UNIVERSITETET I OSLO

Institutt for informatikk

Automatisk og manuell skåring av Javaoppgaver i programmeringstester: En prototypeimplementasjon av persistens for bruk i ferdighetstester

Masteroppgave

(30 studiepoeng)

Linda Prytz Sørlie

10. desember 2007

Oppsummering

Det finnes mange ulike rammeverk for automatisk å evaluere programmeringsoppgaver. Slik programvare er blitt utbredt de siste årene og brukes for eksempel innen utdanning, forskning og programmeringskonkurranser for å vurdere programmeringsferdigheter. Denne oppgaven inngår som et ledd i et større prosjekt ved Simula Research Laboratory som har som formål å bedre kunne vurdere ferdigheter hos profesjonelle programvareutviklere.

Et problem er imidlertid hvordan man skal skåre en oppgave. Det eksisterer mange alternativer for hvordan en prestasjon skal vurderes og deretter skåres, og dette er også avhengig av motivasjonen som ligger bak bruken av rammeverket. Det er også utfordringer med hvordan man skal legge sammen skår fra enkeltoppgaver for å gi en sum som skal kunne fortelle noe generelt om totalprestasjonen til den enkelte programmerer.

Målsetning for denne oppgaven er å utvikle en prototype i Java, basert på åpen kildekode verktøy, som støtter ulike fremgangsmåter for å skåre en oppgave. For å tilby mest mulig fleksibilitet vil fokus være på hvordan kriteriene som brukes i skåringen kan gjøres persistente.

Løsningen som foreslås gjør et klart skille mellom konseptene skår og evaluering og lagrer disse separarat i en database. En evaluering er da en objektiv og verdinøytral observasjon, mens en skår er en verdivurdering av resultatet fra en evaluering. Dette gjør det mulig å skåre hver enkelt oppgave på flere ulike måter. Man kan deretter empirisk sammenligne disse alternativene for å se hvilke fremgangsmåte som er mest hensiktsmessig i hvert spesifikke tilfelle. I tillegg er det mulig å sette sammen skår fra hver enkelt oppgave på forskjellige måter i en totalvurdering. Ved å modellere hver enkelt skårsetter separat er det også mulig å sammenligne skårsettere for oppgaver.

Prototypeimplementasjonen i denne oppgaven er godt egnet for å støtte forskjellige krav til automatisk og manuell vurdering og skåring av programmeringsoppgaver innen forskning og utdanning. Den er også enkel å vedlikeholde, teste og videreutvikle.

Innholdsfortegnelse

Figurer

Tabeller

1 Introduksjon

Automatisk og semi-automatisk vurdering av programmeringsoppgaver er blitt vanligere i de senere år [1].

En automatisk vurdering innebærer at en maskin eller en type programvare utfører en vurdering basert på et sett kriterier. For eksempel kan man ønske å måle hvor funksjonelt korrekt en løsning er. En løsning med få feil vil derfor være mer funksjonelt korrekt enn en med mange feil, gitt at det ikke tas hensyn til om noen av feilene er mer alvorlige enn andre. De mest kjente fordelene ved å bruke denne formen for vurdering er hurtighet, tilgjengelighet, konsistens og objektivitet [2]. Likevel er automatisk vurdering mest egnet for små oppgaver som tar for seg et avgrenset tema. Dette kan resultere i at det fokuseres mer på områder innenfor programmering som enkelt lar seg måle, mens mer kompliserte områder overses [3].

En manuell vurdering av en programmeringsoppgave innebærer at en person bedømmer programmeringskode. Manuelle vurderinger er godt egnet i tilfeller der en programmeringsoppgave berører mange ulike temaer [2]. Det er også hevdet at personlig tilbakemelding som gis ved en manuell vurdering er viktige motivasjonsfaktorer for mange mennesker [4]. Ulemper ved denne typen vurderinger er at de krever mye tid og ressurser, og i mange utdanningsinstitusjoner kuttes det derfor ned på antall oppgaver som gis til studenter i løpet av et semester [1]. Manuelle vurderinger øker også sannsynligheten for at feil og detaljer i koden kan bli oversett [5]. Imidlertid finnes det aspekter ved kode som kun kan vurderes manuelt, som for eksempel kvalitet på dokumentasjon av kode [3]. Det vanligste problemet for denne typen vurderinger er likevel mangel på enighet mellom de som foretar vurderinger. Uavhengige vurderinger av den samme programmeringskoden vil nesten alltid gi ulike resultater avhengig av hvem som har utført vurderingen [5].

En semi-automatisk vurdering er en kombinasjon av automatisk og manuell vurdering og brukes gjerne for å se på store og komplekse oppgaver [2].

