UNIVERSITETET I OSLO
Institutt for informatikk

Analysere dynamiske aspekter av problemløsning innenfor vedlikeholdning av objekt orienterte systemer

Masteroppgave
(60 studiepoeng)

Danny Phi Hung Pham

25. august 2008

Forord

Først vil jeg takke min veileder Amela Karahasanovic for support, oppmuntring og veiledning, og Simula Research Laboratory for muligheten til å skrive denne oppgaven. Denne masteroppgaven varer to semestere og tilsvarer 60 studiepoeng.

Jeg vil også takke My Loan Thi Nguyen for godt samarbeid og support. Deretter vil jeg benytte muligheten til å takke Kaja Kværn, Junisilver Taij og Richard Thomas for dokumentasjon av eksperimentet. Jeg vil også takke studentene fra Universistetet i Western Australia, og alle de andre som har vært med for å muliggjøre dette eksperimentet. Jeg vil også takke Athiraiyan Balachandran og Chi Hung Duong for inspill i løpet av oppgaveskrivingen.

Oslo, august 2008

Danny Pham

”When solving problems, dig at the roots instead of just hacking at the leaves.”

Anthony J. D'Angelo, (The College Blue Book, 1972)

Abstrakt

Fomålet med denne oppgaven er å identifisere problemløsningsmetoder og strategier brukt av informatikk studenter under prosessen av vedlikehold oppgaver på en ukjent applikasjon i Java. Deretter er effektene av utførsel av problemløsningsmetoder og problemløsningsstrategier utforsket.

Å bruke strategier for å løse oppgaver og å løse vanskeligheter som oppstod ved modifisering av et system kan være nyttig fordi kunnskap om dette kan bidra til utvikling av nye modeller, til å støtte opp for vedlikeholdningsprosesser, eller til å forbedre utdanning.

Det ble utførte et kontrollert eksperiment med 34 informatikk studenter som gikk på det tredje året i informatikk. Deltakerne brukte et professjonell Java verktøy til å utføre flere vedlikeholdningsoppgaver på et mediumstort Java applikasjonssystem i en 6 timers lang eksperiment. I denne oppgaven har jeg brukt data fra feedback-collection (tilbakemeldinger av deltakerne gjennom sprettoppvinduer mens de arbeidet med oppgavene) av eksperimentet.

I denne oppgaven var det identifisert tre generelle problemløsningsmetoder for løsning av problemer om oppstod i en vedlikeholdningsprosess. Disse tre metodene var (i) *abstraksjon*, der deltakerne løste et problem ved hjelp av fjerning av unødvendige detaljer, brakte inn nye krav som involverte identifisering av kritiske aspekter ved miljøet og systemet, mens de overså det som var irrelevant. (ii) *Dekomposisjon,* var der deltakerne løste et problem ved å dele problemet inn i mindre sub-problemer, for deretter å løse dem individuelt og kombinerte dem til en samlet løsning. Til slutt hadde vi (iii) *bygg og fiks,* der deltakerne hoppet rett inn i problemet og løste dem med førsteinntrykksløsninger. Forskningen i dette eksperimentet viste at abstraksjon har bedre ytelser til løsing av vanskeligheter med program logikk og struktur, mens dekomposisjon og bygg og fiks yta bedre i løsninger av vanskeligheter med GUI. Det var også identifisert seks problemløsningssteger som grunnprinsipper for problemløsningsstrategier av vedlikeholdningsoppgaver. Disse seks stegene var (1) *formulering av problemet,* (2) *planlegging av løsningen,* (3) *konstruering av løsningen,* (4) *oversettelse av løsningen,* (5) *testing av løsningen* og (6) *levering av løsningen*. Forskningen i eksperimentet viste at strategier med integrasjon av følgende steger, (2) *planlegging av løsningen* og (5) *testing av løsningen*, bidro til høyest riktighetsprosent og lavest tidsbruk ved løsning av oppgavene.

Innholdsfortegnelse

Liste av tabeller

Liste av figurer

1 Introduksjon

1.1 Motivasjon

Den stødige veksten og forbrukernes behov for IKT har gjort program vedlikeholdning (program maintenance) til en viktig og kostbar investering innenfor industrien. En vedlikeholdningsprosess består av to viktige steg: forståelse, og modifisering (Eierman & Dishaw, 2007). Programforståelse (program comprehension) er en prosess for å forstå fremmede programmer og problemløsninger er en prosess for å løse vanskeligheter som oppstod ved modifisering av fremmede programmer. Disse to prosessene er vesentlige aktiviteter for suksessfulle program vedlikeholdningsarbeider ((Koenemann & Robertson, 1991); (von Mayrhauser & Vans, 1995): (Corritore & Wiedenbeck, 2001)). For å forstå et fremmed program og løse de problemene som oppstår er ikke en alminnelig oppgave. Selv om det er rapportert avviker, er mesteparten av forskningen innenfor program vedlikeholdning enige om at mer enn 50 prosent av programmeringsinnsatsen er gjort ved systemer der det ble lagt til forståelser og implementeringer (von Mayrhauser & Vans, 1995).

Erfaringen er anerkjent som den med mest innflytelse på kvalitet og produktivitet av program utvikling og vedlikeholdningsarbeid (Jones, 1998). Likevel har informatikkstudenter hovedsakelig lært hvordan de utvikler forholdsvis små programmer forfra på skolekurs. De får dermed ikke tillært seg kunnskaper innenfor forståelse (Robins, Rountree, & Rountree, 2003), og i å løse komplekse problemer av fremmede koder (Or-Bach & Lavy, 2004). Ved å identifisere programforståelsestrategiene (strategi for sforståelse av ukjent kode), problemløsningsmetodene (metoder for å løse problemer som oppstår) og problemløsningsstrategiene (strategi for å løse oppgaver) studentene bruker ved program vedlikeholdning, kan bidra til nyttig informasjon for både videreutvikling av problemløsningsteknikker og i utdanning innenfor programutvikling.

Programforståelse og problemløsning har blitt forsket bredt på. Det er definert en håndfull av forskjellige måter å tilegne seg forståelse av et ukjent program på ((Détienne, 1997); (Storey, 2005)). Forskning har vært ledende innenfor kontekstet av generelle strategier ((Burkhardt, Detienne, & Wiedenebeck, 1998)), og det tydet på at retning av forståelse er bottom-up (Pennington, 1987), top-down (Brooks, 1983) eller en blanding av disse to metodene (von Mayrhauser & Vans, 1995). Bredden av forståelse er beskrevet i to strategier i form av systematisk og as-needed (Littman, Pinto, Letovsky, & Soloway, 1986). Det har også blitt definert en rekke forskjellige vanskeligheter som oppstod i programmering ((Basili & Perricone, 1984); (Spohrer & Soloway, 1986); (Karahasanović, Levine, & Thomas, 2007); og problemløsningsstrategier for å løse disse problemene på ((Linn & Clancy, 1992); (Deek, Turoff, & McHugh, 1999). To ledende metoder som har blitt meget forsket på mot komplekse problemer er abstraksjon (Larsen & Naumann, 1992) og dekomposisjon (Gallagher & Lyle, 1991). Forskning har blitt gjennomført i kontekst av software vedlikeholdning ((Koenemann & Robertson, 1991); (Corritore & Wiedenbeck, 2001)) debugging ((Kaminski, 1988); (Chmiel & Loui, 2003); (Ahmadzadeh, Elliman, & Higgins, 2005)), design (Détienne, 1997) og problemløsnings aktiviteter ((Vee, Meyer, & Mannock, 200X); (McCracken, et al., 2001); (Lister, et al., 2004); (Hundhausen, Brown, Farley, & Skarpas, 2006)). Koenemann og Robertson (1991) mente at programforståelse var en kompleks problemløsningsprosess, og Corritore og Wiedenbeck (2001) mente at vår begrep av programforståelse er ukomplett, og at det trengs dypere kunnskaper av forståelsestrategier under program vedlikeholdningsoppgaver.

Programmet som har blitt brukt i dette studiet var av et stort bibliotek applikasjon system med 3600 linjer med kode (LOC) skrevet i Java, og kan betraktes som et medium-stort applikasjon i forhold til klassifikasjonen gitt av Mayrhauser og Vans (1995). Deltakerne var 34 studenter. De var i deres tredje års studium i informatikk ved Universitet i Western Australia (UWA). Eksperimentet varte i sju timer og deltakerne utførte tre vedlikeholdnings oppgaver på det gitte applikasjonssystemet, i programmeringsmiljøet JBuilder. Deltakerne var utstyrt med dokumentasjon som beskrev applikasjonenssystemet og JBuilder dokumentasjon. De hadde også adgang til Java online dokumentasjonen.

1.2 Hensikt

Hensikten med dette studiet var å identifisere problemløsningsmetoder brukt av informatikk studenter for å løse vanskeligheter som oppstod under vedlikeholdningsoppgaver av et ukjent medium-stort objekt orientert (OO) applikasjon, og om studentene utviklet noen form for problemløsningsstrategi i prosessen for besvarelse av vedlikeholdningsoppgavene.

Data av feedback-collection (tilbakemeldinger av deltakerne gjennom sprettoppvinduer mens de arbeidet med oppgavene) fra det kontrollerte eksperimentet ble brukt til å besvare følgende forskningsspørsmål:

- Hvilken problemløsningsmetode bruker informatikkstudenter for å løse vanskeligheter som oppstår i prosessen av vedlikeholdningsoppgavene i et OO system?
- Hvilke effekter har problemløsningsmetoder for løsning av vanskeligheter som oppstår i prosessen av vedlikeholdningsoppgaver i et OO system?
- Bruker informatikkstudenter noen form for strategi for å løse gitte oppgaver i en vedlikeholdningsprosess av et OO system?
- Hvilke effekter har problemløsningsstrategiene informatikkstudentene utviklet for å løse gitte oppgaver i en vedlikeholdningsprosess av et OO system?

1.3 Forskningsmetode

Forskningen består av et kontrollert eksperiment med 34 informatikk studenter i deres tredje år i informatikk ved Universitetet i Western Australia. Deltakerne ble introdusert til et 3600 LOC Java applikasjon og ble bedt om å utføre tre vedlikeholdningsoppgaver på det gitte applikasjonsystemet, ved å bruke programmeringsmiljøet JBuilder. Eksperimentet varte i sju timer, og deltakerne var rustet med dokumentasjon som beskrev applikasjonenssystemet og JBuilder dokumentasjonen. De hadde også tilgang til Java sin internett dokumentasjon. Data brukt i denne forskningen var av feedback-collection. Kvaliteten av besvarelsene ble gradert av en ekstern konsulent. Det ble også registrert tidsbruk av prossesene deres gjennom dataverktøyet GRUMPs (Thomas & Kennedy, 2003).

Eksperimentet bestod av 3 vedlikeholdningsoppgaver med følgende vanskelighetsgrader. Forandringsoppgave 1 var lettest og forandringsoppgave 3 var vanskeligst. Før eksperimentet ble det utført treningsoppgaver for å gjøre deltakerne kjent med eksperimentsmiljøet. Kalibreringsoppgaven er for å fastslå deltakernes dyktighet i programmering. Oppvarmningsoppgaven er for å gi deltakerne kjennskap til feedback-collection metode. Tilslutt avsluttet eksperimentet med gruppe intervjuer.

1.4 Forskningssammenheng

Eksperimentet brukt i denne oppgaven var en del av Comprehensive Object-Oriented Learning (COOL) prosjektet som avsluttet i 2005. COOL var et pågående tre års forskningsprosjekt lansert i 2002 av et konsortium av fire norske institusjoner: InterMedia, Norwegian Computing Center, Simula Research Laboratory og Department of Informatics hos Universitetet i Oslo.

COOL sin målsetning var innsikt i det komplekse området av læring og undervisning i objekt orientert konsepter og å forhøye bevistheten av problem området i samfunnet av informatikk utdanningen og data samarbeidsutviklingen. Denne målsettingen var en tilnærmelse av en rekke forskjellige design eksperimenter og eksempler, og gjennom studier av eksisterende treninger. Mer detaljer om COOL kan finnes på prosjektets hjemmeside (COOL, InterMedia COOL homepage, 2005).

1.5 Bidrag

Det er flere bidrag i denne oppgaven:

1. Identifisering av studentenes problemløsningsmetode av vanskeligheter som oppstod i prosessen av vedlikeholdningsoppgaver.
2. Identifisering av effekter ved utførelse av problemløsningsmetoder som oppstod i prosessen av vedlikeholdningsoppgaver.
3. Identifisering av studentenes problemløsningsstrategier ved løsning av vedlikeholdningsoppgaver.
4. Identifisering av effekter ved utførelse av problemløsningsstrategier ved løsning av vedlikeholdningsoppgaver.

1.5.1 Identifisering av problemløsningsmetoder og effekter ved prestasjon

Denne forskningen tilbyr kunnskaper med hensyn til problemløsningsmetoder av vanskeligheter som oppstod i prosessen av vedlikeholdningsoppgavene. Det er identifisert tre generelle problemløsningsmetoder som studentene bruker for å konfrontere vanskeligheter som oppstod ved løsing av vedlikeholdningsoppgaver. Disse tre metodene er (i) *abstraksjon*, der deltaker løser et problem ved hjelp av fjerning av unødvendige detaljer, bringer inn nye krav, som involverer identifisering av kritiske aspekter ved miljøet og systemet, mens man overser det som er irrelevant. (ii) *Dekomposisjon,* der deltaker løste et problem ved å dele problemet inn i mindre sub-problemer, så å løse dem individuelt og kombinere dem til en samlet løsning. Til slutt har vi (iii) *bygg og fiks,* der deltaker hopper rett inn i problemet og løser dem ved førsteinntrykkløsninger.

Et annet bidrag av denne forskningen er hvordan problemløsningsmetoder brukt av deltakere påvirker deres prestasjon. Bedømmelsen av besvarelsene viser at abstraksjon yter bedre til løsning av vanskeligheter med program logikk og struktur, mens dekomposisjon og bygg og fiks yter bedre i løsning av vanskeligheter med GUIer med hensyn til riktighetsprosent og tidsbruk. Funnet er verdifullt med tanke på at ingeniører kan få innsikt i hvordan å løse vanskeligheter som oppstod i en vedlikeholdningsprosess. Funnet i oppgaven viser til anbefaling til å lære problemløsningsmetoder for å takle vanskeligheter som kan oppstå.

1.5.2 Identifisering av problemløsningsstrategier og effekter ved prestasjonen

Forskningen i eksperimentet viser til en rekke forskjellige strategier som studentene utførte under løsingsprosessen av vedlikeholdningsoppgavene. Problemløsningsstrategiene er identifisert av deltakernes kombinasjon av seks problemløsningssteg; (1) formulering av problemet, (2) planlegging av løsningen, (3) konstruering av løsningen, (4) oversettelse av, (5) testing av løsningen og (6) levering av løsningen. Funnet i eksperimentet viser til at strategier med integrasjon av følgende problemløsningsteg; (2) *planlegging av løsningen* og (6) *testing av løsningen*, bidrar til høyest riktighetsprosent og lavest tidsbruk ved løsing av oppgavene gitt i dette eksperiementet.

1.6 Struktur

I denne oppgaven er deler av kapitel 2 (2.1 programforståelsestrategier) skrevet i samarbeid med masterstudent My Loan Thi Nguyen (2008). For å gi en fullstendig forståelse av denne oppgaven er beskrivelsen av eksperimentet, tatt fra Junisilver Taij (2005) studium, med i kapitel 3. Resterende av denne masteroppgaven er strukturert på følgende måte:

- Kapittel 2 presenterer en oversikt over programforståelsestrategier og relaterte studier innenfor problemløsning.
- Kapittel 3 beskriver en detaljert beskrivelse av eksperimentet.
- Kapittel 4 presenterer analysen av data fra feedback-collection i eksperimentet.
- Kapittel 5 beskriver resultatet av analysen.
- Kapittel 6 påpeker validitet av eksperimentet.
- Kapittel 7 presenterer konklusjon og framtidige arbeid av eksperimentet.

2 Identifisering av Relatert arbeid

I dette kapitelet presenterer jeg en oversikt over tidligere forskninger i programforståelse (program comprehension) og problemløsning (problem solving) innenfor programmering.

Relatert arbeid innenfor programforståelse (kapittel 2.1) er blitt gjort i samarbeid med masterstudent My Loan Thi Nguyen (2008). Dette ble identifisert av artikler vi fikk utdelt fra veileder Amela Karahasanovic i 2007. Artiklene var blant annet av (von Mayrhauser & Vans, 1995), (Détienne, 1997), (Corritore & Wiedenbeck, 2001), (Burkhardt, Detienne, & Wiedenebeck, 1998), (Levine, 2005), (Kværn, 2006), (Tømmerberg, 2006) og (Karahasanović, Levine, & Thomas, 2007). En del andre artikler ble funnet fra referanser av disse artiklene.

For å finne fram relatert arbeid av problemløsning har jeg søkt gjennom digitalt bibliotek og referanse databaser. Bibliotekene som jeg har vært innom er ACM Digital Library, IEEE Explore og UiO bibliotek. Google og Google Scholar søkemotor ble også brukt.

Søkene er basert på søkeordene:

- Program
- Maintenance
- Problem solving
- Strategy
- Object oriented
- Abstraction
- Decomposition
- Build and fix

Søket startet med søkeordene ”program problem solving”. Av dette kom det over hundre tusen treff. For å finne mer relevante artikler, kombinerte jeg disse søkeordene i forskjellige rekkefølger. Ettersom jeg la til flere søkeord, ble det mindre antall treff, men av mer relevante artikler. En del artikler ble også funnet ut fra referanser i de relevante artiklene. Ved å lese artiklene sine abstrakter og deres konklusjoner har jeg silt ut 12 artikkler som er meget relevante. Siste søking var i juni 2008.

2.1 Programforståelsesstrategier

Et program vedlikeholdningsprosess ble først kategorisert av Lientz og Swanson (1980) inn i fire grupper; (1) adaptiv (forandring i programmets miljø), (2) perfektivt (nye bruks krav), (3) korrektiv (fikse feil) og (4) preventivt (forhindre problemer in framtiden). I Bennet et al. (2000) studium påpekte de at forskningen som har blitt utført, var det gruppe (1) og (2) som utgjorde 75 % av innsatsen i en vedlikeholdningsprosess, og 21 % utgjorde feilrettingen. De påstod at programforståelse utgjorde en viktig fase før implementeringen, for når endringer og dets innvirkninger har blitt forstått, er det relativt enkelt å gjøre forandringer senere i programmet.

Det har vært mange studier av forskjellige aspekter av programforståelse. Forskningen har vært ledende mot generelle strategier, men også mot program vedlikeholdning og forbedring. Hovedemnene som har blitt studert er innenfor retning av forståelse og bredde av forståelse. Lite har blitt forsket innenfor problemløsning av problemer som utviklerne møter i en

vedlikeholdningsprosess og forandringer av koder. Dette skal jeg drøfter om i senere kapitler. Før vi kommer til det skal jeg forklare litt om de forskjellige programforståelsesmodellene.

Eierman og Dishaw (2007) forklarte en vedlikeholdningsprosess som to viktige steg; (1) forståelse og (2) modifisering. Forståelse var beskrevet som utvikling av en mental modell av problemet som skal løses. Dette krevde at utvikleren måtte forstå aktuelle funksjoner av programmet som skal modifiseres. Forståelse inkluderte identifisering av hvordan programmet fungerte og hvor programmet trengte å bli forandret for å gi korreksjon, perfeksjon eller krav av adapsjon. Modifisering involverte forandringer av programmet, slik at programmet implementerte de nye kravene korrekt. Dette kunne også involvere implementering eller modifisering av kode og testing av forandringene.

2.1.1 Retning av forståelse

2.1.1.1 Bottom-up

Fremgangsmåten for denne teorien er å begynne forståelsesprosessen fra det laveste nivået først, altså selve programkoden. Programmereren danner seg først et bilde (mental modell) av hvordan de forskjellige bitene (chunks (Storey, 2005)) i programmet fungerer, slik at han kan sette dem sammen for å oppnå en høyere forståelse av systemet. Slik fortsetter forståelsesprosessen helt til et høynivå forståelse av systemet er oppnådd.

2.1.1.2 Top-down

Brooks (1983) sin programforståelsesmodell, top-down, går fram ved å formulere en generell hypotese av funksjonaliteten til programmet først, for så å rekonstruere programmet ved hjelp av dybde-først hierarki med tileggshypotese i forhold til kilde kode. Modellen går altså fra høynivå abstraksjon eller konsept, ned til detaljerte nivå av programmet. Soloway, Adelson og Ehrlich, hentet fra Mayrhauser & Vans artikkel (1995), forskning viser til at top-down modellen blir som regel brukt når kode eller kodetype er kjent, for å lette innarbeide kunnskap av domenet. Modellen har 3 typer framgangsmåter: (1) strategi, (2) taktikk og (3) implementering. Prosessen begynner fra en høynivåforståelse og genererer dermed dypere forståelse i lavere nivå, for så å oppnå full forståelse av hele programmet. Program dokumentasjonen og program koden spiller en viktig rolle i framgangsmåten.

Det kan oppstå problemmer i modellen hvis:

1. Koden til hypotese finnes ikke.
2. Forvirring på grunn av at noen koder kan tilfredsstille andre hypotese
3. Koden forstås ikke.

2.1.1.3 Shneidermann og Mayer modell

Denne modellen starter først med å bruke vår korttidshukommelse til å huske systemet. Vi bygger deretter opp et "intern semantikk" representasjon (von Mayrhauser & Vans, 1995) som støttes opp av vår langtidshukommelse. Langtidshukommelsen vår har en kunnskapsbase som består av to deler:

den syntaktiske og den semantiske. Den syntaktiske kunnskapen er programmeringsspråk relatert. Det er lettere for oss å forstå og huske syntaksen eller foreta forandringer i en programmeringskode, dersom vi har nok kunnskap om det programmeringsspråket koden er skrevet i. Den semantiske delen inneholder den generelle kunnskapen innenfor programmering, uavhengig av programmeringsspråket. I motsetning til den syntaktiske delen er den semantiske delen delt opp i flere lag. Disse lagene består av, alt fra lavnivå detaljer til høynivå konsepter. Vi starter med å forstå koden til programmet, og går videre oppover til vi får en full forståelse av problemområdet. Denne modellen kan være enten en top-down eller bottom-up modell.

2.1.1.4 Pennington modell

Pennington (1987) påstod at når en programmerer prøver å sette seg inn i et nytt program, så vil personen alltid starte med å danne seg et bilde av hvordan kontrollflyten i programmet fungerer. Denne representasjonen er, som i de fleste bottom-up modellene, bygget opp fra bunnen. Programmereren må først forstå de essensielle bitene i koden, for så å kunne bygge opp den videre forståelsen. Den første modellen er programmodellen som blir utviklet ved at mikrostrukturene (som kontroll konstruksjon og oversikten over sammenheng til de forskjellige delene) grupperes til å danne helhetsstrukturer. Vi må foreta krysshenvisninger av de overnevnte strukturene for å sikre oss en riktig forståelse. Etter at programmodellen er fullført begynner vi på situasjonsmodellen. Det kreves at vi har nok kunnskap om generelle systemarkitekturer og funksjonalitet for operativsystemer siden denne modellen hjelper oss til å forstå programmet i forhold til faktiske ”real-world” hendelser. Det er viktig at vi har nok forståelse til å se sammenhengen. Som programmodellen er også denne forholdsvis bygd opp av å gruppere sammen mindre forståelse til større forståelser. Disse kunnskapene blir lagret i vår langtidshukommelse og er med på å oppdatere vår kunnskapsbase. Sistnevnte modell er fullført når programmets mål er nådd.

2.1.1.5 Letovsky modell

Letovskys programforståelse modell er et høynivå forståelsesmodell sammensatt av 3 komponenter, nevnt av von Mayhauser og Vans (1995) som følger:

1. kunnskapsbase
2. mental modell
3. assimilasjonsmodell

Kunnskapsbase er programmeringskyndighet, kunnskaper over problemsområde, regler, planer og mål av programmet. Mental modell består av spesifikasjon, implementasjon og kommentar lag. Assimilasjonsmodell er forståelsesprosessen. Prosessen kan da være enten top-down eller bottom-up.

