samme MKSId. Dette er noe som forekommer når MKS-brukerne hos Telenor gjør feil.

En annen årsak er at enkelte javafiler blir benyttet som utgangspunkt når flere nye javafiler opprettes. Det forekommer at disse nye javafilene feilaktig arver MKSId fra orginalfilen dersom MKS-brukeren gjør feil. Dermed får man flere javafiler med ulikt filnavn, men med identisk MKSId.

Når det gjelder problematikk rundt ikke-unik kombinasjon packageName + fileName så forekommer det at en klasse har flere konfigurasjoner. Er eksempel på dette er en klasse som benyttes både for RMI [11] og CORBA [12]. I MKS vil dette forekomme ved at to nokså identiske klasser har samme fileName + packageName, men at de ligger på ulike prosjekter. Ved et slikt tilfelle vet man ikke hvilken av disse to klassene som skal kobles mot matchende tuppel i JHawk-tabellen.

Alle tilfeller av klasser der MKSId, eller packageName + fileName ikke er unik, blir fjernet før class-data lagres i Class-tabellen i databasen. Dette gjøres for ikke å risikere at disse klassene skal kobles med feil informasjon i jhawk- og size-tabellen, noe som ville svekket kvaliteten i datagrunnlaget.

I fremtiden vil det være mulig å benytte klasser med ulik konfigurasjon som nevnt i eksempelet over dersom JHawk modifiseres slik at fullstendig filebane benyttes, dvs. at alle klasser som parses av JHawk har samme rot i filbanen som rapporteres. Dermed ville en slik klasse fremdeles hatt identisk fileName, men ulik packageName, noe som ville gjort det mulig å koble disse klassene mot korrekte verdier i JHawk- og Size-tabellen. Idag fungerer JHawk slik at kun pakken som klassen ligger i blir rapportert, men det foreligger ingen informasjon om hvilken prosjekt denne pakken inngår i.

3.4.4 Filtrering av JHawk- og Size-data

I likhet med Class-filen, inneholder JHawk-filen endel tilfeller der packageName + fileName ikke er unik for en klasse. Disse blir fjernet også i JHawk tabellen. I tillegg er det slik at JHawk trekker ut strukturelle egenskaper også for interne klasser i en hovedklasse. Disse klassene identifiseres bare med navnet på selve den interne klassen, der navnet på hovedklassen ikke er tatt med. Ettersom det finnes mange interne klasser med identisk navn, men som ligger i forskjellige hovedklasser, vil disse ikke kunne skilles. Et annet problem er at feil/endringsdata for interne klasser ikke blir rapportert ved hjelp av navnet på den interne klassen i MKS, men kun ved hjelp av navnet på hovedklassen. Alle interne klasser blir dermed fjernet fra JHawk-filen, ettersom det foreløpig ikke er mulig å koble informasjon om disse klassene på en korrekt måte.

Også Size-filen inneholder klasser som ikke er unike på packageName + fileName (for en gitt release). Disse blir fjernet før dataene lagres i Size-tabellen i databasen.

3.4.5 Filtrering av utvalgte klasser

Når separate filtreringer er foretatt for class-, jhawk-, og size-data, joines disse datasettene for så å bli lagret i en tabell i databasen som heter class_jhawk_size. Tidligere har COS forskningsprosjektet i samarbeid med Telenor kommet frem til at noen typer javafiler som inngår i det joinede datasettet ikke er interessant å ha med som grunnlagsdata. Dette gjelder klasser som inneholder ”TestCase” eller ”ActCommand” i fileName-verdien, og det gjelder klasser som inneholder ”com.telenor.cos.act” eller ”com.telenor.cos.test” i packageName-verdien. Disse klassene blir dermed filtrert ut før resterende klasser importeres til class_jhawk_size-tabellen. Denne tabellen vil dermed kun inneholde klasser som skal inngå i grunnlagsdata for bygging av feilprediksjonsmodellene.

3.5 Generering og anvendelse av historisk datasett

Generering av historisk datasett (view) utgjør en sentral del av COS Data Manager. Prosessen starter med å joine lagringstabellene CR, CP, CP_Class og Class_JHawk_Size (se Figur 4 kap 4.3.2). For å beregne antall arbeidsdager for hver CR benyttes holidays-tabellen, mens historiske verdier for klasser kommer frem ved blant annet å benytte tabellene release_main, release_bug_main eller release_man, avhengig av hvilke releaser som skal inngå i det historiske viewet. Prosessen skjer gjennom en rekke trinn der både midlertidige views og tabeller benyttes.

Når prosessen er fullført, innholdet det historiske viewet alle klasser som inngår i hver utvalgt release. For en gitt release er det blant annet beregnet endrings- og feilrettingsdata for klasser i 3 releaser bakover i tid, samt 1 release frem i tid, så fremt dette er mulig. En beskrivelse av alle attributter som inngår i det historiske viewet er presentert i Tabell 3.

Det historiske viewet eksporteres som csv-fil til Weka der det benyttes for å bygge og oppdatere feilprediksjonsmodeller.