1.1 Eksisterende rammeverk

For automatisk å vurdere programmeringsoppgaver finnes det mange ulike rammeverk. Eksempler på verktøy som brukes er ”Online Judge” [5], ”HoGG” [6], ”Topcoder” [7] og ”JKarelRobot” [8]. Disse rammeverkene brukes hovedsaklig for å teste studenter og er økonomiske og effektive hjelpemidler for å håndtere store antall besvarelser [6]. Ved at hver enkelt vurdering blir mindre tidkrevende er det også mulig å gi studentene flere, mindre og mer emnespesifikke oppgaver fremfor store oppgaver der mange ulike temaer er blandet sammen [5]. Andre typer rammeverk, som for eksempel ”Watcher” [9], brukes for å kartlegge hvordan programmerere disponerer tiden de har tilgjengelig for å løse en oppgave. Analyser av tastetrykk, museklikk og endringer i dokumenter brukes for å avdekke hvilke områder av koden utvikleren har brukt mest tid på. Slike typer rammeverk brukes oftest for å gjennomføre kontrollerte eksperimenter for bruk innen forskning eller for å gjøre analyser i industrien.

De ulike anvendelsesområdene gjør at rammeverkene dermed også er myntet på ulike menneskegrupper, hovedsaklig studenter eller programvareutviklere. Det er også ulik motivasjon som ligger til grunn for bruken av disse verktøyene, både i hva som er hensikten med målingene og hva som måles. Utdanningsinstitusjoner ønsker at studentene skal øke eget ferdighetsnivå. Det gis derfor ofte oppgaver der det testes at et program fungerer i henhold til en spesifikasjon som er utarbeidet i tråd med læringsmålene i et kurs. De industrielle rammeverkene er derimot mer opptatt av å observere og måle ressursbruk under arbeidet med en programmeringsoppgave. Det tas her for gitt at utviklerne allerede kan produsere fungerende kode. Resultatene kan brukes til å oppdage mønstre og avvik i arbeidsmetoder blant ulike typer programmerere

(for eksempel mellom junior og seniorutviklere), eller de kan brukes til å avdekke styrker og svakheter hos den enkelte utvikler. Disse opplysningene kan igjen være nyttige i estimering og ressursallokering for programvareprosjekter i industrien. Rammeverkene kan også brukes i intervjusituasjoner for å teste kandidater.

1.2 Vurderingskriterier

Det er to viktige aspekter som må holdes adskilt ved vurderinger som utføres i programvarerammeverk: (1) Vurderinger for den enkelte oppgave og hva denne måler (2) vurderinger som gjør det mulig sette sammen (aggregere) enkeltoppgaver til en totalvurdering.

For enkeltoppgaver virker ulike former for motivasjon bak målingene inn på hvilke vurderingskriterier som brukes i de respektive rammeverkene. Eksempler på egenskaper ved programmeringskode som måles er tid og kvalitet. Å måle tiden det tar å løse en oppgave er en objektiv observasjon og kan i hovedsak kun måles på én måte. Derimot er kvalitet et abstrakt begrep som må brytes ned i mindre, mer definerte kriterier som for eksempel korrekthet, robusthet og effektivitet for å kunne måles. Disse må igjen operasjonaliseres med spesifikke verktøy eller metoder for å kunne gi håndfaste målinger av kode eller programmer. Det finnes i tillegg mange ulike metoder for å operasjonalisere et vurderingskriterie. Rammeverkene som finnes i dag kan derfor vurdere programmeringskode etter de samme kriteriene, men fordi det brukes ulike metoder for å gjøre målingene har vi likevel ingen garanti for at resultatene er konsistente på tvers av rammeverkene. Dette gjør det vanskelig å sammenligne verdivurderinger fra flere ulike rammeverk på en fornuftig måte.

Basert på resultatene fra vurderinger av enkeltoppgaver kan en person tildeles en poengsum eller sumskår for en prestasjon. Ved å aggregere resultatene fra vurderinger av hver enkelt oppgave vil vi få en endelig skår som skal kunne si noe om ferdighetene til en programmerer sett i lys av hva som er formålet med testen. I denne aggregeringen er det viktig å ha et bevisst forhold til om de ulike målingene skal sees på som likeverdige, eller om noen aspekter er mer interessante enn andre. Det er et sterkt avhengighetsforhold mellom hvilke vurderingskriterier som brukes, hvordan disse vektlegges og hva som blir utfallet av testen. Man bør derfor ha klart for seg hvordan skårene er bygget opp før man trekker slutninger av resultatene.

1.3 Spesifikk anvendelse

Arbeidet med denne oppgaven inngår i et større prosjekt ved Simula Research Laboratory. Prosjektet går ut på å utvikle et rammeverk, Java Computerized Assessment Test (JCAT), for automatisk, semi-automatisk og manuell vurdering av programmeringsoppgaver. Formålet med JCAT er å kunne vurdere ferdigheter hos programvareutviklere. Rammeverket hadde i utgangspunktet delvis støtte for mål som tid, subjektiv skåring og funksjonell korrekthet, men som vi vil vende tilbake til i neste seksjon er et av formålene ved denne oppgaven å utvide dette.