2.1.1.6 Integrated meta-modell

Von Mayrhauser og Vans (1995) mente at i programforståelse brukes det enten top-down modell, bottom-up modell eller en kombinasjon av de to. I deres teori består programforståelse av 4 viktige komponenter:

1. Program modell
2. Situasjon modell
3. Top-down (domain) modell
4. Kunnskapsbase

De første 3 modellene beskriver forståelsesprosessen og nødvendige kunnskaper for å fullføre forståelsesprosessene. Program modellen brukes når programmereren er helt ukjent med programmet eller program koden, men gjenkjenner programmeringsspråk og struktur. Programmereren utvikler da en abstrakt forståelse av programmet, kalt "kontroll-flyten", ved å utføre en systematisk strategi av programmet. Etter at implementasjonisten har fått en abstrakt forståelse av programmet, vil utvikleren gå over til situasjon modellen som gir en funksjonell abstrakt begrep av kode, kalt "data-flyten". Dette kan gjøres ved å bruke bottom-up modell og opportunistisk strategi for å lage data-flyten. Top-down som ble beskrevet over i top-down avsnittet. Dette blir ofte brukt i "as-needed" strategien, som er å undersøke den delen av koden som trengs og ikke hele, og når program språket og kode er kjent. Særpreget av Integrated modellen er at de 3 modellene kan forekomme når som helst i forståelsesprosessen. Rekkefølgen er ikke fastlagt, og kan veksles med hensyn til koden Integrated modellen ble bevist av von Mayrhauser og Vans (1995) gjennom analyse av eksperimenter. Eksperimentet deres består av 9 profesjonelt programvedlikeholdere og 40K+ linjer av C koder. Deltakerne er eksperter innenfor området og språket, kjenner til strukturen og kallfunksjonene og har sett litt på design dokumenter. Utviklerne ble bedt om å tenke høyt mens de prøvde å forstå koden på 2 timer. Alt dette ble tatt opp på lydbånd. Etter at dataene ble analysert, fikk von Mayrhauser og Vans (1995) en statistikk av utført arbeid. 47 % er bruken av program modell, 31 % av top-down modell og 22 % av situasjon modell. Til å begynne med brukte utviklerne opportunistisk strategi for å forstå programmet. Tidlig i prosessen klarte de umiddelbart å avgjøre hvilket deler av koden de ikke trengte å forstå. Dette fungerte til tider som veiviser for utviklerne for programforståelsen gjennom kodene. Hypoteser forekom stadig og er en viktig prosess i forståelsen og gjenkjennelse av koden. Hypotesene kan føre utviklerne videre i prosessen eller føre utvikleren over til en annen forståelse modell. Statistikken referer noe til at en mental modell av programmet ble dannet veldig tidlig når utviklerne starter med og utforske kodene, og henviser til at programmodellen er et fortrinn til en bottom-up forståelse. Resultatet henviser til Integrated modell ettersom de 3 modellene forekommer flere steder i forskjellige tidspunkter i forståelsesprosessen.

2.1.2 Bredden av forståelse

De to kjente strategiene er den opportunistiske og den systematiske. Med den opportunistiske går programmereren bare gjennom den delen av koden personen mener er relevant for sitt formål. Dette kan spare tid, men vil også kunne medføre komplikasjoner i ettertid, siden man mangler en total forståelse av hvordan komponentene fungerer sammen. En annen fremgangsmåte som tar lengre tid, men som fungerer bedre er den systematiske strategien. Der velger programmereren å se gjennom hele programmet, del for del, for å oppnå en helhetsforståelse av systemet. Dette vil antageligvis være den sikreste måten å arbeide på (Tubaishat, 2001).

Littman et al. (1986) observerte at enten programmere systematisk gjennom kodene i detalj, lokaliserte gjennom control-flow og data-flow abstraksjon i programmet for å få en global forståelse

av programmet, eller så brukte de as-needed fremgangsmåten, som da fokuserte bare om koden relatert til en partikulær del av oppgaven.

2.1.3 Oppsumering

Alle de overnevnte modellene i retning av forståelse har noe til felles. De har en såkalt kunnskapsbase i langtidshukommelsen, som brukes til å bygge opp den nye forståelsen. Nivå av detaljer for hver av de modellene er ganske varierende. Noen kan være veldig beskrivende i hvordan tankeprosessen foregår, som Pennington modellen, mens andre er mer høynivå som Letovsky modellen. Retningen for programforståelse av disse modellene er enten top-down, bottom-up eller en blanding av disse to modellene. En ekspert kunne ha valgt å se ting fra et høynivå perspektiv, mens en novise uten erfaring ville kanskje ha begynt fra grunnleggende ting som har et mer bottom-up preg. Følgende faktorer kan ha mye å si hvordan en programmerer velger å gå fram for å forstå programmet; (1) størrelse av programmet, (2) programmeringsspråket og (3) ekspertise.

Mye er blitt forsket på innenfor vedlikeholdning, spesielt om forståelsesprosessen (forståelsesstrategiene og bruken av dem). Men dessverre er blitt lite forsket på innenfor problemløsning av problemer som oppstår i vedlikeholdningsprosessen. Utviklerne må konstant løse problemer som oppstår ved forandringer av koder, og ekstra vanskelig blir det når det er et ukjent program. Profesjonelle programutviklere tillærer seg egne strategier for å løse problemene etter mange år med praksis, og de er flinkere til å dekomponere og abstrahere problemene slik at det blir lettere å løse dem i motsetning til nybegynnere (Koenemann & Robertson, 1991).

Av modellene ovenfor ser jeg at disse modellene inneholder enten dekomposisjon eller abstraksjon eller begge, som en del av prosessen for bedre kjennskap til å forstå ukjent kode ved vedlikeholdning av programmet. Tabell 1 nedenfor viser at disse to generelle problemløsningsmetodene; dekomposisjon og abstraksjon, er meget utbredt og velbrukt, og et av de viktigste fundamentene til retning av programforståelsesmodellene.

Modeller	abstraksjon	dekomposisjon
Storey	---	x
Brooks	x	---
Shneidermann og Mayer	x	x
Pennington	---	x
Levotsky	---	x
Mayhauser og Vans	x	x
Soloway, Adelson og Ehrlich	x	x

Tabell 1. Forståelsemodeller med abstraksjon og dekomposisjon

2.2 Problemløsning innenfor programmering

Programmering er en fundamental del innenfor fagkretsen informatikk. Forskning innenfor programmering har eksistert i nesten to tiår, og arbeidet har ofte hatt fokus på studentenes logiske mistydinger i programmering og debugging (Kaminski, 1988).

2.2.1 Problemer som kan oppstå

Mange problemer kan oppstå etter en endring eller oppdatering av et program. Finne og løse disse problemene er en tidskrevende og vanskelig oppgave for nybegynnere i motsetning til de med erfaring (Spohrer & Soloway, 1986). Feil som oppstår i et data program har Chmiel et al. (2004) kategorisert i tre generelle type defekter; syntaks-, semantikk- og logisk feil. Syntaks feil er identifisert av kompilator og interpreter, mens semantikk og logisk feil må være identifisert av programmereren.

Syntaks feil forekommer på grunn av ukorrekt grammatikk (Ahmadzadeh, Elliman, & Higgins, 2005). Disse feilene er ofte fanget opp av kompilatoren/interpreter som at koden inneholder invalid språk elementer eller inkluderer elementer i feil rekkefølge. Trivial syntaks feil var ofte oversett i litteraturen (Spohrer & Soloway, 1986), kanskje fordi de var lett å fange opp av kompilatoren eller interpreter. Someren (1990) påstod til og med at ”*Syntaksen av de fleste programmeringsspråk er veldig begrenset og, i innledende fase, bare en kilde av feil*”, selv om disse feilene var lett å rette opp, kan de fortsatt være skadelig og frustrerende. Eksempel på syntaks feil kan være mangel på semikolon (;) ved enden av en setning.

Semantikk feil forekommer på grunn av misbruk av programmering konseptet, selv med korrekt syntaktisk struktur (Fleury, 2000). Eksempel av semantikk feil er feilbruk av type i et oppdrag. Semantikk feil blir fanget opp ved kompilering av program koden.

Logisk feil forekommer når programmet ikke løste et problem som programmet var ment for å løse. Dette er feil fra programmereren sin side, som feilet i å programmere riktig løsning (Fleury, 2000). Logisk mistyding innenfor programmering og debugging har ofte blitt fokusert på. I Spohrer og Soloway (1986) studium ramset de opp en rekke forskjellige logiske feil som ofte ble gjort av nybegynnere. I deres studium nevnte de at vanligste logiske feiler som oppstår, var ved flogged feilmelding fra interpreter: **”**Field Not Found”, ”Use of Non-Static Variable Inside the Static Method”, ”Type Mismatch”, ”Using a non-Initialised Variable”, ”Method Call with Wrong Arguments” og ”Method Name Not Found”.

Mistyding i programmering definerer Fleury (2000) i sitt studium som når studenter konstruerte “feil” regler ved misbruk av ”riktig” regler. Et eksempel på dette er at i Java kan dot (.) operator kan bare brukes til metoder og ikke til variabler. Dermed er ”feil” regel, bruken dot (.) operatør til variabler, misbruk av ”riktig” regel, bruken av dot (.) operatør til metoder. Mistyding var også et problem som ble forsket innenfor av forskere som Soloway (1986). Allerede tidlig påstod Soloway og senere Lister et al. (2004) at mistyding var ikke så utbredt likevel.

2.2.2 Problemløsningsmetoder

Problemløsning er kjernen av programutvikling. Når utviklerne tillærer seg programmeringsevner, møter de ofte vanskeligheter i å forstå basiske problemløsningskonsepter. De mangler innsikt innenfor problemløsningsstrategi og taktisk kunnskap. Dette er et kjent fenomen, og det er blitt utført mange studier om ueffektiv pedagogiske programmeringsinstruksjoner ((Dansereau, Collins, McDonald, Holley, Diekhoff, & Evans, 1979); (Ahmadzadeh, Elliman, & Higgins, 2005)).

Å forvente et effektivt og rasjonell løsning av et problem trengs det mye arbeid og kreativitet. Deek et al. (1997) definisjon av en problemløsningsstrategi er som følger: "*Før implementering av en løsning i et program trenger programmereren å utvikle kognitiv kunnskap til å begripe og trekke ut problemet og dets krav. For så deretter utforske og transformere problemet til en logisk form, design og utvikling. Deretter å teste løsningen med andre deler i programmet*".

Nedenfor beskriver jeg en kort oversikt over generelle strategier som ofte brukes til å løse problemer i programmering.

2.2.2.1 Abstraksjon

Abstraksjon er et kognitivt hjelpemiddel som er ingeniørenes svar mot blant annet kompleksivitet, ubegripelighet av programsystemer og evne til å etterse mange emner av programvareteknikk i forskjellige plan av detaljer i henhold til hensikten. Webster (1966) kategoriserte abstraksjon i flere grupper. Ut fra dem har Kramer (2007) silt ut to aktuelle definisjoner av abstraksjon som har mest betydning for ingeniører. Definisjonene er som følger:

1. *Fjerning av detaljer:*
   - Basert på følgende definisjoner:
     - Akten av henting eller fjerning av noe.
     - Akten eller prosessen av utelating med hensyn til en eller flere egenskaper av et kompleks objekt for å ekspedere andre.
   - Fordel:
     - Forenkling.
     - Fokus på oppmerksomhet.

2. *Generalisering:*
   - Basert på følgende definisjoner:
     - Prosessen av å formulere generelle konsepter på fjerning av felles egenskaper i en instans.
     - Et generelt konsept er formulert på ekstrahering av felles kjennetegn fra spesifikke eksempler.
   - Fordeler:
     - Identifisere felles kjernesak.

Flere forskere som ((Kramer & Hazzan, 2006); (Bourdoncle, 1993); (Hoffman & Strooper, 1995); (Webster, 1966); (Ghezzi, Jazayeri, & Mandrioli, 2002); (Verelst, 2005); (Gannon, Hamlet, & Mills, 1987); (Benjamin, Erraguntla, Delen, & Mayer, 1998)) har påpekt at abstraksjon er et av de

fundamentale prinsippene til program utviklere for å løse kompleksivitet. Fjerning av unødvendige detaljer er et selvfølgelig behov i utvikling og design av programmer. Å bringe inn krav som involverer identifisering av kritiske aspekter av miljøet og systemet, men overse det som er irrelevant. Generalisering av aspektet av abstraksjonen kan kjennes igjen innenfor programmering, som bruken av data abstraksjon med klasser i objekt orientert programmering (Arisholm & Sjoberg, 2004), eller generalisering med Abstrakt fortolkning (Bourdoncle, 1993) for program analysering, hvor det konkrete program domenet er mappet til en abstrakt domene for å få fram semantikken av det overlagte for program analysen.

Aspekter innenfor abstraksjon som er viktige er blant annet innkapsling, nivå av avabstraksjon, generalisering og klasse abstraktsjoner. Program modellering og analyse er det som har fått mest oppmerksomhet innenfor forskningen. Kramer (2007) påstod i sin studie at modellering er den viktigste teknikken for ingeniører. Modeller hjelper til med å forstå og analyse store og komplekse problemer, siden modeller er en forenkling av virkeligheten til å forfremme forståelser og begrunnelser.

France et al. (2007) Påpekte at en modell er en abstraksjon av noen aspekter av et system. Modeller er laget på forskjellige grunner, eks. presentere en menneskelig beskrivelse av noen aspekter innenfor et system eller å presentere informasjon i en form som kan analyseres mekanisk. De to kategoriserte modellene er som følger:

1. *Utviklingsmodeller (Development modells)*: disse er modeller av programmet på nivå av abstraksjon over kodenivået. Eks. kravet, arkitekt, implementasjon og anvendelse modeller.
2. *Kjørtidsmodeller (Runtime modells)*: disse modeller presentere overblikk av noen aspekter av et kjørbart system og er dermed abstraksjon av kjørtid fenomen.

Disse modellene kan være dynamiske. Fordeler ved bruk av modeller bidrar kosteffektiv og kontroll i forhold til forandring av programmet.

I Benjamin et al. (1998) studium mente de at abstraksjon kan deles inn i forskjellige nivåer av modeller som avgjør mengden av informasjon som er omfattet av modellen. Kvantiteten av informasjon i en modell minker med nivåer av abstraksjon. Dermed gir et lavnivå abstraksjonsmodell mer informasjon (detaljer) enn et høynivå abstraksjonsmodell. Brukeren bør være oppmerksom på behovet til å tenke på vilkår av forskjellige abstraksjonsnivåer og være i stand til å bevege seg mellom abstraksjonsnivåene (for eksempel å bevege seg fra globalt syn av systemet (høynivå) til lokale detaljer (lavnivå), og motsatt). Dette er ikke lett, og trenger en viss grad av oppmerksomhet og erfaring (for eksempel noen ganger blir utvikleren i høy eller lav nivå for lenge, mens problemet kunne løses direkte i et annet nivå). Benjamin et al. beskrev oppnåelse av nivå av abstraksjon i fire trinn: (1) Determinering et riktig nivå av abstraksjon. (2) Dekomponering/oppløsning. (3) Leggeopp/oppløsning. (4) integrasjon/harmonisering.

Abstraksjonens kognitive betydning er å løse kompleksivitet i et spesifikt stadium av problemløsningssituasjon. I Kramer et al. (2006) studie mente de at å tenke abstrakt er å kunne konsentrere seg på det essensielle særpreget av subjektet, og ignorere irrelevante detaljer. Abstraksjon er spesielt avgjørende i å løse komplekse problemer, for å få problemløseren til å tenke på vilkår av begrepsmessig ideer enn på vilkår av problemets detaljer.

2.2.2.2 Dekomposisjon

Evnen til å dekomponere et problem til håndterlige sub-komponenter er nødvendig i en kompleks problemløsningsoppgave. Teknikken for plan dekomponering er ofte kjent som ”splitt-og-hersk” metoden. Teknikken er som følgende: dekomponere et problem til mindre komponenter (sub-problemer), løse disse sub-problemene individuelt, for deretter å kombinere dem til en samlet løsning. Suksessen bak denne metoden er underlag for interaksjonen som kan forekomme mellom handlingene fra løsningen til forskjellige sub-problemene.

I Sebastian et al. (2006) studium nevnte de at dekomposisjoner har følgende prinsipper; prinsippene som brukes til å dele problemet til sub-problemene, resolusjon prosess brukt til å løse hver sub-problemet individuelt, og resolusjon teknikk brukt til å kombinere hele løsningen.

Fordelene med plan dekomposisjon er at sub-problemene er mye enklere enn original problemet og er dermed lettere å løse. Det er ikke nødvendig å designe en ny problemstilling for å løse et dekomponert problem; den samme teknikken som ble brukt til å løse et komplett problem kan gjenbrukes til å løse sub-problemene som stammer fra dekomponeringen, og dermed oppnå positive interaksjoner (ved oppnåelse av en informasjon som hjelper til å få tak i flere nødvendige informasjoner).

Ulempene til plan dekomponering er at det kan oppstå ved mange negative interaksjoner (ved oppnåelse av en informasjon som fører til sletting av andre nødvendige informasjon) istedenfor positive interaksjoner. Negative interaksjoner kan medføre til komplekse prosesser av plan combination, og må dermed finnes og fikses, noe som kan være mer tidskostbart enn problemet selv. Noe å merke seg er at når plan dekomponering er brukt, kan den helhetlige synet av problemet bli borte, noe som kan føre til degradering av plan kvalitet. Styrken i dekomponering er en teknikk som å muliggjøre og løse sub-problemene samtidig.

2.2.2.3 Bygg og fiks

Bygg og fiks metode har vært kjent lenge. Dette er en metode som studentene utviklet gjennom første året i ”introduksjon til programmering” kurset. Metoden fungerer som følgende: å løse problemer etter eliminering av feilmeldingene. Deek et al. (1997) påstod at alle studenter som har vært gjennom programeringskurs, har tillært seg kunnskap av bygg og fiks metoden. De beskrev at da studentene lærte programmering, ble de avhengig av rammeverkene i læringsprosessen, og ble vant til å tilrette seg i alle programmeringsspråk gjennom en kompilator. Når det er presentert et problem i et program, har studentene en tendens til å strekke seg for tastaturet og starte kodingen. Dette skaper en uvane for ”bygg og fiks” metode for problemløsning. Deek et al. konkluderte også med at denne metoden er en uvane som bør unngås. I deres forskning rapporterte de at nybegynnere hopper som regel inn i problemer, begrense selve problemets oversikt, og velger første-inntrykk løsninger uten å utforske problemets situasjon, mens eksperter generer en målbar kriteria, forbindelse av årsak og effekt, prosedyrer, som avgjør hvordan de går fram for å løse problemet. Arvanitis et al. (2001) var skeptisk til bygg of fikse metoden og hevdet at metoden ikke gir muligheter til å løse komplekser problemer.

2.2.2.4 En felles modell for problem løsning

Strategier for å løse et problem har allerede blitt diskutert i en lang periode. Siden 1910 har Dewey (1997) foreslått fire steg for problemløsning, og deretter foreslo Polya (1962) en velkjent problemløsning som består av fire steg: (1) forstå problemet, (2) konstruere en plan, (3) utføre planen og (4) se tilbake. Disse fire stegene er de mest brukte i allmenn problemløsningsanledninger. Andre problemløsningsstrategier som kom etter Polya er ofte lignende til Polya eller en forbedring i stegene. I 1999 arrangerte Deek et al. (1999) disse problemløsningsstrategiene og generaliserte dem inn i seks stadier:

1. Formulering av problemet (formulating the problem)
2. Planlegging av løsningen (planing the solution)
3. Konstruering av løsningen (designing the solution)
4. Oversettelse av løsningen (translation the solution)
5. Testing av løsningen (testing the solution)
6. Levering av løsningen (delivery the solution)

Deek et al. kalte denne modellen for ”en felles modell”. Denne modellen er en mer omfatende modell som eksplisitt henvender seg til kognitivt og aspekter innenfor programutvikling, altså en modell for programmering domene. De mener at å formere en løsning til et problem ikke kan bli gjort effektivt uten nødvendige problemløsningsteknikker. Denne modellen ble presentert av Deek et al. for å forenkle studien av programmering.

Formulering av problemet er å identifisere de aktuelle faktaene om problemet fra en raffinert problembeskrivelse og ignorere det uvesentlige. Dette stadiet involverer tre aktiviteter: forberedelse av problem beskrivelse, forberedelser av mental modell og strukturert problem representasjon.

Planlegging av løsningen består av tre aktiviteter som strategi oppdagelser, mål dekomposisjon og data modellering. Bruken av kunnskap, fakta og applikasjonens konsepter, teorier og prinsipper til å planlegge en løsning. Løsningen demonstreres ved skisseringene av nødvendige steg til å nå en løsning som løses lettere, relatert problem og ved å tegne kart og grafer visualisere løsningen. Målet er å utvikle en plan, kartlegge en potensiell løsning og bryte ned problemet inn i deler.

Konstruering av løsningen består av tre aktiviteter som organisering og raffinering, data/funksjon spesifikasjon og modul logikk spesifikasjon. Hoved aktiviteten i dette stadiet er syntese, som passer på reintergrasjon av beslektede komponenter i en sammenhengende helhet, rearrangere når det er nødvendig, etablerer forhold, og produserer nye og vel organiserte helhet som en mulig løsning av problemet.

Oversettelse av løsningen består av tre aktiviteter som implementasjon, integrasjon og diagnose av feil. Anvendelse i dette stadiet referer til praktisk evne til å bruke generell kunnskap av språkets syntaks og semantikk til å implementere en kodet løsning.

Testing av løsningen består av tre aktiviteter som kritisk analyse, evaluering og revurdering. Det gjelder å verifisere om løsning spesifikasjonen og resultater er overensstemmelse med problemets krav. Dette kan gjøres ved å utvikle test data, teste programmet ved bruk av denne dataen og eksaminere resultatet for nøyaktighet.

Levering av løsningen består av tre aktiviteter som dokumentasjon, presentasjon og utbredelse av forskjellige løsningsdeler i en organisert og forståelse form.

Deek et al. nevnte også at domenekunnskap, problemkunnskap og strategi kompetanse er de tre viktigste fundamenter for å fullstendig løse et problem.

2.2.3 Tidligere studier

En del forskninger innenfor problemløsning har blitt utført på å evaluere programmerernes problem og deres framgang til å løse problemer.

Innenfor studiene om ***problemmer*** som oppstår i programmeringen finnes en del gamle men fortsatt relevante studier fra Spohrer og Soloway (1986). I deres studium rapporterte de at de fleste av studentenes feil som forekom i programmering stadiet var resultat av ”plan composition problems”, vanskeligheter med å putte sammen delene av programmet, og ikke resultatet av ”construct-based problems”, mistyding av språkets konstrukter. Sphorer og Soloway beskrev disse plan komposisjonsproblemene i ni kategorier; ”Summarisation problem”, bare hoved funksjonen av en plan var regnet med, implementering og sekundær aspekter var ignorert, ”optimalsation problem”, optimaliseringsforsøk var upassende, ”previous-experience problem”, tidligere erfaringer passet dårlig inn, ”specialisation problem”, abstrakt planer var adaptert til spesifisert situasjoner, ”natural language problem”, upassende analogier var dratt inn fra naturlig språk, ”interpretation problem”, implisitt spesifikasjoner var utelatt, eller bare fylt inn når passende planer kunne enkelt gjenopprettes, ”boundary problem”, ved adapterte en plan til å spesifisere situasjoner begrensningspoeng var upassende, ”unexpeced case problem”, uvanlig, usannsynlig, og begrensningstilfeller var ikke regnet med, ”cogniive load problem”, mindre men betydelig del av planer var utelatt, eller plan interaksjoner oversett. Senere var det også mange flere studier som forsket innenfor studentenes vanskeligheter med programmering. Studier som Lister et al. (2004) der de fastslo at mange studenter kan ikke programmere på slutten av deres introduksjonskurs. En velkjent forklaring av dette var at studentene manglet evne til å løse problemer. Manglet evne til å behandle en problembeskrivelse, dekomponere problemet til mindre sub-problemer og deretter komponerer løsningene og implementerer dem til en komplett løsning. Lister et al. beskrev også forklaringen av dårlig problemløsning evne var på grunn av dårlig erkjennelse av både basis programmeringsprinsipper og evne til å systematisk utføre rutine programmeringsoppgaver. I Détienne (1997) studium påstod hun at hoved vanskelighetene som nybegynnere møter på var leddforbindelse mellom ”declarative” og ”procedural” aspekter av løsningen. Noen nybegynnere klarte ikke å fullføre dekomposisjon av store prosedyre inn til mindre funksjonelle komponenter, og de assosierte prosedyren som helheten av en enkel klasse. Hun nevnte også at nybegynnere hadde vanskeligheter i bruk av arv av statisk særpreg, og de hadde enda større vanskeligheter i bruk av arv i funksjonalitet. I 2007 utførte Karahasanovic et al. flere empiri studier om studentenes vanskeligheter under vedlikeholdningsprosessen av et medium stort OO system. I Karahasanovic et al. (2007) studium rapporterte de vanskelighetene utviklerne hadde i to kategorier; (i) forståelsen av applikasjonsstruktur og (ii) bruken av arv av funksjonalitet. Senere i 2007 rapporterte Karahasanovic og Thomas (2007) igjen vanskelighetene som oppstod for studentene mens de vedlikehold et ukjent objekt orientert system. Hoved vanskelighetene Karahasanovic og Thomas raporterte i denne empiriske forskningen var relatert til forståelsen av (1) programmets logikk, algoritmer, finne

påvirkning av forandringer, og (2) arv av funksjonalitet. I Vee et al. (200X) studium, studerte de studentenes feil og hvordan studentene har løst feilene som oppstod. I deres studie fokuserte de på kompileringsfeil. De rapporterte en rekke forskjellige feiltype som oppstod. Feil som ble nevnt var av type ”syntactical issue”, ”type errors”, ”feature call errors”, “rewrite instead of reuse”, “not following hints”, “language overlap”, “extra variables”, “assignment”, “queries”, “expression used as instruction”, “current”, “language of instruction” og “inheritance”. Selv om de mente at studiet deres har ikke kommet fram til noen konklusjon enda, kom de likevel fram til noen interresante konsepter om studentenes løsemønster av feilene som oppstod. De påstod at mange av studentene var meget konsistente på måten de løste problemer på. Når de hadde adoptert en spesiell metode, brukte de den om og om igjen. Vee et al. oppga et eksempel av ”backtracking”, hvor de nevnte at når studentene vil prøve ut noe, forandrer de det til noe annet, så å se hvordan det påvirker produksjonen. Deretter går de tilbake til det forrige svaret og så videre. Noen studenter vil lage mange forandringer på en gang, mens andre vil forandre på en ting om gangen.