Klasser i den siste releasen (som inngår i det historiske viewet) vil naturlig nok ikke ha verdier for én release frem i tid (np1). Den siste releasen trekkes ut av viewet og eksporteres som csv-fil til Weka, der en feilprediksjonsmodell vil bli anvendt for å gi klassene predikerte verdier for om klassen vil inneholde feil eller ikke i neste release. Det predikerte datasettet blir deretter importert til COS-databasen og lagres i tabellen releasepridicted (se fig 4 kap 4.3.2). Denne tabellen består av de samme attributtene som finnes i det historiske viewet, bortsett fra at tabellen releasepredicted inneholder et ekstra attributt som heter Errorpridicted. Etter at feilprediksjonsmodellen er anvendt på datasettet for siste release, vil dette attributtet angi for hver enkelt klasse om klassen vil inneholde feil i neste release eller ikke. Tabellen releasepredicted vil dermed være grunnlaget for et fremtidig presentasjonsverktøy som kan hjelpe COS utviklerne til å foreta fokuserte kvalitetsforbedringstiltak

Tabell 3: Beskrivelse av historisk view med tilhørende attributter

Attributt	Beskrivelse
releaseId	Angir hvilken release klassen hører til
MKSId	Unik identifikator for klasse
packageName	Angir hvilken pakke klassen ligger i
fileName	Navn på klasse
numberCRs	Antall CR'er der klassen har vært endret
crWeekdayExclHoly	Antall arbeidsdager som er brukt på en CR
numberFilesChanged	Totalt antall klasser som er endret i forbindelse med alle CR'er som klassen har vært endret i
numberDevInvolved	Totalt antall utviklere som har jobbet på alle CR'er som klassen har vært endret i
numberTestFailed	Totalt antall ganger systemtestfeil har forekommet i alle CR'er som klassen har vært endret i
numberCps	Totalt antall CP'er som inngår i alle CR'er som klassen har vært endret i
numberCPsForClass	Antall CP'er som har endret klassen
numberPastCR	Total utviklererfaring før endring av klassen startet
nLinesIn	Antall linjer som er lagt til for denne klassen (innebærer alle CP'er som har endret klassen)
nLinesOut	Antall linjer som er slettet i denne klassen (innebærer alle CP'er som har endret klassen)

For CR'er av typen X = {CLL, CKL, CFL, M, CE, E}:	
<X>_CR	Tilsvarende som numberCRs men innebærer kun CR'er av typen X
<X>_CPs	Tilsvarende som numberCPsForClass men innebærer kun CR'er av typen X
<X>_numberCps	Tilsvarende som numberCps men innebærer kun CR'er av typen X
<X>_numberFilesChanged	Tilsvarende som numberFilesChanged men innebærer kun CR'er av typen X
<X>_numberDevInvolved	Tilsvarende som numberDevInvolved men innebærer kun CR'er av typen X
<X>_numberTestFailed	Tilsvarende som numberTestFailed men innebærer kun CR'er av typen X
<X>_numberPastCR	Tilsvarende som numberPastCR men innebærer kun CR'er av typen X
<X>_nLinesIn	Tilsvarende som nLinesIn men innebærer kun CR'er av typen X
<X>_nLinesOut	Tilsvarende som nLinesOut men innebærer kun CR'er av typen X
No.Methods | NOQ | NOC	Antall [implementerte | spørrings | command] metoder i en klasse
LCOM	Mangel på kohesjon i klassen
CCV	syklisk kompleksitet i klassen
TCC | MAXCC | AVCC	[Total | Maks | gjennomsnittlig] syklisk kompleksitet i klassen
MAXCC(2)	Samme som MAXCC (til bruk for egendefinert konfigurasjon)
NOS | UWCS	Klassens størrelse i [antall javauttrykk | antall attributter + antall metoder]
HEFF	Halstead-arbeid for klasse
EXT | LOC	Antall [eksterne | lokale] metoder som kalles av klassen
HIER	Antall metoder som kalles som er i klassehierarkiet til denne klassen
INST	Antall instansvariabler
MOD	Antall "modifiers" for deklarering av denne klassen
INTR	Antall implementerte grensesnitt for klassen
PACK	Antall importerte pakker
RFC	Total respons for klassen
MPC	Meldingsflytkobling
FIN	Antall unike metoder som kaller metodene i klassen
FOUT	Antall unike ikke-arv relaterte klasser som klassen er avhengig av
R/R | S/R	[Gjenbruks | spesialiserings] forhold for klassen
NSUP | NSUB	Antall [sub | super] klasser
MI | MINC	Gjenbruksindeks for klassen [med | uten] kommentarer
For releasene Z = {nm3, nm2, nm1, n, np1}:	
<Z>_CLL_CR	Antall store funksjonalitetsendringer for klassen i release Z
<Z>_CKL_CR	Antall mellomstore funksjonalitetsendringer for klassen i release Z
<Z>_CFL_CR	Antall små funksjonalitetsendringer for klassen i release Z
<Z>_M_CR	Antall vedlikeholdsendringer for klassen i release Z
<Z>_E_CR	Antall ikke-kritiske feilrettinger som er gjort for klassen i release Z
<Z>_CE_CR	Antall kritiske feilrettinger som er gjort for klassen i release Z
NumErrors	Summen av kritiske og ikke-kritiske feilrettinger for klassen
Error	Angir om det er foretatt kritiske/ikke-kritiske feilrettinger av klassen