En type anvendelse av rammeverket er å bruke sumskåren til å kunne vurdere hva det for eksempel vil si å være dyktig som programmerer. For å måle dette kan vi bruke tester. Tester brukes også innen utdanning eller programvareindustrien, men JCAT skal hovedsaklig brukes i forskning for å gjennomføre kontrollerte eksperimenter.

1.4 Problemstilling

Denne oppgaven tar sikte på å kombinere elementer fra eksisterende rammeverk for automatisk evaluering av programmeringsoppgaver for å utvikle en prototype som kan brukes for å måle ferdigheter hos

programmerere. Ferdighet er et abstrakt begrep og for å kunne måle dette er vi derfor nødt til å brekke det opp i mindre, målbare elementer som sammenlagt kan si noe om det å være dyktig. For å gjennomføre dette trenger vi å spesifisere et sett med eksakte regler for hva som skal vektlegges i en vurdering. Vi kan deretter gjøre flere analyser av samme kode med disse ulike regelsettene for å se de ulike skårene i sammenheng med hverandre. Dette vil kunne gi en indikasjon på hvilke spesifikke områder en person er dyktig innenfor. Det er også mulig å sjekke om det er samsvar mellom de forskjellige typene vurderinger.

En begrensning ved andre rammeverk i henhold til våre behov for å måle dyktighet er at det ikke alltid er et eksplisitt skille mellom vurderingskriterier og skårer. Det brukes i stedet et fast regelsett for hvilke vurderinger som skal gjøres og hvordan disse skal vektlegges. Resultatet er at den aggregerte skåren kun viser ferdighetene til en person fra ett standard perspektiv som er definert ut i fra hva som er av interesse i akkurat denne testen. Vi kan derfor ikke si noe om de generelle ferdighetene til en person, men kun hvor dyktig en person er til å programmere i henhold til denne spesifikke oppfatningen av hva som bør måles i en test.

For å måle dyktighet i JCAT ønsker vi derfor ytterligere presisjon av regelsettet eller perspektivet som skal brukes for å vurdere og skårsette en oppgave. Gitt at man vet nøyaktig hva man ønsker å måle med en test skal det være mulig for den enkelte testadministrator å spesifisere hvilket perspektiv man ønsker å bruke for å se på ulike typer ferdigheter. Det bør derfor være mulig å:

- Definere egne perspektiver som eksplisitt spesifiserer hvilke egenskaper ved koden som skal vektlegges
- Endre perspektivet for hvordan en oppgave skal vurderes for å kunne se variasjoner i skårene etter hvilket perspektiv som er valgt

Definisjonen av et perspektiv må både inneholde regler for hvilke vurderingskriterier som skal brukes og hvordan resultatene fra vurderingene skal aggregeres i en skår. I tilfeller hvor det er uklart hvordan man kan aggregere resultater på best mulig måte, bør det være mulig å raskt gjennomføre forskjellige alternativer og sammenligne resultatene fra disse. Ved å gjøre flere målinger med ulikt perspektiv kan man få mer informasjon om den enkelte programmerer sine ferdigheter innenfor ulike områder.

For å oppnå disse målene kan vi bruke en objektiv og bevisst tilnærming til avhengighetene mellom vurderingskriterier og skårer. Fremgangsmåten vi bruker for å komme frem til en skår kan sees på som et perspektiv eller verdisystem for å vurdere en prestasjon. Ved å kunne skreddersy dette verdisystemet til våre egne behov kan vi avdekke nyanser i en programmerers ferdigheter innenfor et gitt felt. Det vil dermed kunne være mulig å si noe om en utvikler sine sterke sider, og det blir også enklere å se områder der det finnes rom for forbedringer.

Det finnes mange måter å vurdere en enkelt oppgave på. I tillegg finnes det mange strategier for å aggregere skårene fra disse enkeltoppgavene på. For å enkelt kunne sammenligne alternativer for dette trenger vi å kunne lagre disse systematisk. Rammeverket trenger derfor støtte for persistens.