Det er velkjent at ***abstraksjon*** er et hjelpemiddel for å løse komplekse problemer. I OrBach og Lavy (2004) studium nevnte de at programmering var et kompleks kongitiv aktivitet assosiert med programeringsmønster. Programmeringsmønsteret veiledet til riktig tenkemåte til problemløsning. De mente at abstraksjon var et fundamentalt konsept i programmering generelt, også innenfor objekt orientert programmering. Definisjonen av en klasse med dets relevante attributer og metoder var et basis krav av abstraksjon. Bruk av klassehierki med abstrakte klasser var regnet som komplekse abstraksjonsoppgaver. De kategoriserte kognitiv abstraksjonsaktiviter studentene brukte i deres løsninger for gitte problemmer som; (1) inklusjon av attributer i abstrakt klasse som ble definert, (2) inklusjon av attributer og implementerte metoder i denne abstrakte klassen og (3) løsninger om abstrakt klassen inkluderte attributer, implementert metoder og abstrakte metoder. Tidligere var det også forsket på mye innenfor abstraksjon. Allerede tidlig i 1986 mente Liskov og Guttag (1986) at abstraksjon kan uttrykkes på flere måter. For eksempel beskrev de abstraksjon i programutvikling som ”*en måte å gjøre dekomponering produktivt ved å forandre på nivå av detaljer som regnes med*”. Prosessen av abstraksjon kan sees som en applikasjon av mange-til-en mapping. Det tillater oss å glemme informasjon og konsekvensene å behandle ting som er annerledes som om de er det samme. Vi gjør dette i håp av å forenkle vår analyse ved å dele attributene som er relevant fra de som er ikke. I Verelst (2005) studium påstod han at bruken av abstraksjon ga fordeler til videreutvikling. Men å bygge abstrakte modeller er ikke en enkelt sak for nybegynnere. Det trengs både domene-spesifikk og metodologi ekspertise og tid. Det har også blitt utviklet mange metoder ved bruk av abstraksjon. Et eksempel er Program slicing. Rilling og Klemola (2003) beskrev program slicing metoden som et program redusering tenikk. Teknikken består av identifisering av kode som var relatert til en gitt funksjon eller variabel av interesse, ved å identifisere bare disse delene av det originale programmet som var relenvant til beregning av en bestemt funksjon eller output. Program slicing ga en forenklet versjon av det originale programmet ved å forbeholde en projeksjon av programmets semantikker. Program slicing er en form for abstraksjon av kode, og hjelper til med å identifisere kildekodesegmenter som er relevante for implementasjoner. Program slicing finnes ofte i programtesting, debugging og maintenance. Rilling og Klemola rapporterte at program slicing hjalp til å korte ned tiden for undersøkelse av kildekoder.

Å ***dekomponere*** et vanskelig problem inn i mindre problemer, er også et velkjent prinsipp til å løse komplekse problemer. Mange metoder tar i bruk av dette prinsippet. Metoder som blant annet plan programmering. Ebrarhimi og Shcweikert (2006) mente at en integrasjon av plan programmering

egnet seg til problemløsning, og bidro til øking av riktige implementasjoner. Plan, i denne sammenhengen, er en abstraksjon av et konsept, krav, programmeringskode eller et objekt. En plan hadde et mål eller en hensikt innenfor en bestemt tid og kontekst og kunne bli integrert i andre planer for å bygge en større plan. Programeringsplan var mal-lignende løsninger brukt i problemløsning. En plan, representerte i programmering, en serie av kode segmenter som tilsammen utgjorde en oppgave i å løse et problem. Dekomposisjon var også viktig for metoden program slicing. Gallagher og Lyle (1991) påstod at bruken av dekomposijon i program slicing kan medføre til en ny programvedlikeholdningsprosesssmodell som fjerne behovet for regresjon testing.

Av metoden ***bygg of fiks*** har Linn og Clancy (1992) studium viste de til spor av bygg og fiks metoden problemløsning. De mente at studentene ofte organiserte deres kunnskap i vilkår med språkets syntaks. Dette resulterte til en splittet og tungvint kunnskap organisering. Som resultat, har studentene vanskeligheter med oppsamlinger av algoritmer og ofte kom i noe uregelmessig prøvinger og feilinger ved å prøve å designe et dataprogram. Arvanitis et al. (2001) påstod i deres studium at objekt orienterte språk er sterkt støttet for bruk av data abstraksjon. I deres studium rapporterte de at 35 % av studenter lagde designer ved bruk av abstraksjon og dekomposisjon metoders i deres studium av software design problemer. Hele 85 % av studentene gjennomførte programdesign problemmer med ingen henvisning til noen av de disposisjonsforslagene, de brukte den såkalte ”bygg-og-fiks” metoden. Og 65 % av studentene kunne ikke skillet mellom syntaks og semantikk feil. Men så var dette snakk om program-in-the small, med andre ord veldig korte kodesnitter. I Prechelt et al. (2001) studium sammenlignet de vedlikholdningskarakteristisker av en rekke forskjellige design modeller. I deres studium konkluderte de at med mindre det var en klar grunn til å velge den enkle løsningen, så var det klokere å velge fleksibiliteten gitt av design modeller. Détienne (1997) påstod at OO design metoder hjelper software gjenbruk gjennom mekanismer av abstraksjon, encapsulation og inheritance.

Av forskninger i ***gjenbruk*** har Pohthong og Budgen (2001) definert to strategier for gjenbruk ved program design. Den første strategien beskrev de som ”*Element First*”, der designere begynte å identifiserte sannsynlige design elementer som var kandidater for gjenbruk i løsing av deres problemer, og deretter konstruerte formen for svaret rundt dette. Den andre strategien var ”*Framework First*”, der designere produserte en plan for å løse dette problemet, og deretter søkte etter komponenter som vil passe inn med resultat løsning framework. De påstod at nybegynnere sannsynligvis vil oppnå suksess ved bruk av Element First når de jobbet innenfor kontekst av ”reuse verbatim”. Lewis et al. (1991) evaluerte i deres studium om påvirkning av objekt orientert paradigmer har noe effekter på software gjenbruk. De konkluderte at OO paradigmer betraktelig improviserte produktivitet, altså en betydelig del av denne forbedringen var på grunn av effektene av gjenbruk. Gjenbruk uten hensyn til språk paradigmer improvisert produkt, språk forskjell var mye mer viktig når programmereren gjenbrukte enn ikke, og OO paradigm hadde en spesiell tiltrekning for gjenbruk. I Burkhardt og Détienne (1995) studium om gjenbruk rapporterte de at gjenbruk av komponenter forutsetter mer enn en tekstbase konstrueringsrepresentasjon av selve kildekomponenter.

Debugging er også et kjent tema innenfor problemløsning. Studier som Chmiel og Loui (2004) mente at debugging har en del å si for effektivisering av problemløsning. Studenter som fikk trening i debugging fant feil raskere og løste dem raskere. Debugging, lignende som problemløsning og skriving, er en viktig og komplisert evne som trenger gjentatt trening for å holde ved like. Chimel og

Loui rapporterte at gjennomgåelse av kode var viktig for å identifisere store deler av totale programdefekter. Programmerer som inspiserte kodene rettet defektene raskere enn de som ikke gjorde det. En rekke studier prøvde å finne årsaken til studentenes problemer med programering. Studier som Ahmadzadeh et al. (2005) studium. Ahmadzadeh et al. mente at årsaken til studentenes problemer med programmering lå i prosessen da studentene lærte å programmere. De mente at det var vanskelig å finne en effektiv metode for læring som var passelig for alle studenter. De kategoriserte studentene i to kategorier: *good programmers* og *weak programmers*. Det som skilte gode programmerer fra de svake var; (1) kunnskaper om det påtenkte programmet, (2) kunnskaper om det faktiske programmet, (3) bruken av debugg metoder (filtrering metoder, eksempel bruken av print stmt), og (5) feilen i seg selv. I deres eksperiment fokuserte de om mønster av kompileringsfeil og debugging av logiske feil. Selv om Ahmadzadeh et al. mente at problemene ligger i pedagogikken, var det også mange forskere som mente at problemet ikke er på grunnet av pedagogikken. Et eksempel er Eliasson et al. (Eliasson, Westin, & Nordström, 2006) studium hvor de mente at studentenes selvsikkerhet spilte en rolle om hvor bra de løste problemene.

Stadig nye studier og metoder var konstruert til å løse problemer. Et eksempel er Kuro et al. (2004) studium. I Kuro et al. sin forskning mente de at problemer kan enklere løses ved bruk av graf og matrise. I deres studie presenterte Kuro et al. to problem metaforer; (1) Problem graf (problem graph) bruker grafisk struktur til å illustrere et problem. Det består av konsepter og linker. (2) Problem matrise (problem matrix) fokuserte på manipuleringer av konsepter og relasjoner som kan brukes til å løse problemmer. Problem matrise definerte mulige manipuleringsmåter av relasjoner og konspeter av et problem. I tilegg til problemløsningsmetoder har en rekke forskere prøvd ut nye analyseringsmetoder for å kartlegge novises oppførsel for bedre forståelse av deres problemmer. Et eksempel er Hundhausen et al. (2006) studium, der de påstod at video analysering var en ny måte å kartlegge novise sine framgangsmåter på.

2.2.3.1 Oppsummering

Studiene ovenfor viser tidligere forskninger i sammenheng med utviklernes problemmer og løsninger de støttet på ved programering. Studiene var på område av program problemløsning og undersøkte forskjellige årsaker som påvirker evnen til å løse problemer og strategiene bak det. Det var en del relevante studier av studentenes kunnskaper om programmering og problemløsning ((Vee, Meyer, & Mannock, 200X); (Hundhausen, Brown, Farley, & Skarpas, 2006); (Lister, et al., 2004); (McCracken, et al., 2001)). Problemene studentene møtet ved utvikling av programmer (Spohrer & Soloway, 1986) og vanskeligheter de møter i vedlikeholdningsprosesser ((Karahasanović & Thomas, 2007); (Karahasanović, Levine, & Thomas, 2007)). Forskninger om hvordan ekspertene løste disse problemene (Linn & Clancy, 1992), generelle problemløsningsmodeller (Deek, Turoff, & McHugh, 1999) og studentenes bruk av problemløsningsmetoder (Arvanitis T. N., Todd, Gibb, & Orihashi, 2001).

To generelle løsningsmetoder som var meget kjent og forsket på mye innenfor vitenskap er abstraksjon ((Larsen & Naumann, 1992); (Verelst, 2005)) og dekomposisjon (Gallagher & Lyle, 1991). Enven til å forstå, abstrahere et problem og krav, utforske og transformere et problem til en logisk form, design, utvikle, og teste svaret mot problemet er også viktige kunnskaper i problemløsning (Deek, Turoff, & McHugh, 1999).

2.3 Oppsummering

En rekke studier har blitt gjort i området av problemer og problemløsning innenfor OO systemer. Tidligere studier har også fokusert på hvordan man løser vanskeligheter deltakerne møtet i en vedlikeholdningsprosess og deres problemløsningsstrategier. Koenemann og Robertson (1991) mente at program forståelse var en kompleks problemløsnings prosess. I deres studie foreslo de at program forståelse er best forstått som målorientert, hypotese-drevet problemløsning prosess. Prosessen av analysering av et problem, komme på en løsning, å fremstille løsning i det høynivå programmeringsspråk kan sees på som en innviklet form for modelling og kan dermed argumentere at program utvikling er hovedsakelig en modell-basert problem løsning aktivitet (France & Rumpe, 2007).

Abstraksjon, dekomposisjon og bygg og fiks metodene har blitt nevnt i en rekke studier for utvikling av metoder, men ikke mange av dem har blitt forsket på innenfor direkte bruk av disse problemløsningsmetodene til å løse vanskeligheter som oppstod i en vedlikeholdningsprosess.

I denne studien har jeg til hensikt å forske innenfor selve bruken av abstraksjon, dekomposisjon og bygg og fiks metodene for å løse vanskeligheter som oppstod i en vedlikeholdningsprosess, og deretter bruken av problemløsningsstrategier ved implementering og modifisering av systemet.

3 Metode

I denne oppgaven analyserte jeg data fra et kontrollert eksperiment som ble gjennomført av Amela Karahasanovic, Richard Thomas og Junisilver Taij i UWA i 2005. For å gi en fullstendig forståelse av oppgaven velger jeg å ta med beskrivelsen av eksperimentet. Beskrivelsen av eksperimentet er tatt fra Taij (2005) studium.

3.1 Design av eksperimentet

Eksperimentet var av randomiseringsmodell. Deltakende ble tilfeldig tildelt til i to grupper med forskjellige behandling. 17 deltaker var tildelt på Feedback Collection gruppe og resterende 17 deltaker på Control Silent gruppe. På hver gruppe var det minst en observatør tilstedet med hensyn til eventuelt tekniske problemer. Observatørene skulle ikke ha noe interaksjon med deltakerne.

3.2 Behandling

3.2.1 Feedback-Collection gruppen

I denne gruppen dukket det opp et skjermbilde hvert 15 minutt med spørsmål: "Hva tenker du på nå?". Skjermbildet var tilgjengelig i 2 minutter for besvarelse. Besvarelsen ble automatisk lagret når skjermbildet forsvinner.

3.2.2 Control-Silent gruppen

Deltakerne i denne gruppen var uforstyrret gjennom eksperimentet.

3.3 Observatørene

2 observatør og 1 teknisk støtte var involvert i gjennomføring av eksperimentet. Observatørene ble trent i den eksperimentelle prosedyre og støtteverktøyer. Alle observatørene var utstyrt med skriftig instruksjoner.

3.4 Deltakere og innstillinger

3.4.1 Innstillinger

Hovedeksperimentet bestod av 34 studenter i Faculty of Engineering, Computing and Mathematics, School of Computer Science og Software Engineering i University of Western Australia (UWA). Eksperimentet var organisert å 7 separate seksjoner på 7 forskjellige dager. Seksjonene foregikk på skolens data laboratorium (rom 2.1) fra kl 9.15 til rundt kl 16.00 og hver seksjon bestod av 2 til 7 deltakere. Lunsj var inkludert og deltakere har lunsj pause fra kl 12.30 til kl 13.00.

3.4.2 Deltakere

Meste parten av deltakere bestod av tredje og noen få fjerde årstudenter i informatikk studiet (Computer Science). Alle deltakere er menn fra mellom 19-47 alder, med et gjennomsnitt alder på 22 år. Studentene har hatt grunnleggende objekt orientert og Java programmering kunnskaper gjennom universitetes kurser. JBuilder 9 (Borland Software Corporation) (JBuilder, 2008) ble brukt som programmeringsmiljø til eksperimentet.

Deltakere ble spurt om å avsette 1 dag i semester fri uke til å delta på eksperimentet. Ingen forfall fra eksperimentet, men 1 deltaker hadde tekniske problemer med datamaskinen og resultatet til personen ble utelukket.

3.5 Framgangsmåten

Eksperimentet ble introdusert til deltakerne gjennom universitetets email system. Emailen bestod av 2 vedlegger, et informasjonsark og et samstykkeark, ble sendt til studenter, med invitasjon om å delta i et programeringsvitenskapingeniør eksperiment for et gitt honorar. Det understreket at deltakelse i eksperimentet er frivillig og er ikke knyttet til Universitetet studium. Studentene som var interessert ble spurt om å fylle ut navnet, studentnummer, email adresse og datoen av deres siste eksamen. Studentene fikk stort sett tildelt time tilfeldig etter deres siste eksamen.

En pilot-test av eksperimentets materiale, organisering og ytelse av støtteverktøy ble utført en uke før gjennomføring av hovedeksperimentet. Anledningen var nødvendig for å avgjøre om det trengtes noen forandringer til materialet, organiseringen og verktøyene. Ingen forandringer ble gjort i hovedeksperimentet etter pilot-testen.

På informasjonsmøtene før hver seksjon ble deltakere informert om eksperimentets formål og framgangsmåte. Framgangsmåte av type av oppgaver, antall av oppgaver, anbefalt tidsbruk av oppgaver, forventet ferdig tid, programmeringsmiljø som ble brukt og om webverktøyet SESE. Deltakere ble også informert om at en logging verktøy som vil logge all deres handling og at de har lov til å trekke seg fra eksperimentet når som helst.

Deltakere måtte signere under på 2 kopier av et samtykke skjema. Den ene ble samlet inn til forskningsgruppen og den andre beholdt deltakeren. Avtalen erkjente deltakerners anonymitet og konfidensialitet er beskyttet og garantere at informasjon om eksperimentet holdes hemmelig under og etter eksperimentet. Etter fullføring av avtalen får hver deltaker et unikt brukernavn og passord for videre bruk i eksperimentet.

Følgende oppgaver skulle deltakerne gjøre var: skrivetest, treningsoppgave, kalibreringsoppgave i taushet (silent condition), 30 minutters øvelse av tilbakemelding innsamling skjerm for feedback-collection gruppen, 3 forandringsoppgave og det hele avsluttet med gruppe intervju etter at deltakerne ble ferdig med oppgavene (oppgavene er beskrevet i en annen seksjon).

3.6 Data innsamling og støtte verktøy

Et web-basert verktøy, Simula Eksperiment Support Environment (SESE) (Arisholm & Sjøberg, 2002), ble brukt til logistikk support i eksperimentet. Gjennom hele eksperimentet brukte deltakere SESE til

å svare på spørreskjema, nedlasting av oppgaver og kildekoder, laste opp deres løsninger og forsyne med tilbakemeldinger (gjelder kun FC gruppen). SESE logger også startid og sluttid.

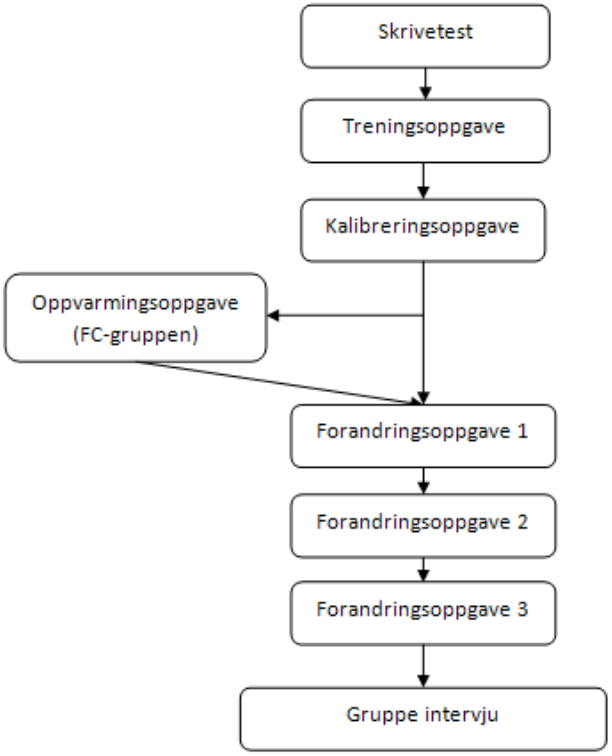
GRUMPS-Lite software (Thomas, 2005) logger hendelsene av tasttrykk (keystroke), mus klikk (mouse click) og vindu fokus med millisekunder i tidsintervalet.

JBuilder 9 (Borland Software Corporation) (JBuilder, 2008) ble brukt som programmeringsmiljø for eksperimentet. Observatørene lagde notater av spørsmålene som ble spurt av deltakere og problemer som oppstod under eksperimentet, og gruppe intervjuene ble audio-lagret for hver seksjon.

Eksperimental materialer i SESE var originalt skrevet på norsk som ble oversatt til engelsk med hensyn til språkbruk i UWA. Oversettelsen ble gjort av to masterstudenter, med hjelp av Taij (2005) for eventuell skrivefeil og misforståelse. Data ressurser ble også konfigurert av Taij for å redegjøre eksperimentet.

3.7 Oppgaver

Oppgavene i eksperimentet er utformet slik at deltakerne vil rask tilpasse seg miljøet. Oppgavene har forskjellige vanskelighetsgrader og de var ordnet etter kompleksiviteten. Som følge vil deltakeren bevege seg fra en oppgave til den neste, ettersom løsningen til den første oppgaven vil hjelpe til å løse den neste oppgave. Figur 1 viser hvordan eksperimentet er satt opp. Den første oppgaven var skrivetest, så trenings- og kalibreringsoppgave og deretter implementering av 3 forandringsoppgaver. Nedenfor beskriver oppgavene i mer detaljer.



Figur 1. Eksperimentet

3.7.1 Skrivetest

Deltakerne ble spurt om å taste en blokk av tekster innpå en notisblokk og lagre filen på en bestem mappe. Deretter skulle hver deltaker fylle inn bakgrunnen sin i et spørreskjema via SESE, spørreskjemaet spurte blant annet etter alder, utdanning og programmeringsdyktighet.

3.7.2 Treningsoppgave

Hensikten med treningsoppgaven er å orientere deltakerne i SESE og eksperimentetsmiljøet. Treningsoppgaven innebærte å skrive et program som leser et arbitrært nummer av tekst linjer (string) og lagret hver string inn i en Vektor. Når brukeren presser ”Enter”, skal programmet printe ut hvor mange string brukeren legget inn, å fremvise stringene i motsatt rekkefølge på skjermen.

Treningsoppgaven er et veldig lite Java program utviklet av Arisholm og Sjøberg (2001), som består av kun 20 LOC. Prøve løsning var lagt med oppgaven for å begrense tidsbruk på oppgaven.

3.7.3 Kalibreringsoppgave

Formål med kalibreringsoppgaven er for å fastslå programmeringsdyktigheten til hver av deltakere. Kalibreringsoppgaven omfatter å legge til ny funksjonalitet i et eksisterende Java kode. Applikasjonen er et lite Java program som består av omtrent 400 LOC med 7 Java klasser. Kalibreringsoppgaven var utviklet av Arisholm og Sjøberg (2001).

3.7.4 Oppvarmingsoppgave

Oppvarmingsoppgaver er kun for FC-gruppen. Oppvarminsoppgaver består av øvelser for deltakerne å gjøre seg kjent med å verbalisere tankene sine for Tilbakemelding Innsamling metoden. Øvelsene er utviklet av Masterstudenter i Oslo (Karahasanovic, et al., 2005). Denne treningen varer i 30 minutter.