4 Teknisk Implementasjon

Dette kapittelet gir en teknisk beskrivelse av løsningen som ligger til grunn for COS Data Manager. Først vil valg av teknologi og verktøy beskrives i kapittel 4.1. Videre gir kapittel 4.2 et innblikk i hvordan programmeringsspråket Python er benyttet for å lage COS Data Manager, mens kapittel 4.3 presenterer en relasjonsdatabase som fungerer som ”back end” løsning i COS Data Manager. Til slutt sier kapittel 4.4 litt om hva som er gjort for å verifisere at COS Data Manager gir korrekte dataverdier.

4.1 Valg av teknologi og verktøy

Ved utvikling av COS Data Manager er det benyttet Python versjon 2.4. Python er et dynamisk objektorientert programmeringsspråk som har god støtte for integrasjon mot andre programmeringsspråk og verktøy [13]. Språket er intuitivt og enkelt å lære, og er kjent for å gi utviklere gode muligheter for å lage oversiktlig og gjenbrukbar kode med høy kvalitet. Python kan kjøre på de fleste plattformer. Det er tilgjengelig gjennom en open source lisens og gratis å bruke.

For å gjøre utviklingsarbeidet i Python mer oversiktlig og effektivt er det benyttet Eclipse versjon 3.2 [14] i kombinasjon Pydev versjon 1.3. Pydev er en plug-in som gjør det mulig å benytte Eclipse som utviklingsverktøy for blant annet Python [15]. Pydev gir brukeren mange nyttige hjelpemidler som blant annet syntaks sjekk, syntaks ”highlighting” og automatisk avslutning av kodeuttrykk.

Som ”back end” løsning er det valgt å bruke MySQL. MySQL har utviklet seg til å bli verdens mest brukte open source database [16]. Stikkord som effektivitet, pålitelighet og brukervennlighet er noe av årsaken til dette. Versjonen som benyttes i dette prosjektet er MySQL Server 5.0. Dette er en gratisversjon som kommer uten grafisk brukergrensesnitt.

For å gjøre databaseutviklingen enklere og mer effektiv er det benyttet et verktøy som heter EMS SQL Manager 2005 for MySQL. Dette er et grafisk verktøy for administrering og utvikling av MySQL databaser [17].

Til slutt, for at Python skal kunne kommunisere med MySQL er det benyttet noe som heter MySQLdb. Dette er et grensesnitt som muliggjør kommunikasjon mellom Python og MySQL [18].

Det er satt opp en virtuell windowsmaskin på Simula der mye av utviklingsarbeidet har foregått. På denne maskinen er alle komponenter som er nødvendig for utvikling og anvendelse COS Data Manager installert. I fremtiden vil COS forskningsprosjektet ha tilgang til denne maskinen for å anvende og eventuelt videreutvikle COS Data Manager. Om ønskelig vil det også være mulig å installere COS Data Manager på andre maskiner.

4.2 Pyhtonscriptet

Pythonsscriptet som er laget i forbindelse med COS Data Manager er et omfattende script på ca 2200 kodelinjer, og er utviklet hovedasakelig for å utføre to ting: Importere data til COS-databasen, samt å generere ulike datasett på grunnlag av dataene som er lagret i databasen. For å få til dette må det utføres en rekke SQL-operasjoner mot databasen. Kommunikasjon mellom Pythonscriptet og databasen skjer ved hjelp av et grensesnitt som heter MySQLdb [18]. Dette grensesnittet gjør det mulig å opprette et ”connection” objekt og et ”cursor”

objekt. Objektene benyttes for å utføre SQL-operasjoner fra python direkte mot databasen, samt å hente ut resultater fra operasjonene. Resultater fra en SQL-operasjon kan blant annet hentes ut i pythonscriptet ved hjelp av cursor-metodene ”fetchone” og ”fetchall” [19].

Pythonscriptet består av en rekke funksjoner og et grafisk brukergrensesnitt som benyttes til å utføre funksjonene. To hovedfunksjoner utgjør mesteparten av funksjonaliteten i scriptet. Den første hovedfunksjonen er kalt ”importRelease” og inneholder alle subfunksjoner som sørger for at MKS-data blir importert til COS-databasen. Dette innebærer formatering og omstrukturering av data, konsistenssjekker, filtreringer og generering av importlogfil.

Den andre hovedfunksjonen er kalt ”createResultVIEW” og inneholder alle subfunksjoner som benyttes for å lage historisk view for utvalgte releaser. Her utføres det blant annet en rekke ”joins” mellom ulike tabeller, modifisering av data, samt beregninger av nye data. Det historiske viewet benyttes både når det skal genereres csv-fil til bruk for bygging/oppdatering av feilprediksjonsmodeller, og når det skal genereres csv-fil for siste release, som feilprediksjonsmodellen skal anvendes på.