1.5 Målsetning

Målsetningen for denne oppgaven er:

- Å forstå viktige konsepter og teori relatert til evalueringsmetoder, skåringsmetoder, verdisystemer og manuelle og automatiske vurderinger
    - Beskrive eksisterende konseptuelle og tekniske rammeverk for automatisk vurdering av programmeringsoppgaver og å kartlegge egenskaper ved disse
    - Foreslå en løsning for hvordan kvaliteter fra andre rammeverk kan kombineres med bruk av ulike verdisystemer i et nytt rammeverk (JCAT)
    - Implementere den foreslåtte løsningen, samt evaluere denne etter et sett konseptuelle krav
- Utvikle en prototypeimplementasjon av hvordan persistens kan integreres i JCAT med spesielt fokus på bruk av forskjellige verdisystemer ved vurderinger
- Evaluere denne prototypeimplementasjonen etter spesifikke krav til evaluering og skåring, funksjonelle og ikke-funksjonelle krav, samt komme med forslag til videre arbeid

Denne oppgaven tar ikke sikte på å ta for seg problemstillingen med hvordan man mest hensiktsmessig kan aggregere ulike målinger i en overordnet skår for enkeltoppgaver.

1.6 Struktur på oppgaven

Resten av oppgaven er strukturert som følger: Kapittel 2 tar for seg viktige konsepter, bakgrunn og andres arbeid som denne oppgaven relaterer seg til. Kapittel 3 gir en kort beskrivelse av arbeidsprosessen med oppgaven. Kapittel 4 er kravspesifikasjon og kravanalyse. Det defineres her spesifikke krav til evaluering og skåring, samt funksjonelle og ikke-funksjonelle krav til prototypeimplementasjonen. Kapittel 5 beskriver design for JCAT og implementasjon av prototypen, mens kapittel 6 er evaluering av løsningen som er implementert. Kapittel 7 konkluderer og oppsummerer anbefalingene for videre arbeid.

2 Bakgrunn

Det presenteres her viktige konsepter, bakgrunn og andres arbeid rundt tester og måling av kvalitet for programvare som denne oppgaven relaterer seg til. Det gis deretter en oversikt over eksisterende eksperimenter og rammeverk for automatisk evaluering av programmeringsoppgaver. Til slutt sees det på persistens i applikasjoner.

2.1 Konsepter

Når vi skal vurdere en oppgave for å sette en skår må vi først foreta en evaluering. Forståelse av hva som legges i konseptene evaluering og skåring utgjør til sammen grunnlaget for å forstå hva et verdisystem er.

2.1.1 Evaluering

For å vurdere kvaliteten på en prestasjon eller et produkt kan vi foreta en evaluering. Å evaluere noe kan også sees på som å måle noe og er en mapping fra den empiriske verden til den teoretiske verden. En måling er derfor et nummer eller symbol som gjennom denne mappingen er tilegnet en entitet for å kunne karakterisere en attributt [10, s 25, siterer [11]]. En entitet er et objekt som kan observeres i den virkelige verden [10, s 26]. I denne oppgaven vil evaluering bli brukt som en objektiv og verdinøytral vurdering av et kodesegment.

Innen forskning vil det at noe er objektivt bety at det kan måles til samme verdi uavhengig av hvilken eller hvor mange forskere som måler det samme [12]. Dette er noe som er relativt gjennomførbart dersom man måler ting som høyde, temperatur og lignende, men det er veldig vanskelig å få til når man måler mennesker. Objektivitet er sterkt relatert til at noe er verifiserbart og reproduserbart [12]. Innenfor måling er objektivitet viktig for forskere for å kunne eliminere subjektive oppfatninger mellom individuelle observatører, og det brukes derfor diverse instrumenter for å gjøre målinger fremfor å spørre en person hva han eller hun føler eller tror [12]. Det finnes det også verdier som ikke kan måles etter en mal og kun kan måles subjektivt. Subjektive vurderinger kan også tilføre mye nyttig informasjon som ikke kan fanges opp i objektive vurderinger. For å måle noe på beste og mest detaljrike måte kan vi derfor se på det samme fenomenet fra ulike perspektiv. Vi kan deretter sammenligne resultatene for å se om disse stemmer overens, eller om det er store avvik i målingene.

For å utføre en evaluering trenger vi et sett med evalueringskriterier for å beskrive hva som skal måles. Disse evalueringskriteriene kan være enten subjektive eller objektive. Et evalueringskriterium som for eksempel at et program er ”godt skrevet” er noe som kan få varierende resultat ut i fra hvilken person som utfører evalueringen, og er følgelig subjektivt. Utfallet av evalueringen er avhengig av denne personen sin oppfatning av hva det betyr at noe er ”godt skrevet”. Dersom det er ting som antall kodelinjer eller ressursbruk under kjøring av et program som skal måles er dette noe som gir samme resultat hver gang og kan derfor måles objektivt. For å få nøyaktige og rettferdige målinger er det derfor viktig å finne så objektive kriterier som mulig for å måle subjektive verdier.