3.7.5 Forandringsoppgave

Hoved oppgaven i eksperimentet var å modifisere et bibliotek applikasjon system utviklet av Eriksson og Penker (1997). Kravet for deltakerne var å implementere 3 forandringer i det eksisterende bibliotek applikasjon systemet. Forandringsoppgavene stiger i vanskelighetsnivå fra 1, 2 til 3. Applikasjonen er et Java program med 3600 LOC fordelt på 26 Java klasser. Grunnen for valget av systemet var under forutsetning at studentene vil være kjent med applikasjons domene. Hver deltaker var utstyrt med en komplett dokumentasjon for Jbuilder og Biblioteket Applikasjon System. Deltakere fikk også tilgang til online Java API dokumentasjon.

Deltakere kom fra forskjellige bakgrunner med forskjellige programmeringsdyktighet. Resultat ble da at noen ble ferdig med oppgavene raskere enn andre. Forandringsoppgave 3 var den mest kompliserte oppgaven og var designet for å sørge for at ingen deltaker klarer å fullføre hele eksperimentet før tiden går ut. Denne oppgaven var ikke inkludert i noen analyse. Oppgaveteksten understreket å legge prioritet mot kvalitet løsninger framfor å kutte ned implementeringstid for å rekke alt.

3.8 Gruppe intervju

På slutten av eksperimentet ble det ført et gruppe intervju. Intervjuets mening var å få fram deltakernes synspunkt. Gruppe intervjuene ble valgt ut av Karashanovic på grunnlag av forskjellige vurderinger, slik at det ble mest praktisk og kostnad-effektivt med hensyn til stort antall deltakere. Intervjuene gir også informasjon med hensyn til følelser, reaksjoner og meninger til deltakerne.

Intervjuene var semistrukturert med en blanding av åpne og spesifikke spørsmål, for å gi rom for uforutsigbar informasjon. Alle deltakerne som hadde eksperimentet samme dag var forventet å være med på samme gruppeintervju. I tilfelle hvor 1 eller 2 deltakerne som ble ferdig tidligere enn andre,

ble disse intervjuet individuelt. 1 intervjuer og 1 observator styrte intervjuet. Intervjueren stiller spørsmålene og observator noterte ned det som ble sagt samt bidrar til tilleggsspørsmål.

3.9 Analyse modell

Tid og riktighet er 2 nødvendige variabler for måling av effektivitet og ytelse.

Tid ble logget ved hjelp av GRUMPs verktøyet, og riktighet er basert på kvalitet av løsningen. Ved hver opplastede løsning ble det gitt poeng på basis av fullstendighet. Tabellen nedenfor viser poeng skalert av fullstendighet.

Riktighet	Poeng
Fullstendig løsning	5
Løsning med litt små kosmetikk feil	4
Løsning med syntaks feil	3
Løsning med logisk feil	2
Løsning med kompileringsfeil eller delvis implementert	1
Ikke prøvd	0

Tabell 2. Tatt fra Taij, 2005. Bedømmelse av besvarelser på oppgavene

Bedømmelse ble gjort av en Ph.D student fra UWA. Hun fikk utgitt oppgave spesifikasjon, en fungerende løsning og retningslinje for angivelse av poeng. Hun var ikke involvert i eksperimentet eller lærer i emnet. Gjennom rettingen ble all løsningen kompilert, kjørt, og gjennomtestet for funksjonalitet. Kilde kode ble også undersøkt manuelt nøye for leting av årsak til problemer og syntaks feil.

4 Analyse

Dette kapitlet beskriver analysen av data innsamlingen og data forberedelsen.

4.1 Bakgrunn

Å forstå kognitive vanskeligheter som studenter har mens de vedlikeholder OO systemer, er en forutsetning for forbedring av deres universitetutdanning, og til å forberede dem for arbeidslivet. I Karahasanovic et al. (2007) studium har de utforsket strategier og vanskeligheter utviklerne hadde da de utførte vedlikeholdningsoppgaver i et ukjent OO program. Deltakerne var studenter i Univiersitet i Oslo og Høgskolen i Oslo. De rapporterte vanskelighetene utviklerne hadde i to kategorier; (i) forståelse av applikasjonsstruktur og (ii) bruk av arv av funksjonalitet. Senere i 2007, fra samme eksperiment (kapitel 5), har Karahasanovic og Thomas (2007) rapportert vanskelighetene som oppstod for studentene i Universitet i Western Australia mens de vedlikeholdt et medium stort og ukjent OO system. Hovedvanskelighetene Karahasanovic og Thomas rapporterte i denne forskningen var relatert til forståelse av programmets logikk, algoritmer, finne påvirkning av forandringer, og arv av funksjonalitet. Av disse to studiene nevnte Karahasanovic og Thomas at vanskelighetene var kategorisert innenfor en raffinering av modellen til von Mayrhauser og Vans (1995). Tidligere har Von Mayrhauser og Vans antydet at vanskeligheter og feil i forståelsesprosessen forekommer av mangel på enten (a) generell kunnskap, som er uavhengig av spesifikk software applikasjon som programmerer prøver å forstå, eller (b) softwarespesifikk kunnskap, som representerer deres nivå av forståelser av software applikasjon. Av vanskelighetene som oppstod identifiserte Karahasanovic et al. følgende sub-kategorier til generell kunnskap: vanskeligheter angående program logikk, grafisk bruker grensesnitt (GUI), objekt orientert programmering, algoritmer og programmeringsmiljø. Av vanskeligheter som var forårsaket av spesifikk kunnskap i følgende sub-kategorier: vanskeligheter angående GUI, OO programmering og testing prosedyre. Tabell 3 gir en oversikt over vanskelighetene. Tabellen er tatt fra Karahasanovic og Thomas (2007) studiet.

Karahasanovic og Thomas (2007) rapporterte at to generelle vanskeligheter som hindret deltakerne fra å fullføre oppgaven eller utgjorde en betydelig forsinkelse, de var relatert til program logikk (23 tilfeller) og algoritmer (10 tilfeller). Innenfor program logikk (kategori 1.1 i tabell 3) var det to typer av problemer som oppstod; det ene var relatert til en ”if-else” statement. Statementen var for implementering av søking av bøker ved tittel, forfatter eller ISBN. De påstod at kanskje fordi deltakerne tolket oppgaven som en ren tekstredigeringsoppgave, og prøvde dermed ikke å forstå logikken av programmet. Dette gjorde at deltakerne enten fjernet for mye eller for lite av denne statementet, og resultatet fungerte ikke ordentlig. Den andre vanskeligheten var forårsaket av kunnskap av algoritmer eller applikasjonens domene (kategori 1.4 i tabell 3). Oppgaven var relatert til å kalkulere utløpsdato av lån. De rapporterte at deltakerne synte at denne oppgaven var vanskelig. Vanligvis vil man bruke bibliotekklasser for forskjellig konvertering og kalkulering, derfor kan det kanskje være vanskelig å lage egne utregninger selv for deltakerne. Ti deltakerer mislykket i å programmere riktig løsning eller prøvde ikke å kalkulere i det hele tatt.

Det var rapportert to vanskeligheter med GUI som ofte oppstod. Den ene var relatert til å legge til eller å fjerne fra GUI komponenter (kategori 2.1.2, 17 tilfelle, i tabell 3) og den andre var fjerne tekst (label) deklarasjoner (kategori 2.1.3, 19 tilfelle, i tabell 3). De rapporterte at deltakerne enten glemte å legge til eller fjerne forskjellige komponenter fra GUI, som radio knapper eller tekstfelter eller

mislyktes de å fjerne alle label deklarasjoner av ISBN i oppgave 1. Selv om det var ingen effekter av å ha igjen noen deklarasjoner av ISBN, men i oppgaveteksten var skrevet at alle referanser til ISBN skulle fjernes.

Karahasanovic og Thomas rapporterte to spesifikke vanskeligheter som hindret deltakerne fra å fullføre oppgaven eller utgjorde en betydelig forsinkelse, de var relatert til å finne innvirkninger av klassene (kategori 2.2.2, 14 tilfeller) og arv funksjonalitetens (kategori 2.2.4, 12 tilfellet). De rapporterte at deltakerne hadde vanskeligheter med å gjennomføre forandringsoppgaver, fordi for forandringsoppgaver trenger man å forstå strukturen av applikasjonen, forståelse av relasjoner mellom klassene og finne de klassene som ble påvirket av forandringene. De rapporterte at deltakerne trenger også å forstå arv av funksjonaliteten for å løse oppgavene. Klasser i applikasjonsbiblioteket som trenges å holde persistent måtte arve en abstrakt klasse, og mislykkethet av å holde data persistent antydet det at dette var vanskelig for deltakerne.

Det var også rapportert at deltakerne hadde lite problemer med vanskeligheter av attributter og metoder i feil klasser (kategori 2.2.2.2-3, totalt 2 tilfelle) eller fjerning av disse (kategori 2.2.3.1, totalt 3 tilfelle). De konkluderte med dette var kanskje årsaken av forståelse av det fundamentale i klassene, objektene og metodene i Java programmering. BlueJ miljø, ble brukt for denne seksjonen, er veldig bra for å illustrere disse konseptene.

Av disse vanskelighetene som Karahasanovic og Thomas (2007) rapporterte, skal jeg analysere hvordan studentene kom seg gjennom disse vanskelighetene. Analysering av hvordan de har tatt i bruk av generelle problemløsningsmetoder, abstraksjon og dekomposisjon, til å løse disse oppgavene, eller hadde de simpelt hoppet rett inn i kode og prøver å løse dem.

	Task1	Task2	Task3
1 General			
1.1 Program logic	20	3	
1.2 GUI			
1.2.1 Forgot to expand the window		6	
1.2.2 Little experience with GUI programming	2	1	
1.3 Object-oriented programming			
1.3.1 Initialise objects			2
1.3.2 Instantiate a class			1
1.3.3 Understand and use a Java API class			2
1.3.4 Reuse of methods		2	
1.4. Algorithms			10
1.5 Programming environment	1		1
2 Specific			
2.1 GUI			
2.1.1 Changing interface		1	
2.1.2 Adding or removing GUI components	10	4	3
2.1.3 Removing label declarations	19		
2.2 OO comprehension and programming			
2.2.1 Overall program structure	2	2	
2.2.2 Impacts on classes	1	10	3
2.2.2.1 Self-reported problems			
2.2.2.2 Attributes in the wrong class			2
2.2.2.3 Methods in the wrong class			
2.2.3 Impacts within class	1		
2.2.3.1 Removing variables and methods	3		
2.2.4 Inherited functionality	2	6	4
2.3 Testing procedure	1	1	1

Tabell 3. Tatt fra Karahasanovic et al. (Karahasanović & Thomas, 2007). Tabellen viser oversikt over vanskeligheter. Vanskelighetene er presentert bare en gang per deltaker per oppgave (se kapitel 5).

4.2 Analyse modell

Jeg skal analysere hvordan deltakerne har kommet seg gjennom de vanskelighetene som Karahasanovic og Thomas (2007) beskrev i studiet sitt da de løste oppgavene i eksperimentet. Jeg skal også analysere om deltakerne tar i bruk noen form for problemløsningsstrategi for løsing av oppgavene.

For å svare på studie spørsmålene ovenfor analyserte jeg kvalitativ data samlet med Feedback collection.

4.2.1 Problemløsningsmetode

Abstraksjon og dekomposisjon er to velkjente generelle metoder, brukt av ingeniører for å løse komplekse problemer. Samtidig er bygg og fiks også en meget kjent metode brukt av utviklere for å løse problemer. I dette studiet skal jeg analysere hvilket av disse metodene deltakerne har tatt i bruk for å løse vanskelighetene som Karahasanovic og Thomas (2007) rapporterte.

I denne analysen har vi følgende definisjoner:

- *Abstraksjon* er når en deltaker løser et problem ved hjelp av fjerning av unødvendige detaljer, bringe inn nye krav som involverer identifisering av kritiske aspekter av miljøet og system, mens overser det som er irrelevant.
- *Dekomposisjon* er når en deltaker løser et problem ved å dele et problem til mindre sub-problemer så løser sub-problemene individuelt og deretter kombinere sub-problemene til en samlet løsning.
- *Bygg og fiks* er når deltaker løse et problem ved å hoppe rett in problemet, så velge første inntrykk løsninger uten å utforske problemet situasjon.

Data fra feedback-collection i eksperimentet har blitt brukt for å forklare og validere deltakernes handlinger. Data av feedback-collection er kategorisert i henhold til koding skjemaet i kapittel 4.2.3.1.

For å identifisere om deltakerne har tatt i bruk metoden abstraksjon, dekomposisjon eller bygg og fiks for å løse vanskelighetene de hadde i vedlikeholdningsprosessen, fokuserte jeg på informasjoner av deltakernes oppgavehandlinger (punkt 3.1 i koding skjema) og deltakernes problemer med oppgavene (punkt 3.3 i koding skjema) i koding skjemaet.

Jeg obseverte at deltakerne ved bruk av metoden abstraksjon utførte lesing av dokumentasjon (punkt 3.1.1 i koding skjema), handling på koden (punkt 3.1.2 i koding skjema) og kjøring av programmet (punkt 3.1.3 i koding skjema). Deltakerne som brukte metoden bygg og fiks har ikke utført lesing av dokumentasjon, men gikk rett på handling på koden og kjøring av programmet, og deltakerne som brukte metoden dekomposisjon utførte for det meste bare innenfor handling på koden.

4.2.2 Problemløsningsstrategi

Problemløsningstrategi er en viktig prosess for å fullstendig løse et problem. Deek at al. (1999) utviklet et sekssteg modell for problemløsning mot programmeringsdomene. I denne oppgaven skal jeg analysere om deltakerne har tatt i bruk noen av de seks stegene for å løse oppgavene i eksperimentet.

De seks stegene har følgende definisjoner i denne analysen:

1. *Formulering av problemet* er om deltakerne utviklet en forståelse av problemet og identifiserte aktuelle fakta om problemet.
2. *Planlegging av løsningen* er om deltakerne kartla løsningen og brøt ned problemet inn i mindre deler.
3. *Konstruering av løsningen* er om deltaker rearrangerte, etablerte og produserte forholder for løsningen.
4. *Oversettelse av løsningen* er om deltakerne implementerte, integrerte og debugget av løsningen.
5. *Testing av løsningen* er om deltakerne verifiserte løsningen, eventuelt med egne data.
6. *(Levering av løsningen* er om deltakerne dokumenterte og presenterte løsningen.)

Det siste steget av strategien ser jeg bort i fra ettersom det ikke er et krav i oppgavene, eller har noen påvirkning i kvaliteten av løsningene. Det er også ikke nødvendigvis for deltaker å ta i bruk alle disse prinsippene for å utvikle en problemløsningstrategi. Data fra feedback-collection i eksperimentet har blitt brukt til å analysere deltakernes bruk av disse seks steg definisjonene i prosessen for å løse oppgavene, og hvor godt de løser dem. Data av feedback-collection er kategorisert i henhold til koding skjemaet i kapittel 4.2.3.1.

For å identifisere om deltakerne har tatt i bruk de seks stegene ovenfor, fokuserte jeg på informasjoner av deltakernes oppgavehandlinger (punkt 3.1 i koding skjema), deltakernes planlegging, strategier og bemerkninger (punkt 3.2 i koding skjema), og deltakernes forståelses av oppgavene (punkt 3.3 i koding skjema).

Jeg obseverte at deltakerne ved bruk av steg 1 gjorde seg forstått av oppgaven (punkt 3.3 i koding skjema) og leste dokumentasjonen (punkt 3.1.1 i koding skjema). Deltakerne som gjorde steg 2 utførte planlegging, strategier og bemerkninger av oppgaven (punkt 3.2 i koding skjema), og leste dokumentasjonen. De som gjorde steg 3 utførte også planlegging, strategier og bemerkninger av oppgaven, og leste dokumentasjonen, men i tillegg utførte de også handling på koden (punkt 3.1.2 i koding skjema). Av deltakerne som gjorde steg 4 og steg 5 utførte handling på koden og kjøring av programmet (punkt 3.1.3 i koding skjema).

4.2.3 Data fra Feedback-Collection

Data fra feedback-collection gir en nærmere innsikt i deltakernes handling. Handlinger om hvordan de løste oppgavene, hvorfor de valgte å løse oppgaven slik og andre faktorer som kan påvirke deltakernes løsninger. Jeg har brukt data fra feedback-collection for å kartlegge hvordan deltakerne løste problemer som oppstod da de løste oppgavene, og om de har brukt noen form for strategi til å konfrontere dette.

Nedenfor i kapittelet 4.2.3.1 viser jeg et koding skjema som brukes i denne forskningen for å kategoriserte deltakernes handlinger. Lengre ned i kapittel 4.2.3.2 viser jeg noen eksempler av deltakernes tilbakemeldinger i feedback-collection i samsvar med analysen av problemløsningsmetoder av vanskeligheter som oppstår i en vedlikeholdningsprosess, og strategier for å løse vedlikeholdningsoppgavene gitt i eksperimentet.

4.2.3.1 Koding skjema

Data av Feedback-collection er verdifult for innsamling av kvalitativ data om deltakernes arbeid i vedlikeholdningsprosessen (Karahasanovic, et al., 2005). Feedback-collection hjelper til å blant annet validere deltakernes årsak for tidsbruk, prosessen for å tilpasse seg, kilder for variasjoner av programmeringsprestasjonen og riktighet av løsningen.

Tabell 5 viser koding skjemaet kategorisert av informasjoner i feedback-collection. Dette koding skjemaet er utdypet av malen gitt i Karahasanovic et al. (2005) studium, men med noen forandringer med hensyn til data av feedback-collection i eksperimentet som brukes i denne oppgaven. I fremgangsmåten kan en segment tilhøre mer enn en kategori.

	Kode	**Eksempel**
1	Eksperiment kontekst	
1.1	Pause og forstyrrelser	"I took a toilet break"
1.2	Bakgrunn kunnskap	"Its difficulty using jbuilder since I am unfamiliar with it"
1.3	Eksperiment material	"One of the task filenames didnt match the information on the webpage"
1.4	Støtte verktøyer	"I went downstairs and had the feedback window explained to me"
2	Deltakernes oppfatninger	
2.1	Stress	"It is frustrating that this has to be done more than once"
2.3	Generelle bemerkninger	"Everything is fine"
3	Eksperiment utførelser	
3.1	Oppgavehandlinger	
3.1.1	Lese dokumentasjon	"I am just reading over the UML and Usermanual"
3.1.2	Handling på kode (read, edit, search, compile)	"I have found and removed all references to ISBN"
3.1.3	Kjøre applikasjonen	"I periodically run the application to check that the things that I expect are happening"
3.2	Planlegging, strategi og bemerkning	"My strategy is to look for each file corsponding to each gui and update it to include the email field"
3.3	Forståelse og problemer	"I am confused about the purpose of the Persistent class"

Tabell 5: Koding skjema

4.2.3.2 Ekspempler fra feedback-collection

I dette kapittelet viser jeg noen eksempler av deltakerne forklaringer i feedback-collection i forhold til metodene de brukte for å løse problemene som Karahasanovic og Thomas (2007) kategoriserte, og hvordan de har løst vedlikeholdningsoppavene gitt i dette eksperimentet.

Dette er noen eksempler av forklaringene fra deltakerne for problemløsningsmetode *abstraksjon*:

> *"I've located two dialogues where the ISDN number appears and made a note of that. I've just read the Library.pdf and think I've identified the .java files where the ISDN appears or are used."*
>
> *"The UML diagram helps to find where exactly one should do it. The code is quite extensive so…"*
>
> *"I started working from the bottom up, looking for references to ISBN numbers. I have followed the flow of logic and commented bits out as appropriate."*

Dette er noen eksempler av forklaringer fra deltakerne for problemløsningsmetode *dekomposisjon*:

> *" I've used a temporary variable for the new email field and I think it shouldn't be hard to change everything to work with the new email settings."*
>
> *"I dislike the way it's been written (still) so I've touched up my code a tiny bit using local variables instead of globals for obvious parts."*
>
> *"My strategy is to look for each file corsponding to each gui and update it to include the email field."*

Dette er noen eksempler av forklaringer fra deltakerne for problemløsningsmetode for *bygg og fiks*:

> *"I commented out the isbn variable and then fixed the compilation errors caused by this by removing references to the variable. This lead to other compilation errors and so on."*
>
> *"I gradually removing peices of code and then compiling. the error messages help to identify further code to remove"*
>
> *"Decided that best way to remove isbn is to find the class in which it sits as an instance variable. Removing it from the class will bring up compilation*

errors in all the places where it used. This way I have removed the isbn variable from the Title class"

"I started by commenting out/removing ISBN functionality from the titles and then compiling. Wherever there was an error there was a problem with ISBNs."

Dette er noen eksempler av forklaringer fra deltakerne for problemløsningsmetode angående *generell kunnskap* ved møte av følgende vanskeligheter:

Programlogikk:

"Working on task2 now, searching the entire source for "isbn" and commenting stuff out. This has follow-on effects, for instance, the Title constructor had a variable for isbm, I removed this, so now I have to look for places in the code where new Title(…).Seems like there were no new Title(...) things to take care of."

"My tactic is to remove any ISBN information from title, and let the compiler sort out where else it need…After removing isbn from title.java and removing the compile errors I noticed the error in my logic. Next I searched the files for isbn (without grep) and commented out each line individually…"

"Since the last update I've run the program a few times to investigate what it actually does and blah. came up against a few problems with the find-replace not finding and replacing ISBN shizen."

GUI:

"I expect I have made some mistakes in setting the field size too small and in not making the window large enough. I will check this when I compile and run and correct it where necessary."

"I'm finding working with a GUI in Java difficult because I have no experience with Java GUIs. Have successfully deleted all ISBN related entries in the source. Code compiles, and GUI reflects these changes, am now proceeding to test functionality with the database."

"Having troubles trying to figure out how to manipulate the Java GUI. ISBN is quite an integral part of the system, thus I must keep in mind how to remove the ISBN without damaging other code. I've decided to go through each class and search for 'ISBN'"

Objekt orientering programmering:

> *"Currently trying to figure out how to do the calendar implimentation. It is a difficult task… Not much information has been given to assist. The calendar class is not that nice as i expected. Im trying to create lentdate and due date but it isnt working as i want. so i will ignore it and try another method."*
>
> *"Trying to learn how to use the Calendar class. Since the loan is aglobal object, changes should be made to the loan class to reflect the due back date. This may entail writing a new method called getDueDate which returns the date object of when this book …"*
>
> *"I tried to change the file and it created a heap of errors so i exracted a new version of it… It seems to be working just debugging the program..."*

Algoritmer:

> *"I am trying to integrate a due date into the Loan class. Mainly, it involves figuring how to use the ObjId class. I think I have the back-end complete for due dates but don't know for sure because I have been unable to test it while doing the coding. I misunderstood what an ObjId was. Originally, I had an ObjId in Loan representing a dueDate but now I have an instan…I am having trouble getting the due date to be persistent."*
>
> *"Have founf the piece of code that deals with displaying a loan item. From looking at the loan class it looks like dates are not handled currently, so will add an instance variable to the loan class."*
>
> *"I have found where the output needs to be which is good but it doesnt seem to be working. Written boolean stuff that is confusing to me... But i have written the code that outputs whether a book is expired or not."*

Dette er noen eksempler av forklaringer fra deltakerne for problemløsningsmetode angående *spesifikk kunnskap* av følgende vanskeligheter:

GUI:

> *"I looked through the GUI class names and the names speak for themselves, so it was easy to find those where there are isbn related GUI components.*

Periodically run the application to check that the things that I expect are happening."

"For this problem I tried to think of clever ways to find all the places where the ISBN is used but decided it would be simpler to just systematically eliminate it from all windows I can find."

"However after some testing of the interface it appears that it can no longer find a title regardless of the information you specify, I am trying to track down the source of this error. I have become bogged down on task 2. What I plan to do is delete all references to ISBN and any associated references and fix the errors from there."

Objekt orientering, forståelse og programmering:

" Having a hell of a lot of trouble setting up the EmailField object, as AddressField etc seem to have sprung up out of nowhere… I suspect this is due to some error in saving or retrieving information from the .dat files. Have littered the code with print statements to try and find the error, but without success."

"For some reason I am having difficulty updating subsequent frames in task2 to accomadate the email address, but yet its correctly implemented for new borrower and it is stored in the text file. Perhaps I am missing something fundamental somewhere."