I tillegg til de to nevnte hovedfunksjonene består scriptet av hjelpefunksjoner som sørger for å skrive status og resultater til det grafiske brukergrensesnittet, fildialog, samt funksjoner som genererer og importerer csv-filer. Disse funksjonene betjenes også gjennom det grafiske brukergrensesnittet (se Figur 1 kap 3.1).

Det grafiske brukergrensesnittet er laget ved hjelp av Tkinter. Tkinter er den mest brukte modulen for utvikling av grafiske brukergrensesnitt i Python [19]. Modulen er basert på Tk, og inneholder en rekke ferdiglagde ”widgets” som gjør det enkelt å lage grafiske komponenter som ”frame”, ”label”, ”textbox”, ”button” og en rekke andre velkjente komponenter. For mer detaljert informasjon om oppbygning av brukergrensesnitt og ulike funksjoner henvises det til kildekoden til COS Data Manager.

For at scriptet skal være enkelt å bruke er det laget en selvstendig exe-fil som gjør det mulig å kjøre scriptet selvstendig uten at Python er installert. Dette gjør det også enklere å installere COS Data Manager på andre maskiner om det skulle være ønskelig.

4.3 COS-databasen

Dette kapittelet beskriver de viktigste konfigurasjonene som er gjort i forbindelse med oppsett av COS-databasen. Det er også gitt en oversikt over de viktigste tabellene databasen, samt datamodell for lagring av grunnlagsdata.

4.3.1 Konfigurering

MySQL tilbyr en rekke alternativer i forbindelse med konfigurering og oppsett av en relasjonsdatabase. COS-databasen er satt opp som en InnoDB database [20]. Dette innebærer at databasen er satt opp som en tradisjonell transaksjonsdatabase som støtter ACID [21] egenskapene i forbindelse med transaksjoner, noe som blant annet gjør det mulig å foreta funksjoner som ”commit”, ”rollback” og ”crash recovery”.

Det er hensiktsmessig at COS-databasen er satt opp som en transaksjonsdatabase fordi dette sikrer blant annet at MKS-data for en release blir importert fullstendig. Hvis noe galt skulle skje i forbindelse med importprosessen, vil det ikke bli foretatt ”commit”, og dermed vil databasen gå tilbake til slik den var før importprosessen startet. Hvis derimot importprosessen

skjer uten problemer, vil det til slutt foretas en ”commit” som vil gjøre endringene på databasen varige.

MySQL databaser kan operere i ulike modi. Valg av modus avgjør hvilke SQL-kommandoer som støttes og hvilke type valideringer som skal gjøres for eksempel i forbindelse med innsetting av data [20]. COS-databasen er satt opp som ”strict_trans_tables”. Dette innebærer hovedsakelig at verdier som settes inn i en tabell ikke kan være NULL, samt at verdiene må ha korrekte datatyper. Dersom verdier ikke kan settes inn i tabellen blir operasjonen avbrutt. For COS-databasen vil dette være en hensiktsmessig måte å sikre at MKS-data importeres korrekt til databasen. Dersom importerte datasett skulle mangle enkelte verdier, eller inneholde feil type verdier, så vil hele importprosessen avbytes slik at ”commit” ikke blir foretatt. Databasen vil dermed gi feilmelding om verdier som mangler og/eller verdier som ikke godtas.