2.1.2 Skår

Motivasjon ved å gjennomføre en test er å få informasjon om noe man er interessert i å vite mer om. En test kan måle hvor gode ferdigheter eller kunnskaper personen som tar testen (et subjekt) har innenfor et område, og det gis en poengsum ut i fra dette. Denne poengsummen kaller vi en skår. Noen tester gir kun tilbakemelding om skåren er over en viss poenggrense for å avgjøre om testen ble godkjent eller underkjent. Ofte vil det derimot være mer hensiktsmessig å gi hvert subjekt en egen skår for en prestasjon for å kunne skille resultatene fra hverandre på et mer detaljert nivå. Skåren kan for eksempel brukes for å vurdere hvor godt subjektet forstår kursmateriale eller hvor gode ferdigheter han eller hun har innenfor det området som måles av testen. Som nevnt i kapittel 1 er det likevel viktig å være bevisst på hvilke verdier som er lagt til grunn for å beregne skåren før man trekker slutninger av resultatene fra testen.

Gitt at man har definert et klart formål med testen, vil man med skårer på oppgaver kunne bruke disse for å vurdere i hvor stor grad et subjekt kan frembringe den oppførselen man er interessert i å vurdere. For å fastsette disse skårene kan det settes opp et sett med skårregler for de ulike evalueringene og oppgavene. Ved beregning av delskårer og skårer er det i tillegg til selve evalueringskriteriene også viktig hvordan de ulike evalueringsresultatene og delskårene vektes. Det er derfor viktig å ha en bevisst tilnærming til hvordan fastsettelsen av delskårer og skårer skal foregå. Dersom alle observasjonene sees på som likeverdige kan vi for eksempel velge å addere verdiene eller å regne ut gjennomsnittet for å gi en totalskår. Dersom vi mener at noen av observasjonene er mer viktige enn andre kan vi velge å vekte disse kvalitetene mer. Vektleggingen bør speile hva vi ønsker å måle og hvor relevante de ulike evalueringene er for feltet vi er interessert i.

2.1.3 Verdisystemer

Sett fra et generelt perspektiv kan et verdisystem defineres som noe som refererer til hvordan et individ eller en gruppe organiserer sine etiske eller ideologiske verdier [13]. I kontekst av skårsetting kan et verdisystem derfor sies å være en spesifikasjon av hvilke evalueringskriterier som sees på som viktige i henhold til målet for testen, og hvordan skårreglene skal settes opp. Hvilket verdisystem man velger å bruke avgjør hva slags skår man får ut av systemet, og denne skal kunne brukes for å si noe om det som er formålet med testen. Ulike personer kan ha ulike preferanser for hva som gjør at noe er bedre eller dårligere enn noe annet, og vi kan følgelig si at de har ulike verdisystemer å sette skår etter. Eksisterende rammeverk for automatisk evaluering ser på et utvalg evalueringskriterier som gjenspeiler det utviklerne av rammeverket ser på som mest viktig å måle. Det gjøres ikke noe klart skille mellom evaluering og skåring - disse sees på som to sider av samme sak. Skåren som fastsettes viser dermed bare ett av mange perspektiv av ferdighetene til programmereren, der det kun er enkelte ferdigheter som vektlegges.

2.2 Tester

En test kan sees på som en systematisk prosedyre for å observere oppførsel og for å beskrive denne ved hjelp av numeriske skalaer eller fastsatte kategorier [14, s 32]. Med systematisk menes det at man ved hjelp av testen samler inn informasjon ved å spørre ut eller observere person etter person på lik måte under like forhold (eller sammenlignbare forhold) [14, s32].

Innenfor utdanning defineres en test som noe som brukes for å måle prestasjoner, kunnskaper og ferdigheter. Innen forskning kan en test sees som en måte å verifisere eller falsifisere en forventning basert på en observasjon [15]. Den sentrale ideen bak en test er at en relativt liten del av et individ sin prestasjon, målt under nøye kontrollerte forhold, skal kunne gi et nøyaktig bilde av denne personen sin evne til å prestere under et mye bredere spekter over en lengre periode [16, s 24]. En test kan bestå av en eller flere oppgaver. Disse oppgavene kan være enten flervalgsoppgaver eller oppgaver i fritt format. Et eksempel på tester med løsninger i fast format er flervalgstester der subjekter velger ett av flere mulige svar for en oppgave. For å vurdere programmeringsferdigheter er det derimot mest hensiktsmessig å bruke oppgaver med fritt format der løsningen ofte er en eller flere filer med programmeringskode.

2.2.1 Ulike tolkninger av en test

For å forstå grunnlaget for hvorfor en bestemt skår er gitt er vi nødt for å se på informasjon relatert til oppgaven som er utført, prestasjonene til de andre som har utført testen, eller begge deler [14, s 104]. Det finnes tre hovedtyper av tolkninger som kan legges som basis for en test [14, s 33]:

- Normreferanse: Hvordan skårene er distribuert innenfor en relevant populasjon. Dette kan for eksempel være karakterer i et kurs.
- Domenereferanse: Resultatet sees i forhold til et område i den virkelige verden som testen relaterer seg til, og skal kunne si noe om hvor godt det er forventet at testdeltakeren vil prestere i forhold til alle mulige oppgaver innenfor dette området. Dette kan for eksempel være hvor stor andel av alle mulige annengradsligninger en person vil klare å løse.
- Kriteriereferanse: Hvordan en person er forventet å prestere og hvordan resultatet kan sees i forhold til eksterne kriterier og praktiske krav bak testen, som for eksempel læringsmål og kursmateriale.