"The persistent storage might be broken if fields disappear from a class… I've just found that my changes broke the persistent store... I'm changing the Title class so that now an empty string is stored in the isdn field, rather than a null reference."

Dette er noen eksempler av forklaringer fra deltakerne for utvikling av *problemløsningsstrategi* før de setter seg inn og løser oppgavene:

"When I read the task the first thing I thought was that I could either read the UML or the code and follow throught to find where code would need to be added to add the email text… My strategy is to look for each file corsponding to each gui and update it to include the email field… Just extending the email addition so that it carries throught the whole program."

"I read the task description and it appears that the task is to remove all references to ISBN in the library and update the relevant fields. The UML

diagram helps to find where exactly one should do it. The code is quite extensive so… I have removed all references to the ISBN no. in the bo and db packages, now working on the user interface… My system now stores the data without the ISBN information. However after some testing of the interface it appears that it can no longer find a title regardless of the information you specify, I am trying to track down the source of this error."

"Currently examining original code structure. Difficult at first to visualise how the classes are dependant on each other. Removing ISBN field information from code - commenting out, debugging code"

4.2.4 Riktighet

I denne oppgaven er analysen av problemløsningsmetoder for å løse vanskeligheter som oppstår i en vedlikeholdningsprosess og strategier for å løse vedlikeholdningsoppgavene gitt i eksperimentet, sammenlignet med hensyn til riktighet og tidsbruk av besvarelsene på oppgavene.

Riktighet er en bedømmelse av kvaliteten av løsningen, som er markert på basis av fullstendighet. En løsning var markert riktig (1) for full funksjonell eller ikke prøvd (0) for de med større defekter. I tilegg er løsningene rangert fra 5 poeng for perfekt løsning til 0 poeng for ikke prøvd.

Bedømmelsen var gitt av en uavhengig konsulent fra et annet forskningsinstitutt. Konsulententen fikk utgitt oppgave spesifikasjon, løsningsforslag, og retningslinje for angivelse av poeng. Hun var ikke involvert i eksperimentet eller lærer i emnet. Gjennom rettingen ble all løsningen kompilert, kjørt, og gjennomtestet for funksjonalitet. Kilde kode ble også undersøkt manuelt nøye for leting av årsak til problemer og syntaks feil.

4.2.5 Tid

Dette er tid brukt, i minutter, for å fullføre (forstå, kode, teste) forandringsoppgavene. Det var kalkulert av end_time minus start_time, hvor start_time er tid når en deltaker lastet ned oppgaveteksten og end_time er tiden når deltaker la opp sin løsning, som ble registrert av SESE verktøy.

Lunsj pause eller andre pauser lenger enn 10 minutter var trukket fra.

5 Resultat

Dette kapittlet presenterer resultatene av studiet. Målet for denne forskningen er å identifisere problemløsningsmetoder og problemløsningsstrategier for studenter i en vedlikeholdning av objekt oritentert applikasjon.

Tabell av deltakerne som var på feedback-collection gruppen med deres problemløsningsmetode og problemløsningstrategi er gitt i Appendiks A.

5.1 Problemløsningsmetode

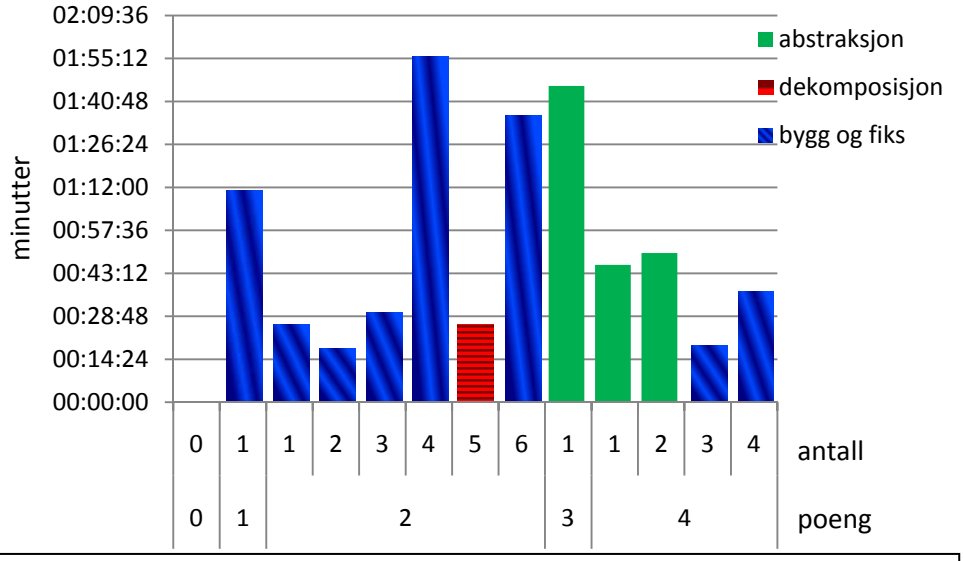
I denne seksjonen presenterer jeg resultatene av analysen av hvilken problemløsningsmetode deltakerne brukte for å kunne løse vanskeligheter av generell- og spesifikk kunnskap som Karahasanovic og Thomas (2007) har kartlagt.

5.1.1 Problemer angående Generell kunnskap

Nedenfor presenterer jeg problemløsningsmetodene deltakerne brukte for å løse vanskelighetene i program logikk, GUI, objekt orientert programmering, algoritmer og programmeringsmiljø.

5.1.1.1 Program logikk

I Karahasanovic og Thomas (2007) studiet påpekte de at studentenes problemer med logikk (sub-kategori 1.2) var relatert til en "if-else" statement fra oppgave 1. Deltakerne fjernet enten for mye eller for lite fra dette statementen og dette gjorde at bibliotek applikasjon ikke fungerte ordentlig. Figur 2 viser antall deltakerne som hadde vanskeligheter med program logikken og metodene de brukte for å konfrontere denne vanskeligheten.



Figur 2: Problemløsningsmetoder for sub-kategori 1.1

I denne analysen var det et flertall subjekter som konfronterte denne vanskeligheten med bruk av metoden bygg og fiks (6 tilfeller) mot abstraksjon (3 tilfeller) og dekomposisjon (1 tilfelle).

Deltakerne som brukte metoden bygg og fiks brukte enten mindre tid på å løse oppgaven eller lengre tid enn gjennomsnittet. Dette kan være på grunn av at de ser denne oppgaven som ren tekst editering som Karahasanovic og Thomas nevnte. Data fra feedback-collectionen viser at kanskje deltakerne undervurderte kompleksiviteten i oppgaven og valgte den enkleste løsningsmetoden de kunne istedenfor å sjekke ut logikken i programmet:

> *"I'm working on Task 2. For this problem I tried to think of clever ways to find all the places where the ISBN is used but decided it would be simpler to just systematically eliminate it from all windows I can find." ID_14*
>
> *" So far the task has been easy and I havn't encountered many problems. I started by commenting out/removing ISBN functionality from the titles and then compiling." ID_25*

To deltakerere rapporterte at de prøvde å løse oppgaven ved bruk av abstraksjon, men hadde vanskeligheter med å forstå logikken av programmet, og valgte dermed å løse oppgaven med fiks og bygg:

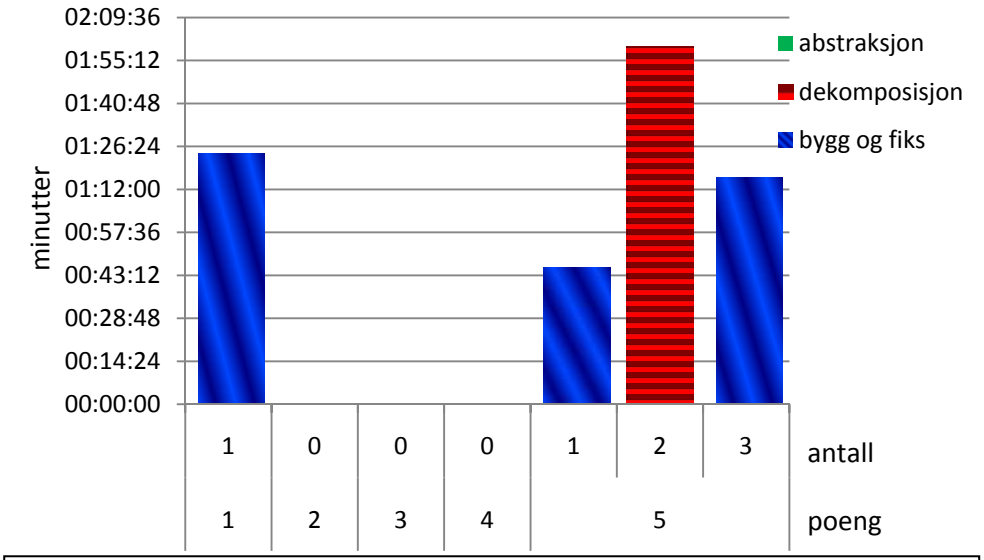
> *" The UML diagram helps to find where exactly one should do it… I have become bogged down on task 2. What I plan to do is delete all references to ISBN and any associated references and fix the errors from there." ID_20*
>
> *"I starting to understand how graphical objects are defined, and don't really want to waste time trying to figure out how to graphically manipulate the GUI. Will instead brute force delete all isbn entries in the source and hope nothing breaks." ID_37*

Mulig på grunn av dette brukte disse to deltakerne lengre tid enn gjennomsnitt for å løse oppgaven. Bedømmelsen av besvarelsene for disse fire deltakerne viser at alle hadde feil i deres besvarelser. De mest vanlige feilene for alle deltakerne var feil i fjerning av en if/else statement og en tekst deklarasjon.

Merk at analysen var av de som hadde problemer med å løse oppgaven. Ikke de som hadde klart å besvare oppgaven fullstendig riktig. Dermed er ikke deltakerne med 5 poeng med.

5.1.1.2 GUI

Karahasanovic et al. (2007) nevnte i sitt tidligere studium, eksperimentet av studenter i UiO, at vanskeligheter i generelle kunnskaper av GUI (sub-kategori 1.2), var det noen deltakere som hadde vanskeligheter med å huske å forstørre GUI vinduet som trengs for å vise ekstra felter i oppgaven 2. Figur 3 viser antall deltaker som hadde vanskeligheter med generelle kunnskaper i GUI og metoden de brukte for å konfrontere denne vanskeligheten.



Figur 3. Problemløsningsmetoder for sub-kategori 1.2

I denne analysen resulterte det til at nesten alle deltakerne konfronterte denne vanskeligheten ved bruk av metoden bygg og fiks (3 tilfeller) og dekomposisjon (1 tilfelle).

Karahasanovic et al. (2007) rapporterte at deltakerne fant denne feilen (glemte å forstørre GUI vinduet) etter å ha testet deres løsninger, og at de klarte å takle denne vanskeligheten. Data fra feedback-collection bekrefter det Karahasanovic et al. konkluderte. Løsningsmetoden de fleste deltakerne brukte for å løse denne vanskeligheten (sub-kategori 1.2.1) er altså bygg of fiks metoden. To av tre som brukte denne metoden fikk bedømmelse 5 av deres besvarelse. Dette kan kanskje forklares med at bygg og fiks metoden gir ikke mulighet for å løse komplekse problemer som Ahmadzadeh et al. (2005) rapporterte, men mestrer derimot enkle problemer.

Det var en deltaker som rapporterte manglende kunnskap om GUI som også hadde denne vanskeligheten. Han hadde brukt en annen problemløsningsmetode for løsing av dette problemet:

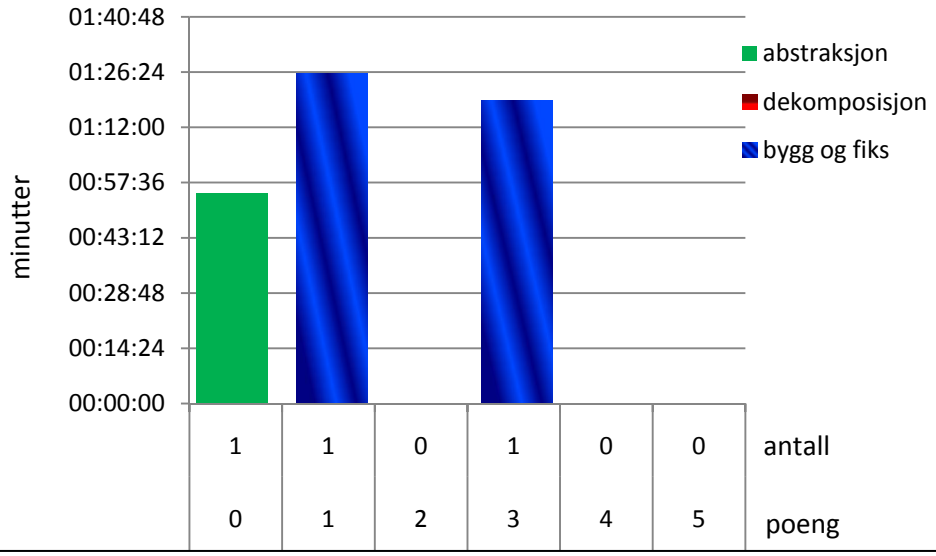
> *"Complete lack of knowledge of how UIs function in Java… This is made easier given what I have learned from doing Task 2… Still adding Email Fields to each borrower window. I expect I have made some mistakes in setting the field size too small and in not making the window large enough. " ID_13*

Deltakeren var fullstendig klar over vanskeligheten og vil løse dette når det trengs. Deltakeren hadde brukt metoden dekomposisjon for å løse denne oppgaven. Han har dekomponert oppgaven i mindre sub-problemer og løste etter hvert hver av sub-problemene, så integrerte han dem sammen og fikk fram en fullstendig besvarelse. Deltakeren klarte å gi en fullstendig besvarelse av oppgaven, selv om han brukte ganske lang tid for å løse oppgaven. Mulig var dette på grunn av manglende kunnskaper i GUI programmering. Denne deltakeren hadde vanskeligheter med både sub-kategori 1.3.3 og 1.3.4.

Merk at analysen var av feedback-collection gruppe. Ikke alle deltakerne hadde besvart på alle oppgavene, så informasjoner om vanskeligheten er begrenset.

5.1.1.3 Objekt Orientert programmering

Figur 4 viser antall deltakere som hadde vanskeligheter med generelle kunnskaper i objekt orientert programmering (sub-kategori 1.3) og metoden de brukte for å konfrontere denne vanskeligheten.



Figur 4. Problemløsningsmetoder for sub-kategori 1.3

I denne analysen resulterte det til at to deltakere konfronterte denne vanskeligheten med bruk av metoden bygg og fiks og en ved metoden abstraksjon. Hvorav to av deltakerne hadde vanskeligheter med forståelse og bruk av Java API klasse (sub-kategori 1.3.3) og en deltaker hadde vanskeligheter med gjenbruk av Java metoder (sub-kategori 1.3.4).

Det var to deltakere som rapporterte vanskeligheter med å forstå bruken av Calender klassen som er integrert i Java API, hvorav en prøvde å få en bedre forståelse av klassen med bruk av Web-dokumentasjon:

”I will probably have to look at the javadoc for Calandar just to familiarise myself with it. I can't figure out where to begin. i am still looking over the code and the manual to work out where i need to store the information about the date when a book is loaned out.” ID_37

Analysen av dette påviste at deltakerne prøvde å få en bedre forståelse og gjøre seg kjent av Calender klassen, og deretter prøvde å bringe inn krav som involverte identifisering av kritiske aspekter av klassen og senere implementerte dette i systemet. Deltakeren har ikke angitt noe besvarelse av oppgaven. Mulig dette er på grunn av deltakeren mislykket å implementere en vellykket løsning, og valgte å fjerne forsøket fra besvarelsen.

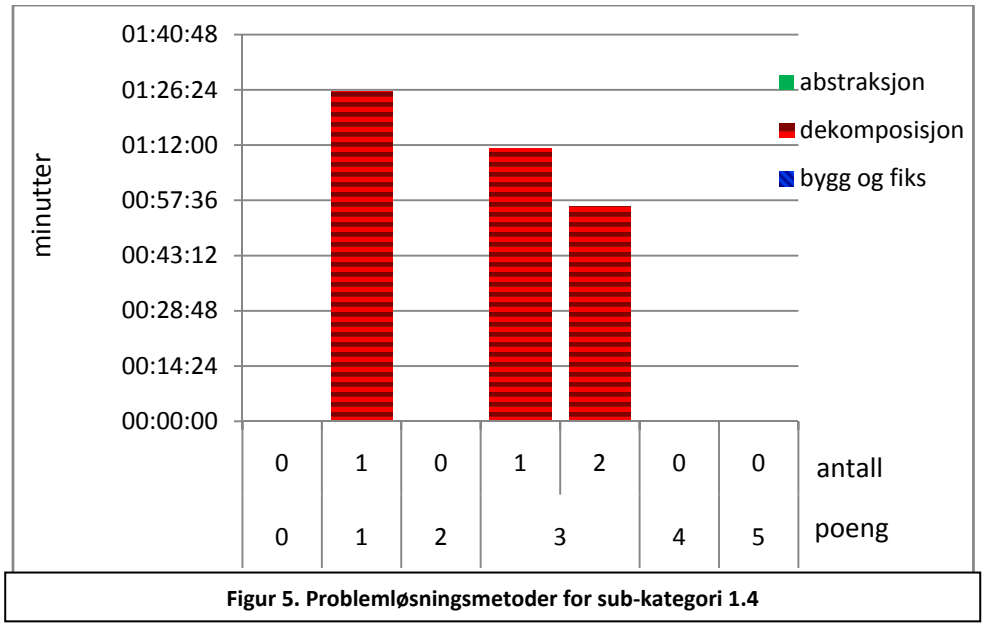
Det var en deltaker som hadde vanskelighet med å gjenbruke metode ved utførelsen av oppgave 2. I Karahasanovic et al. (2007) studium, eksperimentet av studenter ved UiO, nevnte de at årsaken for vanskeligheten med gjenbruk er mulig grunnet at deltakerne i UiO var ikke tilstrekkelig forsiktig, men handlingen kan også indikere mangel på kunnskap i gjenbruk i objekt orientert programmering. Data fra feedback-collection viser at kanskje fordi deltakeren også ikke har forstått koden nok og gjorde dermed ingen forandring av metoden:

”I tried to change the file and it created a heap of errors so i exracted a new version of it… The label names really suck. They should have had more description in them.The code lacks comments... Currently just updating the program to implement email item. It seems to be working just debugging the program.” ID_19

Deltakeren brukte bygg fiks metoden for å løse denne vanskeligheten. Ved å simpelt kopiere og legge til metoden, og deretter teste om det fungerte. Det virker som deltakeren mister problem oversikten. Han fokuserte bare på å få vise elementet på GUI, og glemte å tenke på funksjonaliteten Kommentar fra bedømmelsen av besvarelsen påpekte at kanskje deltakeren kopierte koden fra ”state” og glemme å forandre metoden til ”email”, og dermed fått feil i besvarelsen. Deltakeren brukte lengst tid for å løse vanskeligheten innenfor denne kategorien.

5.1.1.4 Algoritmer

Figur 5 viser antall subjekter som hadde vanskeligheter med algoritmer (sub-kategori 1.4) og metoden de brukte for å løse denne vanskeligheten. Karahasanovic og Thomas (2007) rapporterte at denne vanskeligheten var av kunnskap av algoritmer eller applikasjonens domene (kategori 1.4 i tabell 3).



Figur 5. Problemløsningsmetoder for sub-kategori 1.4

I denne analysen var det tre subjekter som konfronterte denne vanskeligheten med bruk av metoden dekomposisjon.

Tre deltakere brukte dekomposisjon for å løse vanskeligheten med å kalkulere forfall og datoen til lån. Karahasanovic og Thomas (2007) rapporterte at deltakerne synes at denne oppgaven var vanskelig. Vanligvis ville man bruke bibliotek klasser for forskjellige konverteringer og kalkuleringer, kanskje derfor kan det være vanskelig å lage egne utregninger selv for deltakerne. Data fra feedback-collection viser overensstemmelsen med Karahasanovic og Thomas sin påstand. Alle deltakerne prøvde å implementere forskjellige metoder for å få frem forfall datoen av lån:

> *"Thinking about implementing afterLoanPeriod() and printDueDate() into Loan.java." ID_1*
>
> *"I haven't considered the UI changes yet. Just the back-end changes." ID_18*
>
> *"It is a difficult task.. The calendar class is not that nice as i expected. Im trying to create lentdate and due date but it isnt working as i want. so i will ignore it and try another method." ID_19*

Analysen av deltaker med ID_19 viste at personen prøvde å utdype sin forståelse av "Calender" klassen, men likevel tok ikke i bruk av klassen for å løse oppgaven. Dette kan være på grunn av at deltakeren prøvde å forstå Calender klassen for å hjelpe med kalkulering av datoen men synes at dette var vanskelig. Denne deltakeren brukte mer tid enn andre deltakere for å løse oppgaven. Det kan være på grunn av at denne personen brukte mye tid på å forstå Calender klassen. Bedømmelsen

av besvarelsen for begge deltakerne var feil, men deltakeren med ID_1 fikk ros for tilbud av gode metoder for lån.

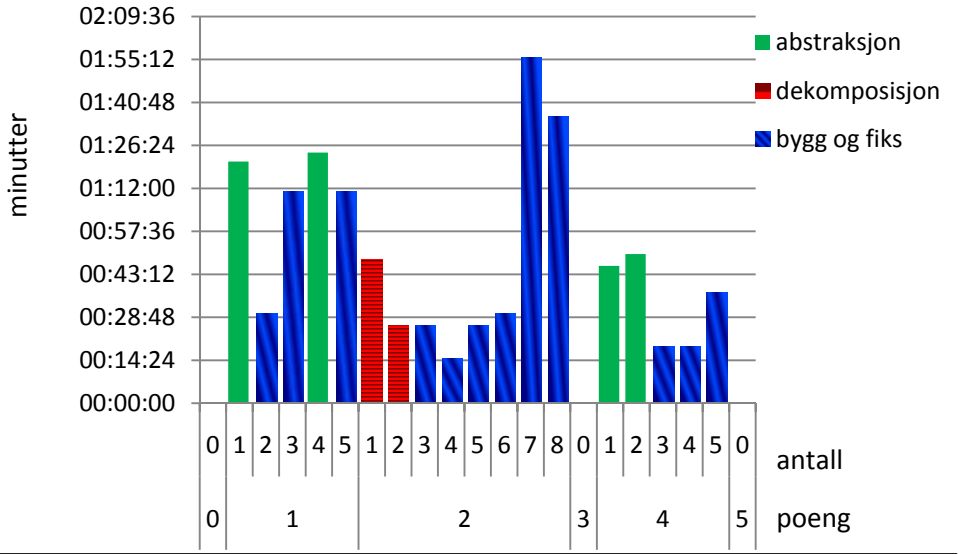
Merk at Karahasanovic og Thomas rapporterte hele 10 deltaker som hadde vanskeligheter å kalkulere datoen, eller ikke prøvd å kalkulere det. Men data fra feedback-collection er begrenset, for mange deltaker har ikke besvart hvordan de har konfrontert dette problemet.

5.1.2 Problemer angående Spesifikk kunnskap

Nedenfor presenterer jeg resultatet av problemløsningsmetodene deltakerne brukte for å løse vanskeligheter med GUI og forståelse av objekt orientering programmering.

5.1.2.1 GUI

I Karahasanovic og Thomas (2007) studium påpekte de at studentenes vanskeligheter med spesifikk kunnskap i GUI (sub-kategori 2.1)var av å legge til eller fjerne fra GUI komponenter og fjerning av tekst deklarasjoner. De rapporterte at deltakerne enten glemte å legge til eller fjerne forskjellige komponenter fra GUI, som radio knapper eller tekstfelter eller mislykket å fjerne alle label deklarasjoner av ISBN i oppgave 1. Figur 6 viser antall deltakere som hadde vanskeligheter med GUI og metoden de brukte for å konfrontere denne vanskeligheten.



Figur 6. Problemløsningsmetoder og sub-kategori 2.1

I denne analysen resulterte det til at mange hadde vanskelighet med GUI. Flertallet av subjekter konfronterte denne vanskeligheten med bruk av metoden bygg og fiks (13 tilfeller) mot abstraksjon (3 tilfeller) og dekomposisjon (2 tilfeller). Hvorav seks tilfeller med bruk av bygg og fiks, et tilfelle abstraksjon, og et tilfelle av dekomposisjon er innenfor å legge til eller fjerning av GUI komponenter. Åtte tilfeller bygg og fiks, tre tilfeller abstraksjon og et tilfelle dekomposisjon metode er innenfor av fjerning av tekst deklarasjoner i oppgave 1.