4.3.2 Tabeller

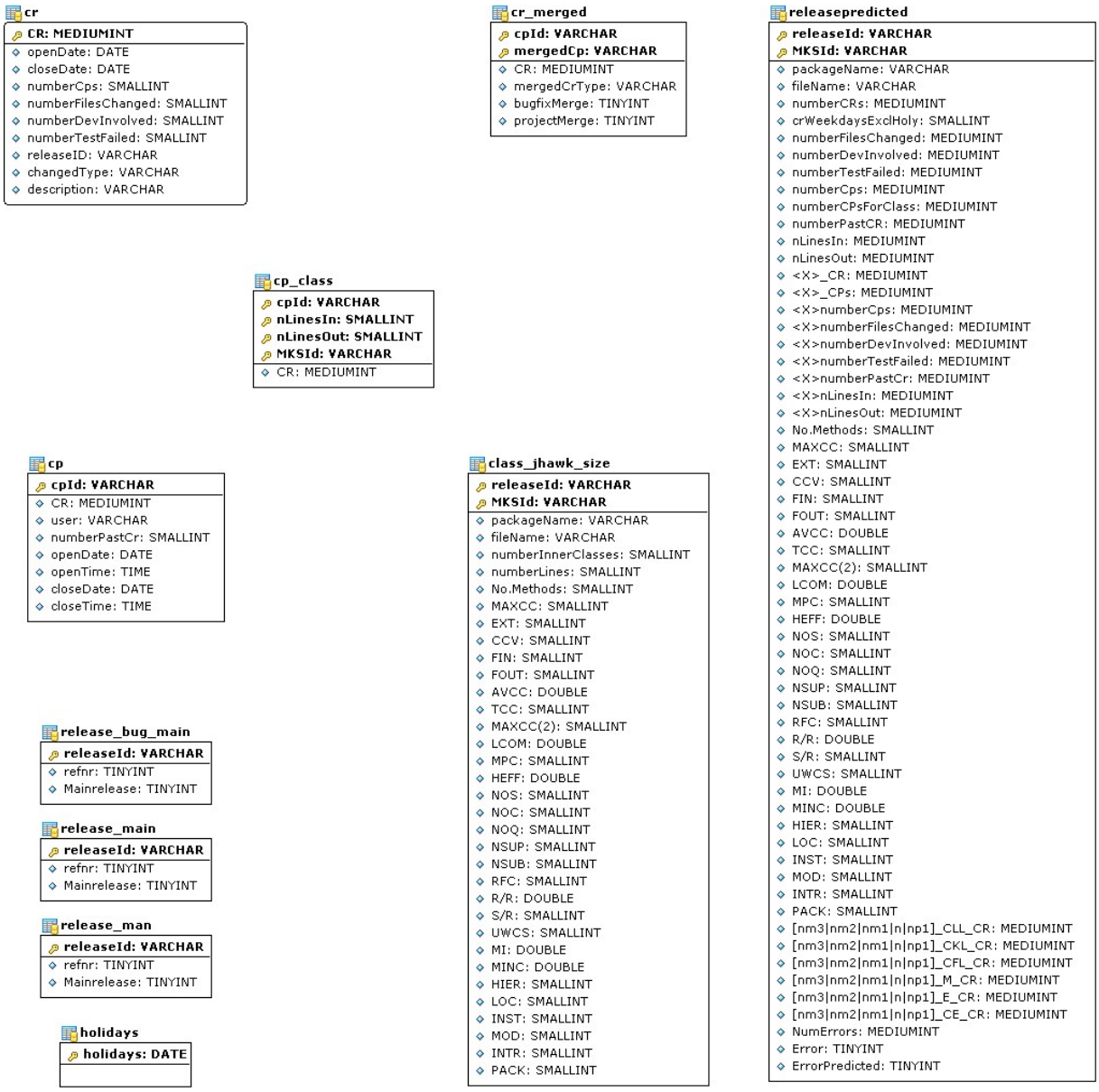
COS-databasen består av to typer tabeller som er kalt importtabeller og lagringstabeller. Hensikten med å benytte importtabeller er at det skal være mulig å importere MKS-data til databasen slik de er presentert i flatfilene (se kap 3), slik at dataene kan modifiseres til ønsket form før de blir satt inn i lagringstabellene. Importtabellene har ingen primærnøkler, noe som blant annet gjør det mulig å sjekke data i importtabellene for duplikater og andre tilfeller av inkonsistens. Dette gir mulighet for å gi en oversikt over hvilke data som fjernes, som igjen er viktig for å gi COS forskningsprosjektet best mulig forståelse av grunnlagsdataene som finnes i lagringstabellene. Dette skjer først og fremst ved at det genereres en importlogfil.

Import til importtabellene foregår ved hjelp av SQL-kommandoen ”Load data infile”. Dette er en kommando som sørger for raskt lasting av data fra en tekstfil [20]. Ettersom COS-databasen opererer i ”strict_trans_tables” modus, må det i tillegg benyttes opsjonen ”ignore into” som sørger for at dataene blir importert til importtabellene til tross for at ikke alle verdier/attributter er representert. Så snart alle modifiseringer av data i importtabellene er gjort, blir disse overført til lagringstabellene. Alle importtabellene tømmes dermed ved hjelp av kommandoen ”truncate”, slik at importtabellene er klare for å ta imot data for neste release.

Lagringstabellene benyttes for å lagre konsistente grunnlagsdata for alle importerte releaser på ønsket/korrekt form. Tabellene inneholder blant annet alle data som benyttes for å generere et historisk view, som igjen danner grunnlaget for bygging av feilprediksjonsmodellene. Det er derfor viktig at innholdet i disse tabellene er mest mulig optimalt.

Når data skal overføres fra importtabellene til lagringstabellene benyttes SQL-kommandoen ”replace into”. Denne kommandoen fungerer identisk med den mer tradisjonelle SQL-kommandoen ”insert into” bortsett fra tilfeller der et eksisterende tuppel har samme verdi på primærnøkkelen som tuppelet som skal settes inn [20]. I et slikt tilfelle blir det gamle tuppelet slettet før det nye settes inn. Dette betyr at hvis en gitt release importeres til databasen flere ganger, så vil verdier som allerede finnes i databasen for denne releasen bli oppdatert, mens øvrige verdier vil bli forbli uendret.