Under gjennomføringen av en test vil det alltid også være en viss feilmargin man er nødt til å ta hensyn til. Eksempler på omstendigheter som kan virke inn på testresultatene er faktorer som stress og nervøsitet, eller at en person rett og slett kan ha en god eller dårlig dag. Det er derfor hensiktsmessig at en test omfatter flere oppgaver for å kunne håndtere denne feilmarginen. For å bedre forstå hva og hvor store de enkelte feilmarginene er kan man bruke ANOVA-analyser [17]. ANOVA står for Analysis of variance eller varianseanalyse.

2.2.2 Oppbygning av skårer og delskårer i en test

Skåren for en test er aggregert fra flere delskårer, som igjen er basert på evalueringer. En delskår kan derfor sees på som en tolkning av en evaluering. Ved at det foretas en verdivurdering av en evaluering transformeres resultatet av evalueringen til en delskår som skal kunne si noe om kvaliteten på den aktuelle evalueringen.

2.2.3 Validitet og pålitelighet

For denne oppgaven er hensikten med en test å måle dyktighet innenfor et område. Gitt at en person skårer høyt på en test vil vedkommende bli betraktet som god på sitt felt. Spørsmålet er

om vi er sikre at testen måler nettopp de ferdighetene som vi ønsker å måle. Dersom testen egentlig måler andre ferdigheter enn det vi tror er også resultatene feil. Validiteten til en test vil derfor si at den er konstruert slik at den faktisk måler det som er hensikten at den skal måle. Pålitelighet går derimot på at vi måler noe riktig [16, s 11].

For å kunne stole på resultatene fra en test må vi også forsikre oss om at målingene er pålitelige. Påliteligheten til en test kan undersøkes for eksempel ved å bruke en statisk metode som kalles test-retest. Denne metoden går ut på at en test utføres flere ganger, for eksempel kan den samme testen gis til en gruppe subjekter på to ulike tidspunkt. Hvert subjekt bør skåre forskjellig fra de andre subjektene, men dersom testen er pålitelig bør hvert subjekt få samme resultat for begge testene [18]. Test-retest er noe som er vanlig å bruke for mange andre typer tester, men det er imidlertid vanskelig å bruke for tester som skal måle dyktighet.

2.2.4 Item og Itembank

For å evaluere noe, er vi nødt for å gi en oppgave til et subjekt og motta et svar på oppgaven. Evalueringen skal deretter definere hvor godt dette svaret tilfredsstiller kravene som er gitt i oppgaven. En oppgave kan i denne sammenhengen kalles et item. Et item består av item stimuli som er oppgavebeskrivelsen samt eventuell tilleggsinformasjon, som for eksempel tekst, programmeringskode, bilder eller filmklipp. Svaret som subjektet returnerer til systemet kalles et response. For at en oppgave skal kunne kalles et item må det finnes en skåringregel for oppgaven [16, s.24]. Dette betyr at for enhver valid test må det for hvert eneste item finnes en spesifikasjon av skåring eller kategori som skal tildeles et hvert mulig svar. Dette garanterer ikke nødvendigvis at bruk av et gitt item vil føre til god måling, men det sikrer at vi vet hvordan testen er bygget opp, og vi kan dermed lettere se om itemet bidrar til å måle det som er forventet at testen skal måle.

Et item kan inngå i en itembank som er en samling av alle itemer som eksisterer. I en itembank er alle itemer klassifisert etter innhold eller emne og vanskelighetsgrad. Tester kan deretter bygges opp ved å velge et sett med itemer etter en bestemt plan [14, s 46]. For eksempel kan en vedlikeholdstest legge vekt på subjektive evalueringer av vedlikeholdsoppgaver, mens en programmeringstest for Javakode kan vektlegge dette annerledes ved å inkludere flere oppgaver der man må skrive alt fra bunnen av.

En itembank er spesielt fordelaktig dersom man kjører en test for mange subjekter over en lengre tidsperiode. Det kan da genereres en ulik test for hvert enkelt subjekt. Likevel skal vanskelighetsgraden på testene være den samme, og derfor kan også skårene fra de ulike testene sammenlignes med hverandre. På denne måten øker også testsikkerheten ved at det ikke er mulig å lære seg svarene på oppgavene på forhånd.