Data fra feedback-collection viser at, det var de som brukte metoden bygg og fiks som både brukte minst og mest tid på å løse sine oppgaver der det gjelder disse GUI vanskelighetene. Deltakeren som har brukt minst tid for å løse disse vanskelighetene rapporterte følgende i feedback-collection:

> *"I'm operating basically on autopilot and just checking as I go." ID_14*

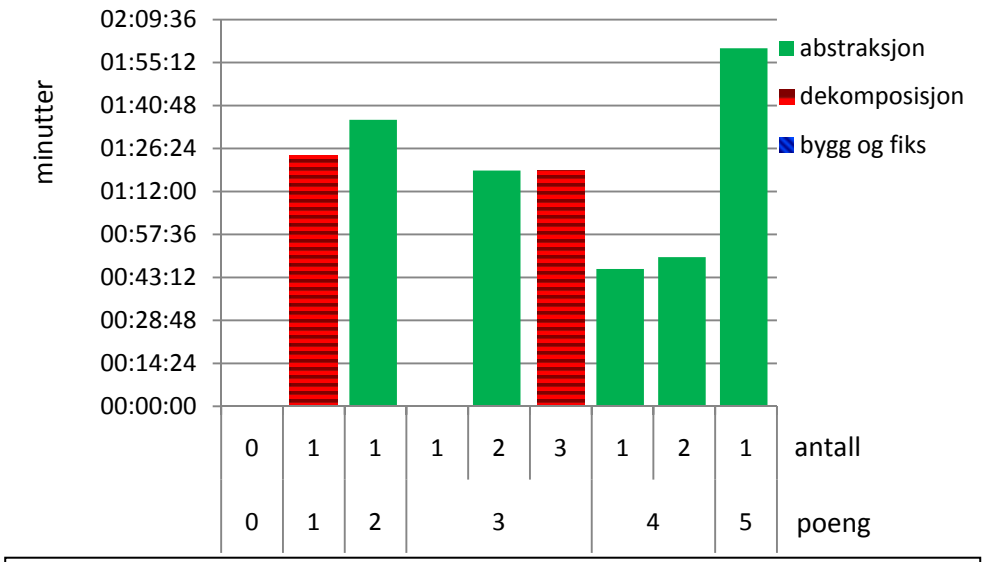
Dette kan kanskje forklares med at deltakeren brukte veldig lite tid på besvarelsen av oppgavene sine. Ved bruk av bygg og fiks metoden førte det til at deltakeren begrenset seg selv av programmets oversikt og brukte ikke tid til å utforske problemets situasjon. De som hadde brukt mest tid på å løse disse vanskelighetene rapporterte følgende i feedback-collection:

> *"My productivity is a little low. I attribute this to the fact I feel a little tired." ID_20*
>
> *"I'm finding working with a GUI in Java difficult because I have no experience with Java GUIs." ID_37*

Dette kan kanskje skyldes ved ekstern årsaker og manglende kunnskaper i GUI programmering fører til ekstra tid for å løse oppgavene. Noe av det kan også skyldes at begge ID_20 og ID_37 brukte en del tid på å lese gjennom dokumentasjon etter at de leste oppgaveteksten.

5.1.2.2 Objekt Orientert forståelse og programmering

Karahasanovic og Thomas (2007) rapporterte studentenes problemer med objekt orientert forståelse og programmering (sub-kategori 2.2) var relatert til programstruktur, innvirkninger på klassene, innvirkninger i klassene og arv av funksjonalitetens. For forandringsoppgaver trenger deltakerne å forstå struktur av applikasjonen, forståelse av relasjoner mellom klassene å finne de klassene som ble påvirket av forandringene. De rapporterte at deltakerne trengte også å forstå arv av funksjonalitetens for å løse oppgavene. Klasser i applikasjonsbiblioteket som trengtes å holde persistent måtte arve en abstrakt klasse, og feilet med å holde dataene "persistent", antyde at dette var vanskelig for deltakerne. Figur 7 viser antall subjekter som hadde vanskeligheter med forståelse og programmering innenfor objekt orientering og metodene de har brukt for å konfrontere disse problemene.



Figur 7. Problemløsningsmetoder for sub-kategori 2.2

I denne analysen resulterte det til deltakere som konfronterte disse vanskelighetene ved bruk av metoden abstraksjon (5 tilfelle) og dekomposisjon (1 tilfelle). Hvorav fire av tilfellene var relatert til programmets struktur, et tilfelle var relatert til innvirkninger på klassene og 2 tilfeller var relatert til arv av funksjonalitet.

Data av feedback-collection viser at alle de som hadde problemer med forståelse av programmets struktur har prøvd å løse denne vanskeligheten ved å sette seg inn i dokumentasjonen. To av subjektene rapporterte:

> *"Currently examining original code structure. Difficult at first to visualise how the classes are dependant on each other." ID_9*
>
> *"Am reading through the documentation for the system (Library.pdf). Confused about the purpose of the Persistent class." ID_13*

Det er mulig at ved bruk av dokumentasjonen har begge deltakerne klart å bygge en abstrakt modell av programmet og dermed løst problemet med forståelse av programmets struktur. I tillegg rapporterte også deltaker ID_13 at han hadde vanskeligheter med forståelse av Persistent klassen som var av vanskelighet kategorisert under kategorien av arv av funksjonalitet. Ved bruk av metoden abstraksjon klarte deltaker ID_13 å komme seg gjennom både vanskeligheten med programmets struktur og vanskeligheten med arv av funksjonalitet. Å ha en fullstendig forståelse av programmet førte kanskje til at deltakeren fikk "fullstendig riktig løsning" i bedømmelsen av oppgave 1.

5.1.3 Effekter av problemløsningsmetoder i en vedlikeholdningprosess

I denne seksjonen presenterer jeg resultatet funnet ved å sammenligne deltakernes bruk av problemløsningsmetoder mot vanskelighetene de møter i vedlikeholdningsprosessen, med hensyn til riktighet av besvarelsen og tid brukt for besvarelsen.

5.1.3.1 Problemløsningsmetode vs. riktighet

Jeg har sammenlignet resultatet av deltakerne med hensyn til forskjellige problemløsningsmetoder og riktighet av besvarelsen. Resultatet vises i Tabell 5.

Vanskeligheter Sub-kategori	poeng fra oppgaven				Deltakernes metode og riktighet					
	antall	min	max	Median	% abstraktsjon	% riktighet	% dekomposisjon	% riktighet	% bygg og fiks	% riktighet
1.1	12	1	4	2,5	25 %	73 %	8 %	40 %	67 %	48 %
1.2	4	1	5	3	0 %	---	25 %	100 %	75 %	73 %
1.3	3	0	3	1,5	33 %	Ikke prøvd	0 %	---	67 %	40 %
1.4	3	1	3	2	0 %	---	100 %	47 %	---	---
2.1	18	1	4	2,5	22 %	50 %	11 %	40 %	67 %	45 %
2.2	7	1	5	3	71 %	72 %	29 %	40 %	0 %	---
gjennom snitt		0,83	4	2,41	25 %	65 %	29 %	53 %	46 %	51 %

Tabell 5. Problemløsningsmetoder vs. riktighet

Tabellen viser vanskelighetene med program logikk, og abstraksjon er den mest suksessfulle metoden for å løse denne vanskeligheten med 73 % riktighet. Metoden abstraksjon fikk også 72 % riktighet med vanskeligheten med OO forståelse og programmering i spesifikk kunnskap. Årsaken til den høye prosenten med riktighet av besvarelsen kan kanskje forklares at metoden abstraksjon gir deltakerne et forenklet syn av program logikken og strukturen, og bedre fokus på det som er viktig og evne til å lettere identifiserer kjernen av programmet.

Dekomposisjon metoden utmerket seg i løsning av vanskeligheter med GUI i generell kunnskap kategorien. Her klarte deltakerne som har tatt i bruk av dekomposisjon løse vanskeligheten med 100 % riktighet. Problemer der kravet er å modifisere flere deler av systemet, og deretter kombinere delene for å en felles løsning, er dekomposisjon et godt valg. Siden teknikken av metoden dekomposisjon er å dele et problem i mindre deler, for deretter å arbeide med delene av problemet hver for seg og deretter integrere dem sammen og få fram en helhetsløsning. Framgangsmåten av dekomposisjon ser ut til å stemme overens med denne oppgaven, og dermed passer dekomposisjon meget bra til å løse dette problemet.

Arvanitis et al. (2001) påstod at ”bygg og fiks metode ofres ikke mulighet til å løse komplekse problemer”. Hvorav vanskeligheter med GUI i generelle kunnskap klarte bygg og fiks å løse denne vanskeligheten med 73 % riktighet, etter dekomposisjon med 100 % riktighet. Den mulige årsaken for

dette var at ”glemme å forstørre GUI” er ikke et så veldig kompleks problem, og dermed kan bygg og fiks også være en god metode til å løse denne vanskeligheten.

Deltakerne som utførte metode abstraksjon for å konfrontere sine vanskeligheter har levert bedre løsninger enn dekomposisjon og bygg og fiks med 65 % riktighet mot 53 % og 51 % riktighet gjennomsnitt av alle vanskelighetene. Dette viser at metoden abstraksjon gir større riktighet for å løse vanskelighetene som Karahasanovic og Thomas (2007) kategoriserte i deres studium.

5.1.3.2 Problemløsningsmetode vs. tid

Jeg har sammenlignet resultatene av deltakerne av de forskjellige problemløsningsmetodene med hensyn til tidsbruk av besvarelsen. Resultat vises i Tabell 6.

Vanskelighet Sub-kategori	antall	Tid fra oppgaven				subjektenes metode og tid				
		min	max	median	% abstraksjon	gjennomsnitt tid	% dekompos isjon	gjennomsnitt tid	% bygg og fiks	gjennomsnitt tid
1.1	12	00:18:00	01:56:00	01:07:00	25 %	01:07:20	8 %	00:26:00	67 %	00:51:38
1.2	4	00:46:00	02:00:00	01:23:00	0 %	---	25 %	02:00:00	75 %	01:08:40
1.3	3	00:55:00	01:26:00	01:10:30	33 %	00:55:00	0 %	---	67 %	01:22:30
1.4	3	00:56:00	01:26:00	01:11:00	0 %	---	100 %	01:11:00	---	---
2.1	18	00:15:00	01:56:00	01:05:30	22 %	01:05:15	11 %	00:37:00	67 %	00:46:20
2.2	7	00:46:00	02:00:00	01:23:00	71 %	01:18:12	29 %	01:21:30	0 %	---
gjennomsnitt		00:39:20	01:47:20	01:13:20	25 %	01:06:27	29 %	01:07:06	46 %	01:02:17

Tabell 6. Problemløsningsmetoder vs. tid

Tabell 6 viser vanskeligheten med programlogikk (sub-kategori 1.1) at bare 8 % av 12 deltakere brukte bare 26 minutter for å løse oppgaven der problemet oppstod. Dette viser til at enten deltakeren har veldig god forståelse av programmet og vet akkurat hva og hvor i programmet det trenger å modifiseres for å fikse problemet, eller det motsatte ved å ikke klare å løse alle aspektene av problemet. Dessverre er resultat fra besvarelsen henviser til sistnevnte. Deltakerene klarte ikke å fullstendig løse oppgaven, glemte å fjerne ”else keyword” fra FindTitleDialog og ikke fjernet alle label deklarasjoner. Dermed viser overensstemmelse med Sebastian et al. om ulempen ved bruk av dekomposisjon; helhetsynet av problemet bli borte, og føre til degradering av plan kvalitet. På en annen side er tidsbruken av dekomposisjon ved løsing av vanskelighet GUI i generelle kunnskap kategori den lengste tidsbruk av alle på hele 2 timer. Dette kan kanskje forklares med at negative interaksjoner som oppstod fra utførelsen av dekomposisjonen. ”*Ved oppnåelse av en informasjon som føre til sletting av andre nødvendige informasjoner*” var Sebastian et al. (2006) forklaring av negativt interaksjon. De nevnte også å finne og fikse disse negative interaksjoner kan være mer tidskostbar enn problemet selv.

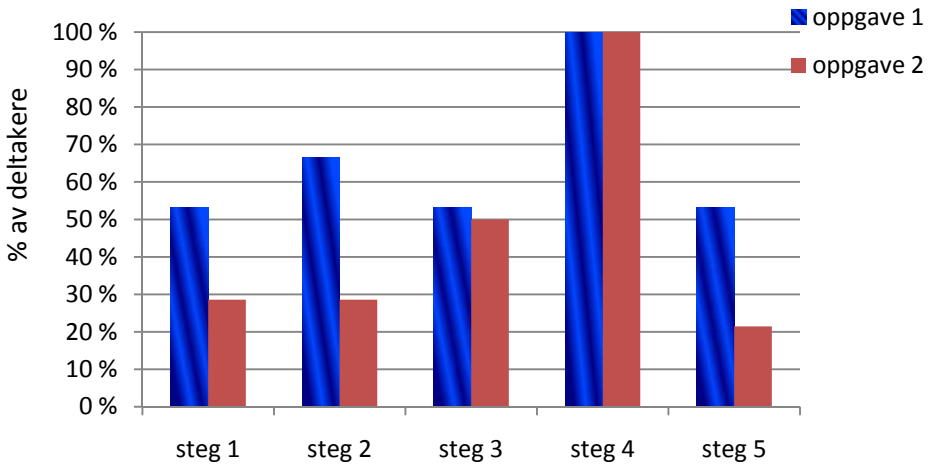
Metoden abstraksjon tidsbrukt for å konfrontere forskjellige vanskeligheter ligger rundt en time. Grunnen til dette kan være at ved bruk av metode abstraksjon valgte mange deltakere å se gjennom dokumentasjonen for å få en abstrakt modell av programmet. Som Kramer et al. (2006) påstod at

modeller er en forenkling av virkeligheten til å forfremme forståelser og begrunnelser, og dette hjelper til å forstå og analysere store og komplekse problemer. Men bruken av dokumentasjon er tidskostbar.

Deltakerne som utførte metode bygg og fiks for å konfrontere sine vanskeligheter bruker minst tid for å løse sine oppgaver med 01:02:17. Med abstraksjon er tidsbruken 01:06:27 og dekomposisjon med 01:07:06. Dette viser at forskjellene for bruken av disse problemløsningsmetodene er minimalt, og det tar gjennomsnittlig en time for å løse de enkelte vanskelighetene som Karahasanovic og Thomas (2007) kategoriserte, uansett valg av metode.

5.2 Problemløsningsstrategi

I denne seksjonen presentere jeg resultatet av analysen om deltakerne har tatt i bruk av den seksstegs problemløsningsstrategi som var utviklet av Deek et al. (1999), for å løse oppgavene gitt i eksperimentet. Det er ikke nødvendigvis for deltakerne å ta i bruk av alle disse prinsippene for å danne en strategi. Data fra feedback-collection har blitt brukt å analysere om deltakerne tok i bruk noen av disse definisjonene i prosessen for å løse oppgavene, og hvor godt de løste dem. Det siste steget av strategien ser jeg bort i fra ettersom dette ikke påvirke kvaliteten av løsningene.



Figur 8. Problemløsningssteg vs. riktighetsprosent

Figur 8 viser at ved løsing av oppgave 1 var det 53 % deltakere uførte steg 1 (formulering av problemet), 67 % deltakere utførte steg 2 (planlegging av løsningen), 53 % deltakere utførte steg 3 (konstruering av løsningen), 100 % deltakere utførte steg 4 (oversettelse av løsningen) og 53 % deltaker utførte steg 5(levering av løsningen). Av oppgave 2 var det 29 % deltakere uførte steg 1 (formulering av problemet), 29 % deltakere utførte steg 2 (planlegging av løsningen), 50 % deltakere utførte steg 3 (konstruering av løsningen), 100 % deltakere utførte steg 4 (oversettelse av løsningen) og 21 % deltaker utførte steg 5 (levering av løsningen). Jeg har ikke tatt med analysen av oppgave 3, for en stor del av deltakerne ikke fullførte oppgaven og det var ikke mange nok besvarelsen på feedback-collection for å analyse i forhold til oppgave 1 og oppgave 2. Henvises til Validitet.

Data fra feedback-collection viser store forskjeller mellom oppgave 1 og oppgave 2 på steg1. Nesten dobbelt så mange deltakere utførte ”formulering av problemet” og ”planlegging av løsningen” i oppgave 1 enn oppgave 2. Det er mulig fordi noen deltaker følte de har gjort seg kjent med programmet fra oppgave 1 og følte ikke behov for å gjøre dette igjen i oppgave 2, siden begge oppgavene er er av samme program. Dette er illustrert av følgende kommentar skrevet på feedback-collection:

> *”This [oppgave2] is made easier given what I have learned from doing Task 1” ID_13*

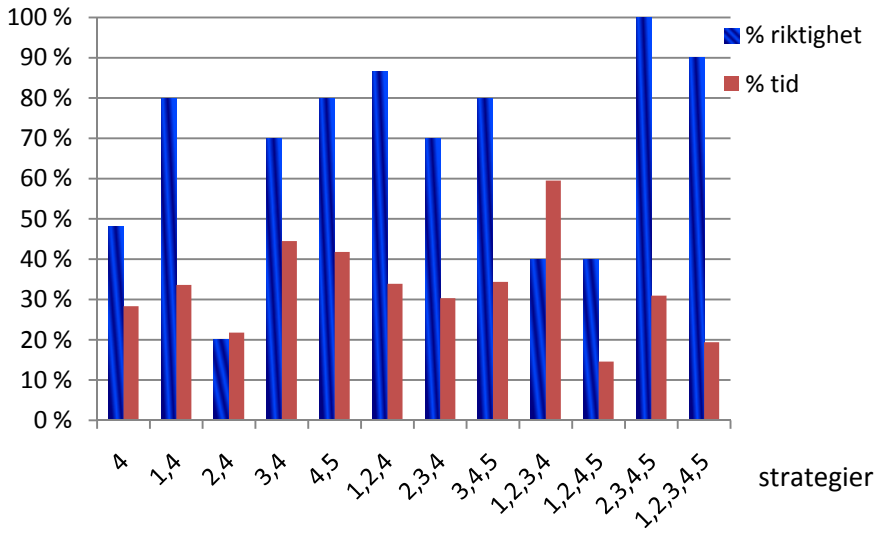
Det viser også store forskjellige mellom oppgave 1 og oppgave 2 på steg 2. Mer en dobbelt så mange har rapportert planleggingen av sin løsning i oppgave 1 enn oppgave 2. Kansje er årsaken på grunn av at oppgave 1 er den letteste av alle oppgavene gitt i eksperimentet, så ved løsing av oppgaven 1 involverte ikke implementasjonen og integrasjoner av andre koder. Dermed er det lettere å få fram en plan til å løse oppgave 1 enn 2. Følgende utdrag fra feedback-collection illustrerte dette:

> *”For this problem I tried to think of clever ways to find all the places where the ISBN is used but decided it would be simpler to just systematically eliminate it from all windows I can find.” ID_14*

Data fra feedback-collection viser at 100 % av deltakere utførte steg 4 på både oppgave 1 og oppgave 2. Denne ”oversettelse av løsningen” steget består av tre aktiviteter; implementasjon, integrasjon og debugging. Disse tre aktiviteten er en grunnleggende del av programmeringen, og er noe som må utføres for å modifisere/implementere noe i et program. Kansje dette kan forklares at utførelsen av steg 4 er noe som må gjøres i en vedlikeholdningsprosess.

Steg 5 viser også store forskjeller på oppgave 1 og oppgave 2, der oppgave 1 har 21 % og oppgave 2 53 % av deltakere som tar i bruk av dette steget. Dette”testing av løsningen” steget består i hovedsak av 3 aktiviteter; kritisk analysering, evaluering og revidering av løsningen. Dette kan kanskje forklares at oppgave 2 er mer rettet mot dette steget enn oppgave 1. Oppgave 2 er forandring av GUI ved å legge til et email felt, og få email-feltet til å fungere med systemet. Dette gir muligheter til å analysere med testing av egne data, evaluere om emailen fungere i samsvar med systemet og eventuelt oppdatere systemet for feil ved hjelp av øyeblikkelig visuell respons i GUIen ved forandringer. Hvorav oppgave 1 består kun av fjerning av ISBN som eneste visuell respons fra GUIen.

Ved å se om deltakerne har tatt i bruk av noen form for strategi, analyserte jeg nærmere på hvordan deltakerne har gått fram til en besvarelse av oppgave 1 og oppgave 2. Figur 9 viser hvordan deltakerne har gått fram til å løse oppgave 1 og oppgave 2. En rekke forskjellige kombinasjoner av problemløsningsstrategistegene var blitt brukt av deltakerne da de løste oppgavene.



Figur 9. Problemløsningsstrategi vs. riktighet og tid

Tabellen viser de strategiene som har fått mest riktighet og minst tidsbruk inkluderer bruk av steg 5 i strategien ((2,3,4,5) og (1,2,3,4,5)). Dette viser til at Deek et al. (1999) originale seksstegstrategi gir best utelling på besvarelse av oppgavene.

5.2.1 Effekter av problemløsningsstrategier i en vedlikeholdningprosess

I denne seksjonen presenterer jeg resultatet funnet ved å sammenligne deltakernes bruk av problemløsningsstrategier mot oppgaver i vedlikeholdningsprosessen, med hensyn til riktighet av besvarelsen og tid brukt for besvarelsen.

5.2.1.1 Problemløsningsstrategi vs. riktighet

Jeg har sammenlignet resultatene av hvilken deltaker har utført hvilket av de forskjellige problemløsningsstrategistegene med hensyn til riktighet av besvarelsen. Resultat er vist i Tabell 7.

steg	Oppgave 1					Oppgave 2				
	antall deltaker	poeng			% riktighet	antall deltaker	Poeng			% riktighet
		min	maks	median			min	maks	median	
1	8	2	5	3,5	65 %	3	5	5	5	100 %
2	10	1	5	3	60 %	4	5	5	5	100 %
3	8	2	5	3,5	65 %	7	1	5	3	71 %
4	---	---	---	---	---	---	---	---	---	---
5	8	2	5	3,5	70 %	3	5	5	5	100 %
gjennomsnitt		1,75	5	3,375	65 %		4	5	4,5	93 %

Steg 4 er ikke tatt med i analysen

Tabell 7. Problemløsningssteg vs. riktighet

Som tidligere nevnt I avsnittet 5.2 var utførelsen av steg 4 (oversettelse av løsningen) noe som må gjøres i en vedlikeholdningsprosess. Dermed har alle deltakerne som har gitt besvarelse i oppgave 1 og oppgave 2 utført dette steget. I denne analysen velger jeg å se bort fra dette steget og fokusere på de andre stegene og analysere dette med riktighet av besvarelsene av oppgavene deltakerne har levert.

Tabellen viser høy riktighet prosent av besvarelsene til deltakerne som har utført minimum et av stegene (1, 2, 3 og 5) i prosessen deres for å løse oppgavene. Deltakere som har utført steg 5 har det høyeste riktighet prosenten i gjennomsnittet på løsningen på oppgave 1 med 70 %, enn 65 % med steg 1 og steg 3 og 60 % med steg 2, og kan dermed si at dette steget er noe viktigere enn de andre stegene for å utvikle en fullstendig besvarelse på oppgave 1. Dette steget består av tre viktige aktiviteter som kritisk analyse, evaluering og revidering av løsningen, hvorav disse aktivitetene er nødvendige for å passe på at løsningen har besvart problemstillingene stilt i oppgaveteksten. Av oppgave 2 viser tabell 18 at steg 1, steg 2 og steg 5 ga høyeste riktighetsprosent i gjennomsnittet med hele 100 % riktighet. Totalt sett av analysen fra tabell 19 kan dette tolkes som ved bruk av disse problemløsningsstrategistegene frambringe positive virkninger til resultatet.