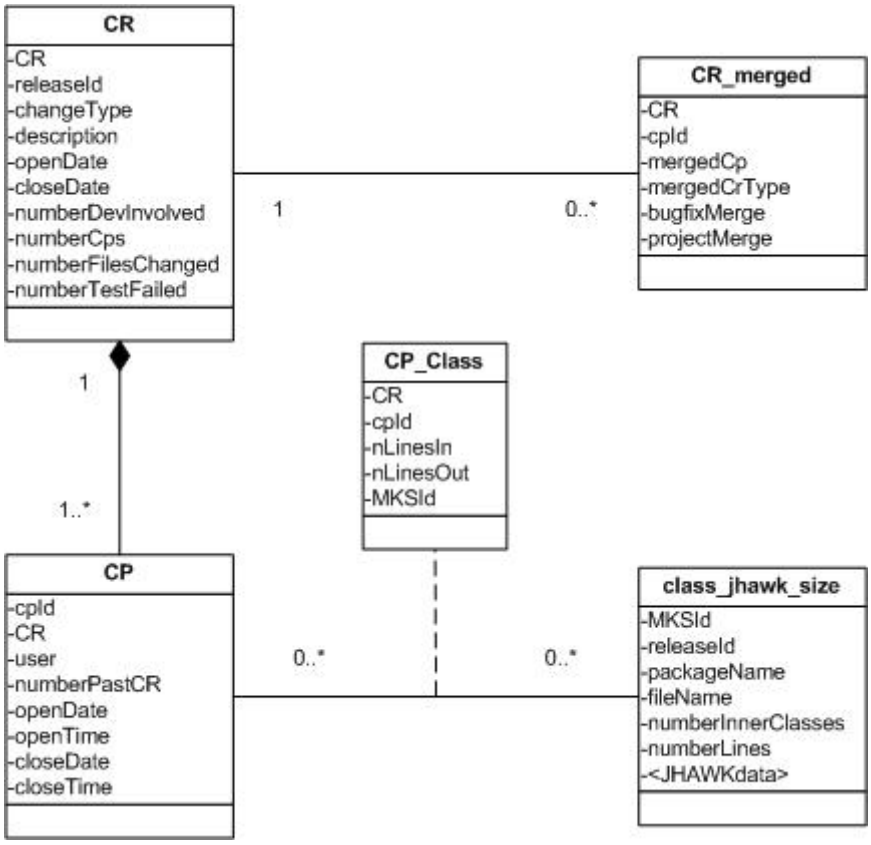
For å sikre konsistent lagring av data i lagringstabellene er det definert en primærnøkkel for hver enkelt tabell. Primærnøkkelen sikrer at det bare finnes unike tupler i tabellene. En komplett oversikt over alle lagringstabeller i COS-databasen er presentert i Figur 4. Av denne figuren fremgår også primærnøkkel for hver enkelt tabell.



Figur 4: Komplett oversikt over lagringstabeller og primærnøkler i COS-databasen

Tabellene cr, cr_merged, cp, cp_class og class_jhawk_size benyttes for å lagre grunnlagsdata som samles inn fra MKS. Datamodell for lagring av data i disse tabellene er presentert i Figur 5. Tabellene release_bug_main, release_main, release_man og holiday fungerer som hjelpetabeller når det skal genereres historisk view for utvalgte releaser. Tabellen releasepredicted er en tabell som benyttes for å lagre predikerte datasett. Når en csv-fil importeres ved hjelp av felt D i Figur 1 (se kap3.1) vil datasettet lagres i denne tabellen. Denne tabellen vil dermed kunne benyttes som grunnlag når det skal lages et presentasjonsverktøy som tilbyr COS-utviklerne daglige oppdaterte prediksjoner til bruk for fokuserte kvalitetsforbedringstiltak.

Ved import og generering av historisk view foretas det blant annet en rekke SQL-joiner av de nevnte tabellene. For at dette skal foregå så effektivt som mulig er det definert en rekke indekser. Som en hovedregel er det opprettet indekser på attributter som det joines på, der indeksene er av typen B-tree.



Figur 5: Datamodell for lagring av grunnlagsdata

I tabellene som inngår i datamodellen i Figur 5, er det kun er primærnøklene som bestemmer andre attributter i hver enkelt tabell, noe som betyr at datamodellen oppfyller kravet til ”Domain-Key Normal Form” [22].

Hvis man sammenlikner datamodellen i Figur 5 med datamodellen i Figur 3 så kommer det frem at CR tabellen i Figur 3 splittes opp i tabellene cr og cr_merged underveis i importprosessen. Dette er først og fremst gjort fordi dataene i cr_merged tabellen foreløpig ikke benyttes i COS forskningsprosjektet.

En annen ting som kommer frem ved å sammenlikne figurene er at Class, JHawk og Size i Figur 3 er smeltet sammen til én stor tabell i Figur 5. Dette skjer ved hjelp en en SQL-join som foretas underveis i importprosessen. Dette er gjort for at grunnlagsdataene skal være mest mulig klargjort når historisk view skal genereres.