2.2.5 Ulike typer tester

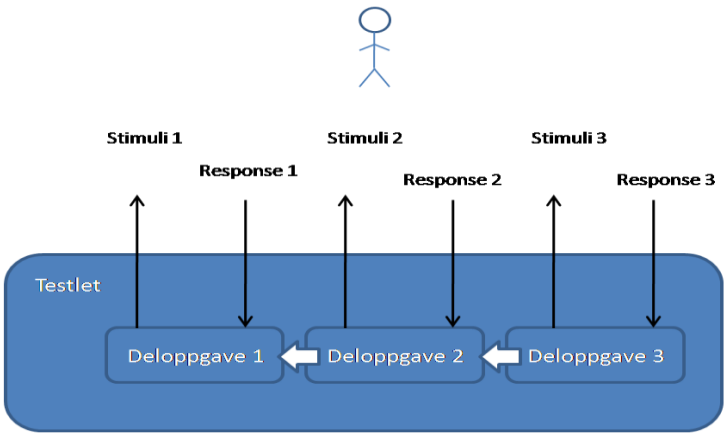
Det finnes tre ulike typer tester som er relevante for denne oppgaven. Disse er eksperiment, testlet og computer adaptive test.

Eksperiment: Et eksperiment er en type test der rekkefølgen på alle oppgavene er fastsatt ved testens start. Under gjennomføringen av et eksperiment får alle subjektene nøyaktig de samme oppgavene. Denne typen tester tar ikke hensyn til ferdighetsnivået til den enkelte deltaker, og

man risikerer at subjekter kan bli sittende og bruke mye tid på oppgaver som er enten for vanskelige eller for lette.

Testlet: En testlet trenger ikke nødvendigvis å være en type test, men det kan også sees på som en oppgavestruktur. En testlet er en gruppe av itemer som relatert til et bestemt område. Den er utviklet som en enhet og har et fast antall predefinerte ”ruter” som et subjekt kan ta gjennom oppgaven [14, s 173]. Testleter brukes som en løsning på problemet med lokal avhengighet [14, s 173]. Dette problemet går ut på at en test kan inneholde to eller flere itemer som er innbyrdes avhengige av hverandre (for eksempel ved at den ene oppgaven bygger på den neste). Hvordan en person gjør det på den andre oppgaven er derfor ikke statistisk uavhengig av hvordan personen presterer på den første oppgaven. På grunn av denne strukturen bør itemene dermed sees som en testlet [14, s173]. Hver enkelt testlet er uavhengig av de andre itemene i testen og kan inkluderes i testskåren på lik linje med individuelle itemer.

Den normale gjennomgangen av en testlet kan sees i figur 1 nedenfor. Et subjekt får itemstimuli og returnerer et response. Subjektet får dermed et nytt stimuli og leverer så et nytt response. Denne prosedyren gjentas helt til alle deloppgavene er besvart. Subjektet vil deretter få en enkelt totalskår for testleten.



Figur 1: Gjennomføring av en testlet

Computer Adaptiv Test (CAT): Computer adaptiv testing er bruk av tester som vet vanskelighetsgraden på itemene i itembanken og som presenterer neste oppgave i testen basert på hvor bra svaret på forrige oppgave var. Rekkefølgen på itemene er ikke fastsatt, men subjektet får nye oppgaver avhengig av hvordan svar (responses) har blitt vurdert i tidligere deler av testen [16, s 15]. Ved å tilpasse seg subjektets ferdighetsnivå og hverken gi oppgaver som er for enkle eller vanskelige kan testen gi mest mulig informasjon om et subjekt innen tiden som er til rådighet. Dette virker også mer motiverende for subjektene, og de vil dermed ha en bedre sjanse til å gjøre sitt beste [14, s 49].

En oppgave kan vurderes enten i sanntid, altså med én gang den blir sendt inn i systemet, eller den kan vurderes i etterkant. Å ha støtte for å kunne vurdere og skårsette oppgaver i etterkant av en test er nødvendig dersom man ønsker å beregne resultatene på nytt etter andre kriterier. Computer adaptiv testing krever vurdering av oppgaver i sanntid fordi neste oppgave i testen er avhengig av hva subjektet har svart på de foregående oppgavene. En mulighet er at man først gir oppgaver som kan automatisk evalueres for å estimere ferdighetsnivået til et subjekt. Man kan

deretter gå over til manuell evaluering av oppgaver med det estimerte vanskelighetsnivået. Et viktig spørsmål er imidlertid om oppgaver som evalueres manuelt og automatisk måler det samme. Dette vil det ikke gås nærmere inn på i denne oppgaven.