Tabell 8 viser strategiene deltakerne har utført ved løsing av oppgave 1 og oppgave 2 med gjennomsnitt riktighetsprosent. Tabellen viser overensstemmelse med funnet ovenfor at de som har tatt i brukt steg 5 med i deres strategi, strategier (4 og 5), (3, 4 og 5), (2, 3, 4 og5) og (1, 2, 3, 4 og 5), har fått høyere riktighet prosent av deres besvarelse. Dette viser til at ”testing av løsningen” er en viktig prosess for å levere et riktig resultat. Dette steget gir deltakeren en siste oppsjekking at løsningen før levering av oppgaven. Resultatetet av tabellen viser til at Deek et al. (1999) originale seks steg strategi gir mest utelling på besvarelse av oppgavene

	Riktighet i oppgave 1 og oppgave 2											
strategi	4	1,4	2,4	3,4	4,5	1,2,4	2,3,4	3,4,5	1,2,3,4	1,2,4,5	2,3,4,5	1,2,3,4,5
antall	5	1	1	4	1	3	2	3	1	2	1	4
% riktighet	48 %	80 %	20 %	70 %	80 %	87 %	70 %	80 %	40 %	40 %	100 %	90 %

Tabell 8. Problemløsningsstrategi riktighetsprosent i oppgave1 og oppgave2

5.2.1.2 Problemløsningsstrategi vs. tid

Jeg har sammenlignet resultatene av hvilken deltaker har utført hvilket av de forskjellige problemløsningsstrategistegene med hensyn til tidsbruk av besvarelsen. Resultat er vist i tabell 9.

	oppgave 1					oppgave 2				
steg	deltaker	tid			gjennom	deltaker	tid			gjennom
	antall	min	max	median	snitt	antall	min	max	median	snitt
1	8	00:26:00	01:56:00	01:11:00	01:03:15	4	00:46:00	02:00:00	01:23:00	00:58:30
2	10	00:19:00	01:56:00	01:07:30	01:00:42	4	00:46:00	02:00:00	01:23:00	01:17:30
3	8	00:18:00	01:46:00	01:02:00	00:41:00	7	00:33:00	02:00:00	01:16:30	01:10:09
4	---	---	---	---	---	---	---	---	---	---
5	8	00:25:00	01:56:00	01:10:30	01:03:23	3	00:46:00	02:00:00	01:23:00	01:22:00
gjennomsnitt		00:22:00	01:53:30	01:07:45	00:57:05		00:42:45	02:00:00	01:21:23	01:12:02

Steg 4 er ikke tatt med i analysen

Tabell 9. Problemløsningssteg vs. tid

Som tidligere nevnt I avsnittet 5.2 at utførelsen av steg 4 (oversettelse av løsningen) er noe som må gjøres i en vedlikeholdningsprosess, så dermed har alle deltakerne som har gitt besvarelse i oppgave 1 og oppgave 2 har utført dette steget. I denne analysen velger jeg å se bort fra dette steget og fokusere på de andre stegene og analysere dette med tidsbruk av besvarelsene av oppgavene deltakerne har levert.

Tabell 9 viser til ved utførelsene av stegene resulterte til samme tidsbruk for å løse oppgaven. Hvorav er det unntak av steg 3 i oppgave 1. Steg 3 (konstruering av løsningen) yter til betydelig mindre tidsbruk på å løse oppgaven enn de andre stegene. Hovedaktiviteten i dette stadiet er syntese, som passer på reintergrasjon av beslektet komponenter i en sammenhendende helhet, rearrangere når det er nødvendig, etablere forholder og produsere nye og vel organisert helhet som en mulig løsning av problemet. I oppgave 1 består det av å fjerne ISBN fra programmet, hvorav ISBN involvere bare noen få klasser i systemet, og dermed fører dette til at aktiviteten av dette steget ikke trenger å integrere og rearrangere mange komponenter i koden, og løsningen er dermed bare enkelt å finne ut hvor ISBN er i koden og fjerne dette. Dette steget bidrar dermed til å enkelt og rask løsning av oppgave 1. Data i feedback-collection illustrerte dette:

> *"I looked through the GUI class names and the names speak for themselves, so it was easy to find those where there are isbn related GUI components."*
> *ID_8*

Steg 3 yter ikke like bra tidsbruk for oppgave 2. Dette kan kanskje forklares at i oppgave 2 angås det en rekke flere klasser og implementering av løsningen er mer kompleks enn oppgave 1. Oppgave 2

trenger dermed flere reintegrasjoner av komponenter, mer rearrangeringer av kodene, flere etableringer og større produksjon av komponenter for å løse oppgaven, og dermed krever mer tidsbruk. Tabellen viser at de fleste stegene har samme tidsbruk og ingen av stegene gir noe spesiell forkortelse av tidsbruk for å løse et problem.

Tabell 10 viser strategi deltakerne har valgt ved løsing av oppgave1 og oppgave 2 med tidsbruk av oppgaven i prosent. Tabell 10 viser de strategiene som har tatt i bruk av steg 3 ikke gir noe særlig mindre tidsbruk i prosessen løsing av oppgavene, slik som funnet i tabell 21. Men derimot er strategiene som inneholder steg 2, strategier (2 og 4), (1, 2, 4 og 5), (2, 3, 4 og 5) og (1, 2, 3, 4 og 5), brukte minimal av gjennomsnitt tid for løsing av oppgave 1 og oppgave 2. Dette viser at aktiviteter som planlegging, kartlegge alternative løsninger og deler problemer i mindre sub-problemer er viktige prinsipper for deltakere å raskere problemløse oppgave 1 og oppgave 2. Dette viser også til at Deek et al. (1999) original seksstegstrategi er blant de som gir minst tidsbruk i prosess av besvarelse av oppgavene.

	Tidsbruk I oppgave 1 og oppgave 2											
strategi	4	1,4	2,4	3,4	4,5	1,2,4	2,3,4	3,4,5	1,2,3,4	1,2,4,5	2,3,4,5	1,2,3,4,5
antall	5	1	1	4	1	3	2	3	1	2	1	4
% tidsbruk	28 %	34 %	22 %	45 %	42 %	34 %	30 %	34 %	60 %	15 %	31 %	19 %

Tabell 10. Problemløsningsstrategi tidsbruk i oppgave1 og oppgave2

5.3 Oppsumering

Eksperimentet som ble brukt i dette studium var av 3600 LOC bibliotek applikasjon system, skrevet i Java og kan betrakte som et mediumstor applikasjon I forhold til klassifikasjon gitt av Mayhauser og Vans (1995). Deltakere var 34 studenter I deres tredje år I informatikk I UWA. Disse studentene var delt I to grupper, 17 på feedback-collection og 17 på Control Silent. Analysen var av feedback-collection gruppe med hensyn til problemløsningsmetoder de har brukt for å konfrontere vanskeligheter som oppstod, kategorisert av Karahasanovic og Thomas (2007), og om de brukte noen form for problemløsningsstrategi ved utførselen av oppgavene i vedlikeholdningsprosessen. Resultatene var sammenlignet med hensyn til problemløsningsmetoder (abstraksjon, dekomposisjon og bygg og fiks) og problemløsningsstrategi (seksstegs problemløsningsstrategi) med hensyn til riktighet av besvarelsen og tidsbrukt i prosessen.

Vanskelighet Sub-kategori	Deltakernes metode, riktighet og tid								
	% abstraksjon	% riktighet	tid	% dekomposisjon	% riktighet	tid	% bygg og fiks	% riktighet	tid
1.1	25 %	73 %	01:07:20	8 %	40 %	00:26:00	67 %	48 %	00:51:38
1.2	0 %	---	---	25 %	100 %	02:00:00	75 %	73 %	01:08:40
1.3	33 %	ikke prøvd	00:55:00	0 %	---	---	67 %	40 %	01:22:30
1.4	0 %	---	---	100 %	47 %	01:11:00	---	---	---
2.1	22 %	50 %	01:05:15	11 %	40 %	00:37:00	67 %	45 %	00:46:20
2.2	71 %	72 %	01:18:12	29 %	40 %	01:21:30	0 %	---	---
gjennom snitt	25 %	65 %	01:06:27	29 %	53 %	01:07:06	46 %	51 %	01:02:17

Tabell 11. Problemløsningsmetoder vs. riktighet og tid

Tabell 11 er en sammenkobling av tabell 5 og tabell 6. Tabell 11 viser at ved tidsbruk av metodene abstraksjon, dekomposisjon og bygg og fiks er meget lik. Forskjellene mellom metodene er minimalt, men abstraksjonsmetoden gir noe høyere riktighet med 65 % enn dekomposisjon med 53 % og bygg og fiks med 51 % for å løse vanskelighetene som Karahasanovic og Thomas (2007) kategoriserte. Dermed er det nesten ingen forskjeller enten ved bruk av den ene eller den andre metoden for å løse disse vanskelighetene. Ved bruk av dekomposisjon eller bygg og fiks gir omtrent samme resultat, og ved bruk av abstraksjon har noe fortrinn for bedre resultat med hensyn til gjennomsnittlige riktighetsprosent. Dette kan kanskje forklares at ved bruk av abstraksjon får brukeren en helhetlig forståelse av programmet og dermed lettere vet hva og hvor i koden skal modifiseres. Abstraksjon utmerket seg i løsing av vanskeligheter som program logikk (sub-kategori 1.1, 73 % riktighet) og program struktur (sub-kategori 2.2, 72 % riktighet). Hvorav metodene dekomposisjon og bygg og fiks fungerte utmerket ved vanskeligheter med GUI (sub-kategori 1.2, 100 % riktighet og 73 % riktighet).

antall steg brukt	oppg1 antall deltaker	riktighet %	gjennomsnitt tidbrukt	oppg2 antall deltaker	riktighet %	gjennomsnitt tidbrukt
1	27 %	55 %	22,54 %	38 %	80 %	30,23 %
2	33 %	68 %	19,66 %	38 %	100 %	40,07 %
3	27 %	55 %	37,75 %	0 %	---	---
4	13 %	80 %	20,05 %	25 %	100 %	29,70 %
totalt	100 %	65 %	100 %	100 %	93 %	100,00 %

steg 4 er ikke tatt med i analysen

Tabell 12. Problemløsningssteg riktighetsprosent i oppgave1 og oppgave2

Tabell 12 viser at det er 27 % av deltakere som tar i bruk av et av stegene til ved utførelsen av oppgave 1 og 38 % av deltakere ved oppgave 2, 33 % av deltakere tar i bruk av to av stegene ved utførsel av oppgave 1 og 38 % av deltakere ved oppgave 2, 27 % av deltakere tar i bruk tre av stegene ved utførsel av oppgave 1 og ingen ved oppgave 2, og 13 % av deltakere utførte alle stegene ved

oppgave 1 og 25 % av deltakere ved oppgave 2. Dette viser til at alle deltakere tar i bruk minimum et av stegene i utførsel av oppgaveløsning for oppgave 1 og oppgave 2. Tabellen viser også at de som har utført alle stegene har fått mest riktighet prosent av besvarelsen og noe lavere tidsbruk i gjennomsnittet for begge oppgave 1 og oppgave 2, og kan dermed konkludere at disse stegene gir en positiv virkning ved utførselen av oppgaveløsning.

Tabell 7 fra problemløsningsstrategiene viser til at ikke alle stegene er like viktig i en løsningsprosess. Steg 5 viser seg å være noe viktigere enn steg 1 og steg 3 og deretter følger steg 2, og tabell 9 viser at stegene utmerket seg ikke noe spesielt til mindre tidsbruk for å løse oppgavene. Av strategiene som deltakerne brukte for løsing av oppgavene (1 og 2) viser det til at strategiene som inneholder steg 5 (strategier (4,5), (3,4,5), (2,3,4,5) og (1,2,3,4,5)) gir gjennomsnittlig høyere riktighetsprosent og steg 2 (strategier (2,4), (1,2,4,5), (2,3,4,5) og (1,2,3,4,5)), bidrar til raskere å løse oppgavene. Steg 2 og steg 5 går igjen på strategi (1,2,3,4,5). Resultatet fra tabell 10 viser at denne strategien utmerket seg best resultat og tidsbruk for å løse oppgavene 1 og 2. Og dermed bekreft dette at Deek et al. (1999) original seksstegstrategi gir mest utelling på besvarelse av oppgavene i dette eksperimentet.

En rekke forskere mener at abstraksjon og dekomposisjon er de viktigste fundamentene til å løse komplekse problemer, og jeg ser at aspekter av abstraksjon og dekomposisjon går igjen i Deek en al. sekssteg problemløsningsstrategi modell. Eksempel henvises til steg 1 (formulering av problemet) tar i bruk av abstraksjon (forberedelse av problem beskrivelse, forberedelser av mental modell og strukturert problem representasjon) og steg 2 (planlegging en løsning) tar i bruk av dekomposisjon (utvikle en plan, kartlegge alternativ løsninger, bryte ned problemer i mindre deler). Dette bidrar til enigheten i allmennkunnskap at abstraksjon og dekomposisjon er en av de viktigste grunnleggende prinsipper for å løse problemer.

6 Validitet

De viktigste farene for validiteten av dette eksperimentet er diskutert i dette kapitlet.

6.1 Eksperimentet, programmet og oppgaver.

Dette eksperimentet består av 3600 LOC medium stor applikasjon. Størrelsen av programmet og oppgavene er mindre enn i arbeidslivet. Det er mulig at resultatet av denne data innsamlingen ville vært annerledes hvis programmet og oppgavene er større. Det er også en mulighet at vi kunne få litt forskjellige data hvis vi hadde større antall oppgaver.

Eksperimentet varte i bare sju timer, noe som kan være for kort tid å bli kjent med applikasjonen. Fremtidige arbeid bør inkludere case studie som varer lengere.

6.2 Programmeringsmiljø

De fleste studentene har ikke erfaringene med programmeringsmiljøet, Jbuilder9. Selv om deltakerne har hatt øvelse i verktøyet, kan mangel av erfaring Jbuilder9 ha en effekt på løsningene. For å redusere trussel av validitet, fikk deltakerne tilbud med Jbuilder dokumentasjon og muligheter for hjelp av observatørene ved tekniske årsaker.

6.3 Deltakere

Alle deltakerne som var med i eksperimentet var studenter. Selv om disse er studenter, kan de forsatt ha forskjellige nivå av erfaringer med programmering, så resultatet kan være forskjellige fra de som har god erfaringer og de som ikke har.

I alt var det 34 deltaker som var med i dette eksperimentet. Data innsamlingen av dette utvalget kan ha innvirkninger med hensyn til forskjellige behandlinger og personal forskjeller mellom deltakerne. Eks er at feedback-collection gruppe må besvare feedback skjemaene.

6.4 Analysen

Dataen av problemløsningsmetoder og problemløsningsstrategier av deltakerne er av kvalitativ data av feedback-collection. En bør være klar over at det kan være noen forskjeller mellom deltakerne. Noen deltakere kan ha glemt å rapportere sin prosess for å løse oppgavene, og dermed kan noen av metodene og strategiene som deltakerne gjorde i eksperimentet vært tapt i listene.

Også å merke seg er at hoved mengden av deltakere ikke gjorde ferdig oppgave 3, dermed er listen av problemløsningsmetoder og problemløsningsstrategi ikke fullstendige.

7 Konklusjon og fremtidig arbeid

Forskningen presentert i denne oppgaven var å identifisere problemløsningsmetoder og strategier brukt av informatikk studenter under prosessen av vedlikeholdsoppgaver på et mellomstort Java applikasjonsystem.

Det var identifisert tre generelle problemløsningsmetoder som studentene brukte for å konfrontere vanskeligheter som oppstod ved løsing av vedlikeholdningsoppgavene. Disse tre metodene var (i) *abstraksjon*, der deltakerne løste et problem ved hjelp av fjerning av unødvendige detaljer, brakte inn nye krav som involverte identifisering av kritiske aspekter ved miljøet og systemet, mens de overså det som var irrelevant. (ii) *Dekomposisjon,* var der deltakerne løste et problem ved å dele problemet inn i mindre sub-problemer, for deretter å løse dem individuelt og kombinerte dem til en samlet løsning. Til slutt har vi (iii) *bygg og fiks,* der deltakerne hoppet rett inn i problemet og løste dem med førsteinntrykksløsninger. Resultatet fra analysen viste at forskjellene ved tidsbruk av problemløsningsmetodene var minimal. Uansett valg av metode tok det gjennomsnittlig en time for deltakerne å komme seg gjennom de enkelte vanskelighetene som Karahasanovic og Thomas (2007) kategoriserte. Resultatet viste også at i gjennomsnittet hadde abstraksjonsmetoden levert noe bedre løsninger enn dekomposisjon og bygg og fiks med 65 % riktighet mot 53 % og 51 % riktighet i løsning av vanskelighetene. Dermed ga metoden abstraksjon noe høyere i riktighetsprosent ved løsning av disse vanskelighetene, mens metodene dekomposisjon og bygg og fiks ga noe lavere riktighetsprosent.

I Arvanitis et al. (2001) studium, påstod de at studentene ingnorerte prosessen av abstraksjon og dekomposisjon i problemløsning av software design i favør av prosessen ”bygg og fiks”. De konkluderte også at abstraksjon og dekomposisjon var viktige ferdigheter, for studentenes framtid i industrien. Det trengtes mer focus på abstraksjon og dekomposisjon innenfor utdanning av programering. Av resultatet i denne oppgaven viste det overenstemmelse med påstanden til Arvanistis et al. at mesteparten av deltakerne tok i bruk av metoden bygg og fiks, framfor abstraksjon eller dekomposisjon. I dette resultatet var det analysen av deltakernes handlinger i løsning av problemer som oppstod i en vedlikeholdningsprosess, istedenfor software design. Resultatet viste også at abstraksjon og dekomposisjon ga noe bedre resultat enn bygg of fiks. Dette kan kansje bekreftet at prosessen av abstraksjon og dekomposisjon var viktige ferdigheter for studenter ved løsning av problemer. For å forbedre studentenes evne i å løse problemer som oppstod i en vedlikeholdningsprosess, var jeg enig i at undervisning av abstraksjon og dekomposisjon i programernigsutdanning kunne være nyttig for forbedring av studentenes problemløsningsevne.

Det er velkjent at strategier for å løse et problem består av en firestegs problemløsningsstrategi. Denne firestegsstrategien består hovedsaklig av (a) forstå problemet, (b) konstruer en plan, (c) utføre planen og (d) se tilbake. Senere utdypet Deek et al. (1999) to ekstra steger i strategien for å bedre løse problemer i programutvikling. Strategien bestod da av (1) formulering av problemet, (2) planlegging av løsningen, (3) konstruering av løsningen, (4) oversettelse av løsningen, (5) testing avløsningen og (6) levering av problemet. Forskningen i dette eksperimentet viste til at strategier med integrasjon av følgende prinsipper; (2) *planlegging av løsningen* og (5) *testing av løsningen*, bidro til høyest riktighetsprosent og lavest tidsbruk ved løsning av forandringsoppgavene gitt i eksperimentet. Resultatet viste også at de som hadde utført alle disse seks strategistegene har i gjennomsnittet fått mest riktighetsprosent av besvarelsen og noe lavere tidsbruk for begge oppgave 1 og oppgave 2.

Studier av problemløsningstrategi, har ofte vært innenfor utvikling av design modeller eller metoder for å hjelpe studenter til å konfrontere problemer (Deek, Turoff, & McHugh, 1999). Et eksempel på dette var Kuro et al. (2004) studium, der de baserte på en firestegs problemløsningsstrategi (a, b, c og d), foreslo en metode for å kalkulere vanskeligheten av problemer. De mente at vanskeligheten av problemer bidro til å fremheve problemløsningsstrategier i basis problemer. Deek et al. (1997) påstod at ferdigheter i problemløsning var essensielt for forståelse av fundamentalene i programmering, og dette burde undervises ved siden av studiet i programmering. I denne oppgaven tok jeg i bruk av en seksstegs problemløsningsstrategi (1, 2, 3, 4, 5 og 6), som var definert av Deek et al. (1999), for å analysere deltakernes handlinger i prosessen for å løse vedlikholdningsoppgaver i dette eksperimentet. Av resultatet, viste det til at denne seksstegs problemløsningsstrategi ga positive resultater i en vedlikeholdningsprosess, og dermed var jeg enig i at undervisning om problemløsningstrategier i programmering kan kansje forbedre studenters oppgaveløsning av vedlikeholdningsoppgaver.

7.1 Fremtidig arbeid

Analysen brukt i dette eksperimentet var av feedback-collection. I fremtidige studier bør informasjoner av *gruppe intervju*ene av deltakerne, og *loggene* av GRUMPs brukes for å få et rikere bilde og bedre forståelse av deltakernes handlinger. Handlinger som hvordan deltakerne løste vanskeligheter som oppstod og *hvorfor* de gjorde det slik, og handlinger som hvordan de løste vedlikeholdningsoppgavene og *hvorfor* de gjorde det slik.

Et deltajert studie om deltakernes bruk av dokumentasjon ville også vært veldig nyttig. Dette kan forklare og gi innsikt i hva deltakerne leste, hvorfor de gjorde det og hvordan dokumentasjonen hjalp dem for å løse oppgavene og problemene som oppstod da de løste oppgavene.

8 Referanser

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* , 84-88.

Arisholm, E., & Sjoberg, D. I. (2004). Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software. *IEEE Transactions on Software Engineering. Vol 30, issue 8* , 521-534.

Arisholm, E., & Sjøberg, D. (2002). A web-based support environment for software engineering experiments. *In Nordic Journal of Computing. Vol. 9. Issue 4* , 231-247.

Arisholm, E., & Sjøberg, D. (2001). Assessing the changeability of two object-oriented design alternatives - a controlled experiment. *Empirical Software Engineering. Vol 6 issue 3* , 231-277.

Arvanitis, T. N., Todd, M. J., Gibb, A. J., & Orihashi, E. (2001). Understanding students' problem-solving performance in the context of programming-in-the-small: an ethnographic field study. *Proceedings of the Frontiers in Education Conference, 2001. 31st Annual - Volume 02* , F1D-20-3vol.2.

Basili, V. R., & Perricone, B. T. (1984). Software errors and complexity: an empirical investigation0. *Communications of the ACM. Vol 27, issue 1* , 42-52.

Benjamin, P., Erraguntla, M., Delen, D., & Mayer, R. (1998). Simulation modeling at multiple levels of abstraction. *Proceedings of the 30th conference on Winter simulation* , 391-398.

Bennett, K. H., & Rajlich, V. T. (2000). Software Maintenance and Evolution: a Roadmap. *Proceedings of the Conference on The Future of Software Engineering* , 73-87.

Bourdoncle, F. (1993). Abstract debugging of higher-order imperative languages. *ACM SIGPLAN Notices. Vol 28, issue 6* , 46-55.

Brooks, R. (1983). Towrads a thoery of the comprehension of computer programs. *Int. J. Man-Mach. Stud. Vol 18 issue 6* , 543-554.

Burkhardt, J. -M., Detienne, F., & Wiedenebeck, S. (1998). The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies. *Program Comprehension, 1998. IWPC '98* , 82-89.

Burkhardt, J.-M., & Détienne, F. (1995). An empirical study of software reuse by experts in object-oriented design. *Human-Computer Interaction (cs.HC)* , 133–138.

Chmiel, R., & Loui, M. C. (2003). An integrated approach to instruction in debugging computer programs. *Frontiers in Education, 2003. FIE 2003. 33rd Annual. vol 3* , S4C-1-6.

Chmiel, R., & Loui, M. C. (2004). Debugging: from novice to expert. *Proceedings of the 35th SIGCSE technical symposium on Computer science education* , 17-21.

COOL. (2005). *InterMedia COOL homepage*. Hentet August 25, 2008 fra InterMedia COOL homepage: http://www.intermedia.uio.no/home/projects/research-projects-1/c00l

Corritore, C. L., & Wiedenbeck, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies, Volume 54, Number 1* , 1-23.

Dansereau, D. F., Collins, K. W., McDonald, B. A., Holley, C. D., Diekhoff, G., & Evans, S. H. (1979). Development and evaluation of an effective learning strategy. *Journal of educational psychology. Vol:79, issue 1* , 64.

Deek, F. P., MeHugh, J. A., Hiltz, S. R., Rotter, N., & Kimmel, H. (1997). On the evaluation of a problem solving and program developmentenvironment. *Frontiers in Education Conference, 1997. 27th Annual Conference. 'Teaching and Learning in an Era of Change'. Proceedings.* , 1249.

Deek, F. P., Turoff, M., & McHugh, J. A. (1999). A common model for problem solving and program development. *IEEE Transactions on Education. Vol 42, Issue 4* , 331-336.

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: a survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers, Volume 9, Number 1* , 47-72.

Dewey, J. (1997). *How we think.* Boston: Dover Publications, Inc.