4.4 Verifikasjon av automatisert prosess

Ettersom denne masteroppgaven har et nokså begrenset tidsomfang er COS Data Manager foreløpig ikke tatt i bruk i COS forskningsprosjektet. Men for å sikre at verktøyet fungerer som forventet er det foretatt en rekke testkjøringer der alle releaser har blitt importert til COS-databasen, og deretter har det historiske viewet blitt generert. For å kontrollere at denne prosessen gir korrekte verdier, er det foretatt sammenlikninger med resultatverdier som dagens manuelle prosess gir.

Hittil i COS forskningsprosjektet har det historiske viewet blitt laget ved hjelp av et script laget i statistikkverktøyet SAS. Det historiske viewet som dette scriptet genererer har dermed fungert som en slags fasit for hvordan verdiene i det historiske viewet generert av COS Data Manager bør fremstilles. Det er foretatt en rekke manuelle stikkprøvesammenlikninger mellom de historiske viewene. Her er det blant annet kontrollert at hver klasse gir nøyaktig samme verdier for alle attributter.

Det fremkommer også noen forskjeller ved en slik sammenlikning. Dette gjelder først og fremst antall klasser som inngår i det historiske viewet. Viewet som genereres av SAS scriptet inneholder for eksempel ca 33000 rader når mainreleasene 12-22 inngår. Til sammenlikning inneholder det historiske viewet generert av COS Data Manager ca 32500 rader for de samme releasene. Ved å undersøke forskjellene videre, fremkommer det at det historiske viewet generert av COS Data Manager er mer konsistent enn det man klarte å oppnå ved dagens manuelle prosess, der SAS scriptet benyttes.

5 Konklusjon og videre arbeid

Arbeidet med å automatisere og kvalitetsikre datainnsamlings- og håndteringsprosessen i COS forskningsprosjektet har hatt to hovedformål. Som et første punkt har det vært viktig å sikre et best mulig datagrunnlag for å kunne bygge mest mulig presise feilprediksjonsmodeller. Her har spesielt automatisering av konsistens- og kvalitetssjekker vært viktig. Det har også vært nødvendig å lage filtreringsrutiner som eksempelvis fjerner data som ikke kan identifiseres. Dette vil bidra til bedre kontroll over innsamlede data, noe som igjen vil øke muligheten for å kunne lage mer presise prediksjonsmodeller.

Det andre formålet har vært å legge til rette for å øke anvendelsesområdet for prediksjonsmodellene. Hittil i forskningsprosjektet har det blitt lagt vekt på at modellene skal benyttes til fokusert testing for utvalgte klasser i en dedikert testfase av prosjektet. Videre i forskningsprosjektet ønsker man også at feilprediksjonsmodellene blant annet skal benyttes for å gi COS-utviklerne oppdaterte prediksjoner daglig, under selve utviklingsarbeidet. For å få til dette er det lagt vekt på å lage en enkel og rask datahåndteringsprosess der en databaseløsning sørger for oppdaterte og konsistente prediksjoner.

Perlscriptet som benyttes til å samle inn MKS-data er modifisert slik at det har blitt enklere og raskere å foreta innsamlingsprosessen. Scriptet bruker fremdeles rundt 1 time på å samle data for en release, men oppgaver som tidligere ble utført manuelt i denne prosessen er nå blitt automatisert.

Den store tidsbesparelsen skjer imidlertid når grunnlagsdata skal omformes til datasett som benyttes for modellbygging og bruk av feilprediksjonsmodellen. Tidligere tok denne prosessen opp mot 1 uke å utføre, der gjerne flere personer var involvert. Ved hjelp av COS Data Manager kan én person enkelt utføre den samme jobben i løpet av få minutter, der kvalitets og konsistenssjekker foregår automatisk.

Gjennom arbeidet som er beskrevet i denne oppgaven er det funnet flere årsaker til at inkonsistens forekommer blant innsamlet MKS-data. Dette skyldes blant annet inkonsistens i Telenors MKS-system, noe som kan oppstå dersom COS-utviklerne anvender MKS feilaktig. Det finnes også klasser med samme navn ettersom liknende klasser benyttes til ulike formål. Disse ligger gjerne på forskjellige prosjekter og kan ikke skilles ved hjelp av data som samles inn. For at disse klassene skal kunne være med i datagrunnlaget som benyttes til å bygge modeller, er det nødvendig at JHawk modifiseres. ”Inner classes” blir heller ikke tatt med i datagrunnlaget ettersom disse ikke vises separat i MKS, samt at de heller ikke kan skilles av JHawk. Arbeidet med å konsistenssjekke MKS-data har ført til at forskningsprosjektet har fått en bedre forståelse av årsakene til inkonsistens i dataene, noe som i sin tur kan bidra til å lage mer nøyaktige prediksjonsmodeller.

På grunn av oppgavens begrensede tidsomfang, er den automatiserte prosessen foreløpig ikke anvendt i forskningsprosjektet, men for å sikre at metodikken fungerer som forventet og at den gir korrekte verdier, er det foretatt gjentatte kontroller. Dette er utført ved å foreta manuelle sammenlikninger mot den eksisterende/manuelle metodikken. Forskjellene som kom frem ved en slik sammenlikning var blant annet at det totale antall klasser som inngår i datagrunnlaget for modellbygging var redusert noe (fra ca 33000 til 32500 rader). Dette er imidlertid som forventet ettersom den automatiserte prosessen inneholder en rekke filtreringer som ikke ble gjort tidligere.

For ta forskningsprosjektet enda et skritt videre vil det være behov for å gjøre ytterligere automatiseringsarbeid for å oppnå en helautomatisert prosess. Dette innebærer blant annet at datainnsamlingsprosessen som foregår ved hjelp av Perlscript i dag, bør integreres med øvrig funksjonalitet for datahåndtering (Pythonscript).

Løsningen som er presentert i denne oppgaven beskriver COS Data Manager som et verktøy for blant annet å generere datasett som benyttes for å bygge feilprediksjonsmodeller i et maskinlæringsverktøy (Weka). For å få til en mer helautomatisert prosess bør anvendelse av feilprediksjonsmodeller foregå som en del av den automatiserte prosessen, der feilprediksjonsmodellen som er bygget i Weka lagres i databasen og anvendes automatisk gjennom funksjonalitet som tilbys av Pythonscriptet. Dette vil også kreve mulighet for å laste inn oppdaterte prediksjonsmodeller til databasen.

Et viktig punkt som vil være avgjørende for å øke tilgjengeligheten og anvendelsesområdene av metodikken er å utvikle et presentasjonsverktøy som tilbyr COS-utviklerne ulike visuelle fremstillinger av oppdaterte prediksjoner. En typisk utvikler vil kunne ha nytte av visualiserte prediksjoner under det daglige utviklingsarbeidet ved at de for eksempel blir varslet når de arbeider på kritiske deler av systemet. En arkitekt vil være mer opptatt av å se det store bildet, og vil følgelig ha nytte av å ha et visuelt hjelpemiddel som kan benyttes for å omstrukturere kode slik at kvaliteten på systemet kan bedres. QA personellet på sin side vil være mest opptatt av hvor i systemet det er størst sannsynlighet for feil – både komponent/klassevis og for moduler eller pakker. De ulike aktørene skaper et sammensatt bilde over hva porteføljen av visuelle fremstillinger bør inneholde.

Telenor bruker i dag blant annet Eclipse for utviklingsarbeidet i COS. Det er viktig at visualiseringene gjøres enkelt tilgjengelig for alle som har nytte av dem, og det vil derfor være hensiktsmessig å integrere visualiseringsverktøyet som en ”plugin” til Eclipse. Arbeidet med å utvikle et presentasjonsverktøy er allerede i gang gjennom en masteroppgave som utføres av Eivind Berg Johannesen.

Hvis metodikken på sikt skal kunne kommersialiseres og anvendes i andre utviklingsprosjekter, er det nødvendig at det lages en mer universell løsning. Dette innebærer blant annet at dataene som samles inn må kunne defineres i mer generelle datatyper, og at datamodellen som er grunnlag for datainnsamlingen må være konfigurerbar slik at systemet kan tilpasses til andre systemutviklingsprosjekter og konfigurasjonsstyringssystemer.

6 Kilder

[1] E. Arisholm, L.C. Briand and M.J. Fuglerud, *"Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software"*, Submitted to International Symposium in Software Reliability Engineering (ISSRE), 2007.

[2] E. Arisholm & L. C. Briand, "Predicting Fault-prone Components in a Java Legacy System" , ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), 2006.

[4] A. Mockus & D. M. Weiss, "Predicting risk of software changes", Bell Labs Technical Journal, 2000.

[5] Witten and E. Frank, "Data Mining: Practical Machine Learning Tools and Techniques", Second edition: Morgan Kaufman, 2005.

[6] R. Quinlan, "C4.5: Programs for Machine Learning" Morgan Kaufmann, 1993.

[7] MKS, http://www.mks.com

[8] XRadar, http://xradar.sourceforge.net

[9] JHawk, http://www.virtualmachinery.com/jhawkprod.htm

[10] Ian H. Witten and Eibe Frank (2005) "Data Mining: Practical machine learning tools and techniques", 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[11] RMI, http://java.sun.com/docs/books/tutorial/rmi

[12] CORBA, http://java.sun.com/javase/technologies/core/corba

[13] Python, http://www.python.org

[14] Eclipse, http://www.eclipse.org

[15] Pydev, http://pydev.sourceforge.net

[16] MySQL, http://www.mysql.com

[17] EMS SQL Manager 2005 for MySQL, http://www.mysqlmanager.com

[18] MySQLdb, http://sourceforge.net/projects/mysql-python

[19] Python Documentation, http://www.python.org/doc

[20] MySQL Documentation, http://dev.mysql.com/doc/refman/5.0/en

[21] Garcia-Molina, Ullman, Widom: "Database Systems - The Complete Book", Prentice Hall 2002.

[22] http://www.uio.no/studier/emner/matnat/ifi/INF3100/v05/undervisningsmateriale/lysark/kap03MVD.pdf