Ebrahimi, A., & Schweikert, C. (2006). Empirical study of novice programming with plans and objects. *Working group reports on ITiCSE on Innovation and technology in computer science education* , 52-54.

Eierman, M. A., & Dishaw, M. T. (2007). The process of software maintenance: a comparison of object-oriented and third-generation development languages. *Journal of Software Maintenance: Research and Practice Software Focus. Vol 19, issue 1* , 33-47.

Eliasson, J., Westin, L. K., & Nordström, M. (2006). Investigating students' confidence in programming and problem solving. *Proceedings. Frontiers in Education. 36th Annual Conference* , 22-27.

Eriksson, H.-E., & Penker, M. (1997). *UML toolkit.* New York, NY, USA: John Wiley & Sons, Inc.

Fleury, A. E. (2000). Programming in java: student-constructed rules. *ACM SIGCSE Bulletin. Vol 32, Issue 1* , 197-201.

France, R., & Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. *International Conference on Software Engineering* , 37-54.

Gagne, E. D., Yekovich, C. W., & Yekovich, F. R. (1993). *The cognitive psychology of school learning.* New York: New York : HarperCollins College Publishers, c1993.

Gallagher, K. B., & Lyle, J. R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* , 751-761.

Gannon, J. D., Hamlet, R. G., & Mills, H. D. (1987). Theory of Modules. *Software Engineering, IEEE Transactions on. Vol SE-13, issue 7* , 820-829.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering, 2nd edition.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

Hoffman, D., & Strooper, P. (1995). State abstraction and modular software development. *ACM SIGSOFT Software Engineering Notes* , 53-61.

Hundhausen, C. D., Brown, J. L., Farley, S., & Skarpas, D. (2006). A methodology for analyzing the temporal evolution of novice programs based on semantic components. *Proceedings of the 2006 international workshop on Computing education research* , 59-71.

JBuilder. (2008). *JBuilder product page*. Hentet August 2008 fra JBuilder product page: http://www.codegear.com/products/jbuilder

Jones, C. T. (1998). *Estimating Software Costs.* New York NY: McGraw Hill.

Kaminski, D. M. (1988). An analysis of advanced C.S. students' experience with software maintenance. *Proceedings of the 1988 ACM sixteenth annual conference on Computer science* , 546 - 550.

Karahasanović, A., & Thomas, R. C. (2007). Difficulties experienced by students in maintaining object-oriented systems:an empirical study. *Proceedings of the ninth Australasian conference on Computing education - Volume 66* , 81-87.

Karahasanovic, A., Anda, B., Arisholm, E., Hove, S., Jørgensen, M., Sjøberg, D., et al. (2005). Collecting feedback during software engineering experiment. *Empirical Software Engineering. Vol 10 issue 2* , 113-147.

Karahasanović, A., Levine, A. K., & Thomas, R. (2007). Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software. Vol 80, issue 9* , ISSN:0164-1212.

Koenemann, J., & Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology* , 125-130.

Kramer, J. (2007). Abstraction – the key to Computing? *Communications of the ACM. Volume 50 , Issue 4* , 36-42.

Kramer, J., & Hazzan, O. (2006). The Role of Abstraction in Software Engineering. *ACM SIGSOFT Software Engineering Notes. Vol 31, issue 6* , 38-39.

Kuro, R., Lien, W.-P., Chang, M., & Heh, J.-S. (2004). Analyzing Problem's Difficulty based on Neural Networks and Knowledge. *Educational technology & society. Vol 7, issue 2* , 42-50.

Kværn, K. (2006). *Effects of Expertise and Strategies on Program Comprehension in Maintenance of Object-Oriented Systems: A controlled experiment with professional developers.* Masteroppgave: University of Oslo.

Larsen, T. J., & Naumann, J. D. (1992). An experimental comparison of abstract and concrete representations in systems analysis. *Information and Management. Vol 22, issue 1* , 29-40.

Levine, A. K. (2005). *A Study of Comprehension Strategies and Difficulties by Novice Programmers Performing Maintenance Tasks on Object-Oriented Systems.* Masteroppgave: University of Oslo.

Lewis, J. A., Henry, S. M., Kafure, D. G., & Schulman, R. S. (1991). An Empirical study of the Object-Oriented Paradigm and Software Reuse. *ACM SIGPLAN Notices Vol 26, issue 11* , 184-196.

Lientz, B. P., & Swanson, B. (1980). *Software Maintenance Management.* MA: Addison Wesley.

Linn, M. C., & Clancy, M. J. (1992). The case for case studies of programming problems. *Communications of the ACM. Vol 35, issue 3* , 121-132.

Liskov, B., & Guttag, J. (1986). Abstraction and specification in program development. *Mit Electrical Engineering And Computing Science Series* , 469.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIG6, Issue 3CSE Bulletin. Vol 3* , 119-150.

Littman, D. C., Pinto, J., Letovsky, S., & Soloway, E. (1986). Mental models and software maintenance. *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers* , 80-98.

Mayer, R. E. (1981). *The promise of cognitive psychology.* New York: New York: Free Man.

Mayrhauser, A. v., & Vans, M. A. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer Vol. 28, Issue 8.* , 44-55.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCE Bulletin. Vol 33, issue 4* , 125-180.

McIver, L. (2000). The Effect of Programming Language on Error Rates of Novice Programmers. *In A.F. Blackwell & E. Bilotta (Eds). Proc. PPIG 12* , 181-192.

Maani, K. E., & Maharaj, V. (2004). Links between systems thinking and complex decision making. *System Dynamics Review. Vol 20, issue 1* , 21-48.

Nguyen, M. L. (2008). *Analyse av empiribasert modeller for programforståelse innenfor vedlikehold av objektorienterte systemer.* Masteroppgave: University of Oslo.

Novak, J. D., & Gowin, D. B. (1984). *Learning How to Learning.* New York: Cambridge Univeristy Press.

Or-Bach, R., & Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *ACM SIGCSE Bulletin. Vol 36, issue 2* , 82-86.

Pennington, N. (1987). Comprehension strategies in programming. *Empirical Studies of programmers: second workshop, Ablex Publishing Corp* , 100-113.

Pohthong, A., & Budgen, D. (2001). Reuse strategies in software development: an empirical study. *Information and Software Technology. Vol 43, issue 9* , 561-575.

Polya, G. (1962). *Mathematical Discovery.*

Prechelt, L., Unger, B., Tichy, W. F., Brossler, P., & Votta, L. G. (2001). A controlled experiment in maintenance: comparing design patterns to simpler solutions. *Software Engineering, IEEE Transactions on vol 27, issue 12* , 1134-1144.

Repenning, A., & Summer, T. (1994). Programming as problem solving: a participatory theater approach. *Proceedings of the workshop on Advanced visual interfaces* , 182-191.

Rilling, J., & Klemola, T. (2003). Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexty Metrics. *Program Comprehension, 2003. 11th IEEE International Workshop on* , 115-124.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A Review and discussion. *Computer Science Education. Vol 13. No. 2* , 137-172.

Sebastia, L., Onaindia, E., & Marzal, E. (2006). Decomposition of planning problems. *AI Communications. Volume 19, Number 1/2006* , 49-81.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM. Volume 29 , Issue 9* , 850-858.

Someren, M. W. (1990). What's wrong? Understanding beginners' problems with Prolog. *Instructional Science. Volume 19, Numbers 4-5* , 257-282.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM. Vol 29, Issue 7* , 624-632.

Storey, M.-A. (2005). Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on* , 181-191.

Taij, J. (2005). *Controlled Experiment to Investigate the Correlation between Keystroke Latency and Programming.* Masteroppgave: The University of Western Australia.

Thomas, R. (2005). The comprehension of object-oriented systems, applicastion to uwa human research ethics committee.

Thomas, R., & Kennedy, G. E. (2003). Generic usage monitoring of programming students. *Proceedings of the 20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE)* , 715-719.

Tonella, P. (2003). Using a concept lattice of decomposition slices for program understanding and impact analysis. *This paper appears in: Software Engineering, IEEE Transactions on Vol 29, issue 6* , 495-509.

Tubaishat, A. (2001). Knowledge Base for Program Debugging. *Computer Systems and Applications, ACS/IEEE International Conference on. 2001* , 321-327.

Tømmerberg, G. (2006). *Comprehension-Related Activities during Maintenance of Object-Oriented Systems: An In-Depth Study.* Masteroppgave: University of Oslo.

Vee, M.-H. N., Meyer, B., & Mannock, K. L. (200X). Empirical study of novice errors and error paths. *Object-oriented programming-Pedagogy. CS Ed Research. Curriculum Issue* .

Verelst, J. (2005). the Influence of the Level of Abstraction on the Evolvability of Conceptual Models of Information Systems. *Empirical Software Engineering, vol 10* , 467-494.

von Mayrhauser, A., & Vans, A. M. (1997). Hypothesis-driven understanding processes during correctivemaintenance of large scale software. *International Conference on Software Maintenance, 1997.* , 12-20.

von Mayrhauser, A., & Vans, A. M. (1995). Industrial experience with an integrated code comprehension model. *Software Engineering Journal. Vol 10, Issue 5* , 171-182.

von Mayrhauser, A., & Vans, M. A. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer Volume 28, Issue 8* , 44 - 55 .

Webster. (1966). *Webster's Third New International Dictionary.*

Yokomoto, C. F., Buchanan, W. W., & Ware, R. (1995). Problem solving: an assessment of student attitudes, expectations,and beliefs. *Frontiers in Education Conference, 1995. Proceedings. Vol 2* , Session 3c3.

Zweben, S. H., Edwards, S. H., Weide, B. W., & Hollingworth, J. E. (1995). The effects of layering and encapsulation on software development cost and quality. *Software Engineering, IEEE Transactions on Vol 21, issue 3* , 200-208.

Appendiks A – Tabell av deltakere og problemløsnignsmetode

Deltakere og problemløsningmetoder og vanskeligheter (sub-kategorier).

	sub-kategori		
Subjekt_id	**abstraksjon**	**dekomposisjon**	**bygg og fiks**
1	---	1.4	---
2	1.1	2.1	---
8	---	---	1.1, 2.1
9	1.1, 2.1, 2.2	---	---
11	---	---	1.1, 1.2
13	1.1, 2.1, 2.2	1.2, 1.4	---
14	---	---	1.1, 2.1
18	---	---	---
19	2.2	1.4, 2.2	1.1, 1.3, 2.1
20	---	---	1.1, 2.1
25	---	---	1.1, 2.1
30	---	---	1.2
31	2.1	1.1, 2.1	---
33	---	2.2	1.1, 1.2, 2.1
37	1.3, 2.2	---	1.1, 2.1

Appendiks B – Tabell av deltakere og problemløsningstrategi

Deltakere og problemløsningstratgi

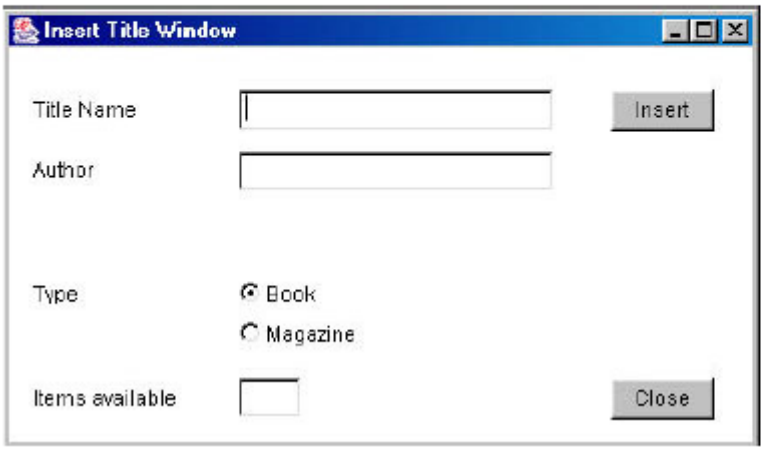
Subjekt_id	oppgave1	oppgave2
1	1,2,3,4,5	1,2,4
2	1,2,3,4,5	3,4
8	1,2,3,4	3,4,5
9	1,4	4
11	3,4	1,2,3,4,5
13	1,2,4	1,2,3,4,5
14	1,2,4	4
18	2,3,4,5	3,4
19	3,4,5	4
20	1,2,4,5	---
25	4,5	3,4
30	3,4,5	2,3,4
31	2,3,4	---
33	2,4	4
37	1,2,4,5	4

Appendiks C – Eksperimentet

Oppgavetekstene og beskrivelser av dem er tatt fra Kaja Kværn (Kværn, 2006) studium.

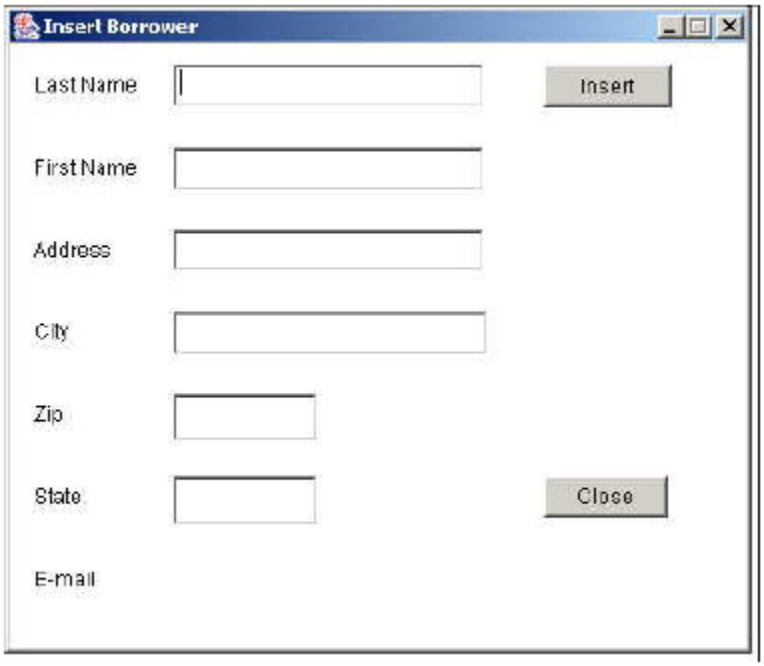
Task 1

Until now the ISBN-numbers have been stored in the system. A new international system for categorisation has been accepted, so this information is no longer needed. You must remove ISBN information from all the places where it has been used (you may comment it out). Also remember to remove all ISBN-text fields from the user interface. Here is an example on how one of the windows should look when you complete the task:



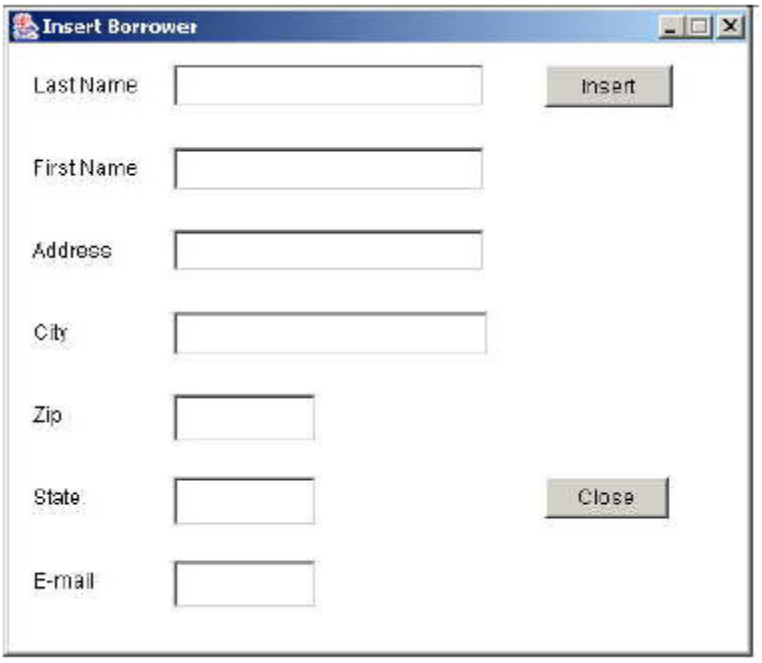
Task 2A

You are going to add the text "E-mail" to the window to insert new borrowers. You don't have to create a new text field for reading the E-mail in this part of the task. The text "E-mail" should be placed under "State" as shown in the picture:



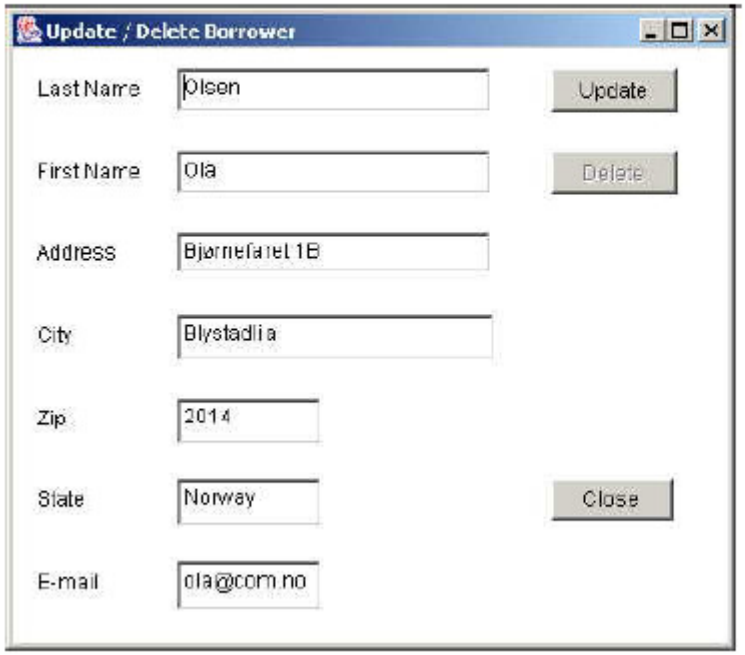
Task 2B

You should add a text field for E-mail to the class that contains data regarding borrowers. To write this field to the file and read from it you have to change the methods read() and write(). You must change the window for inserting borrowers so that E-mail can be entered. Remember to remove the *.dat-files before testing. The window should look like the screenshot below. You can test if you have written E-mail to the file by opening the .dat-file in Notepad.



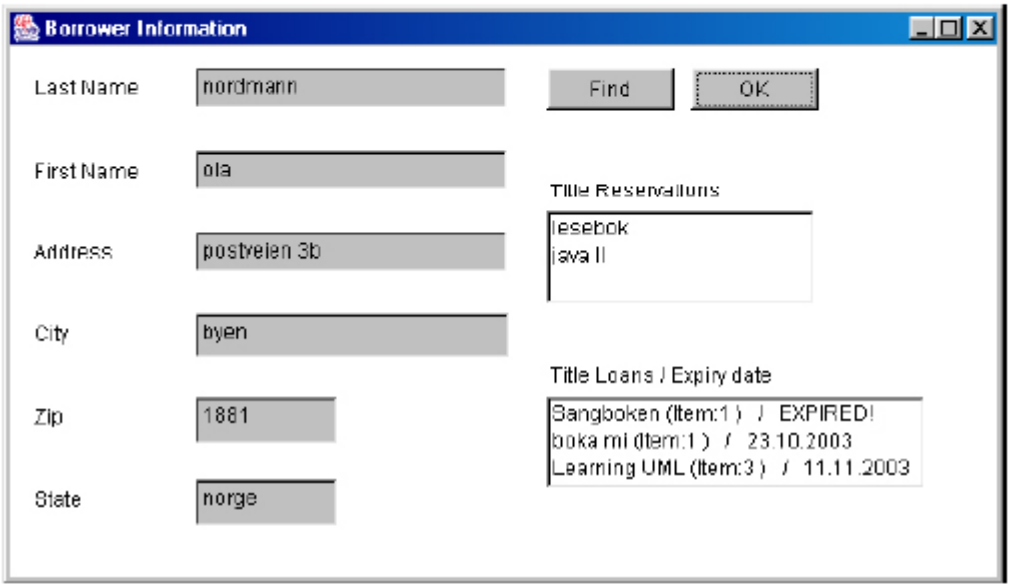
Task 2C

You should now change the window for updating borrower so that E-mail can be shown. The window should look like the screenshot below. Then you should change all other windows that show borrower's information in the same way.



Task 3

You should add functionality for presenting the borrower loan due. The loan is due 4 weeks after the loan was made. You are going to show in the BorrowerInfoWindow-class that a borrower's loan has expired. An illustration of the window after new functionality has been added is shown below:



You can use Java's Calendar-class http://java.sun.com/j2se/1.4.2/docs/api/java/util/Calendar.html) to handle dates (`java.util.Calendar`).

`Calendar.getInstance()` returns a GregorianCalendar-object where the time fields are initialized with date and time.

Some examples (you might need some methods in this task):

```
Calendar rightNow = Calendar.getInstance();
int year = rightNow.get(Calendar.YEAR);
int day = rightNow.get(Calendar.DATE);
int month = rightNow.get(Calendar.MONTH) + 1;

// Subtract 5 days from the date
rightNow.add(Calendar.DATE, -5);

// Returns true when rightNow is earlier than baseCal
boolean b = rigthNow.before(baseCal);

// Set the Calendar to a specific date (year, month, day,
hour, minute, second)
rightNow.set(2003, 7, 21, 10, 30, 30);
```

Beskrivelser av oppgavene

Task 1

This was an alteration task which did not require the participant to make any addition of code, just remove some of the existing code. However, the application was rather large, and the task required the participants to make changes several places in the code. Although the task required changes in quite a few classes, it could be done simply by performing a search for ISBN through all the files (classes) and deleting the findings. From the total of 27 classes in the application, there were 5 classes containing ISBN.

Table 26 shows an overview of the classes that had to be altered in this task, with a description of what had to be done for each of the classes.

Table 26: Necessary alterations for Task1

File name	Changes required
Title.java	This is the entity object in the BO-package where ISBN was stored. Changes needed on: - Constructor - Get/set methods - Read and write methods for persistency - A find method (if else)
FindTitleDialog.java	UI-package. Changes needed - Fields and labels with corresponding getText/setText methods - Declarations - FindButton_Clicked
TitleFrame.java	UI-package. Changes needed: - Fields and labels with corresponding getText/setText methods - Declarations - AddButton_Clicked
TitleInfoWindow.java	UI-package. Changes needed: - Fields and labels with corresponding setText method - Declarations
UpdateTitleFrame.java	UI-package. Changes needed : - Fields and labels with corresponding getText/setText methods - Declarations - UpdateButton_Clicked

Task 2

This task was divided into three parts to make it somewhat less complex for the participants:

The first part was merely to add a piece of text to the user interface. A screen shot of the window to be altered was given in the task text, so the participants could easily see in which class to do the editing.

The second part was a bit more complex. The participants should add an input field to the UI-class they had already altered in the former part. Also, they had to alter two methods in the entity class – read and write - to make sure that E-mail could be saved to file. They were given a hint to which class they had to alter ("the class containing information about a borrower"), and also which methods to alter ("read" and "write").

The third and last part was the most extensive of the three parts. The participants had to alter all the windows in the user interface which contained information about E-mail. However, the alteration task itself was similar to the one performed in the previous parts of this task. Table 27 shows an overview of the alterations that had to be made.

Table 27: Necessary alterations for Task 2

File name	Changes required
BorrowerInformation.java	This is the entity object in the BO-package where E-mail should be stored. Changes needed on: • Constructor • Get/set methods • Read and write methods for persistency
BorrowerInfoWindow.java.	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations
BorrowerFrame.java	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations
FindBorrowerDialog.java	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations
UpdateBorrowerFrame.java	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations

Task 3

This task was the most complex of the three. The participants had to introduce a new variable containing the date of a loan, and this variable had to be saved and loaded from file. The participants were given no hints as to where to place this variable (in which class). They also had to change the user interface, calculate an expiry date (4 weeks after the loan was made)

and include logic to check the expiry date against today's date. They were given the name of the class in which to do this. What also complicated this task was that they had to use a class – Calendar – which was previously unknown for most of the participants. Some information on the usage was given in the task text, but not all the methods required solving the task.

Table 28 shows an overview of the alterations that had to be made during Task 3

Table 28: Necessary alterations for Task 3

File name	Changes required
Loan.java	This is the entity object in the BO-package where the loan date should be stored. Changes needed on: • Constructor • Get/set methods • Read and write methods for persistency • Logic for calculating expiry date
BorrowerInfoWindow.java	UI-package. Changes needed: • Method for displaying the expiry date in the window