For at computer adaptiv testing skal være mulig å gjennomføre er det også nødvendig at oppgavene har en anslått vanskelighetsgrad. Denne vanskelighetsgraden fastsettes ut i fra skårene som er oppnådd av tidligere subjekter for et item. Ved å bruke disse verdiene kan en adaptiv algoritme velge neste optimale item i testen for akkurat dette subjektet. Etterhvert som flere og flere subjekter utfører tester og nye skårer lagres i databasen blir det nødvendig å rekalibrere itembanken. Dette gjøres blant annet for å oppdatere vanskelighetsgraden på oppgavene ut i fra hvilke nye skårer som er oppnådd for et item. Ved å rekalibrere itembanken kan vi derfor hele tiden være sikre på at vi har et oppdatert oppgavesett og optimaliserte tester. Det vil også bli aktuelt å fjerne eller legge til nye oppgaver for testene uten at vi ønsker å slette skårene som er relatert til disse oppgavene. Det er derfor viktig at et rammeverk har støtte for lagring av historiske data.

2.2.6 Klassisk test-teori

Klassisk test-teori er en formell metode for mentale tester. Den ble utviklet på bakgrunn av heuristiske ideer om hvilke krav som stilles til en valid og pålitelig test. Den fundamentale ideen bak denne teorien kalles Concept of reliability [16, s26] og kan uttrykkes i en enkelt ligning, vist i figur 2:

Figur 2: Concept of reliability

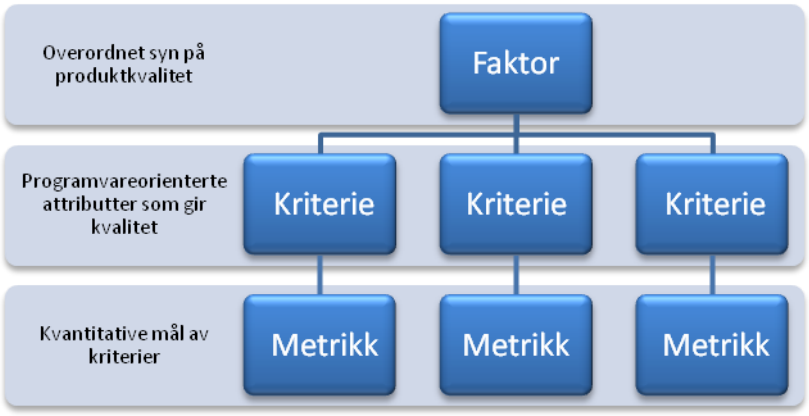
”Ekte skår” vil si hva som er forventet å være den gjennomsnittlige skåren til et subjekt dersom han eller hun tar nøyaktig samme test flere ganger parallelt. Variabelen ”feilmargin” står for forskjellen mellom forventet skår og observert skår [19]. Klassisk test-teori bruker forholdet mellom disse tre variablene for å kunne si noe om hvor pålitelig en test er. Påliteligheten til en test øker i grad med at feilvariabelen synker og omvendt [19].

En test består ofte av flere oppgaver eller deler, der man får en skår på hver ulik del. Disse skårene legges deretter sammen for å utgjøre en totalskår. Denne totalskåren skal dermed kunne være representativ for alle deloppgavene personen har utført [16, s 10]. Det finnes mange problemstillinger relatert til hvordan man bygger opp den totale skåren for en test. I denne oppgaven er fokuset derimot på hvordan hver enkelt oppgaveskår kan bygges opp.

2.3 Måling av kvalitet for programvare

For å kvalitetssikre programvare og måle kvalitet innen programvareutvikling har det blitt utviklet konseptuelle rammeverk. Disse rammeverkene har struktur som beskrevet i figur 3 [20]. På høyeste nivå i figuren har vi kvalitetsfaktorene. Dette er kvalitetsaspekter som brukere eller kunder vil se på som relatert til den overordnede kvaliteten på et produkt. Det neste nivået er

kriterier eller attributter ved programvaren som relaterer seg til disse faktorene. Oppfyllelse av disse kriteriene fører til at de relaterte kvalitetsfaktorene oppnås. På tredje og nederste nivå finner vi metrikker, eller målinger. Metrikkene måler i hvor stor grad de ovenforliggende kriteriene oppfylles [20].



Figur 3: Oppbygning for rammeverk for programvarekvalitet

Figuren ovenfor beskriver hvordan man kan oppnå ønsket kvalitet for programvare, men denne modellen kan også brukes for å se på hvordan rammeverk for automatisk evaluering av programmeringsoppgaver er bygget opp. Vi har abstrakte begreper som beskriver overordnede kvalitetsaspekter vi ønsker å vektlegge (faktorer). Disse begrepene brytes ned i mer presise beskrivelser av hva det vil si at et kodesegment oppfyller de abstrakte kravene (kriterier). For å måle tilstedeværelsen av disse attributtene definerer vi konkrete, målbare krav til koden (metrikker). Eksempel på metrikker kan være hvor mange testcaser som er passert eller hvor lang tid man har brukt på å skrive koden.

2.3.1 Faktorer for måling av kvalitet for programvare

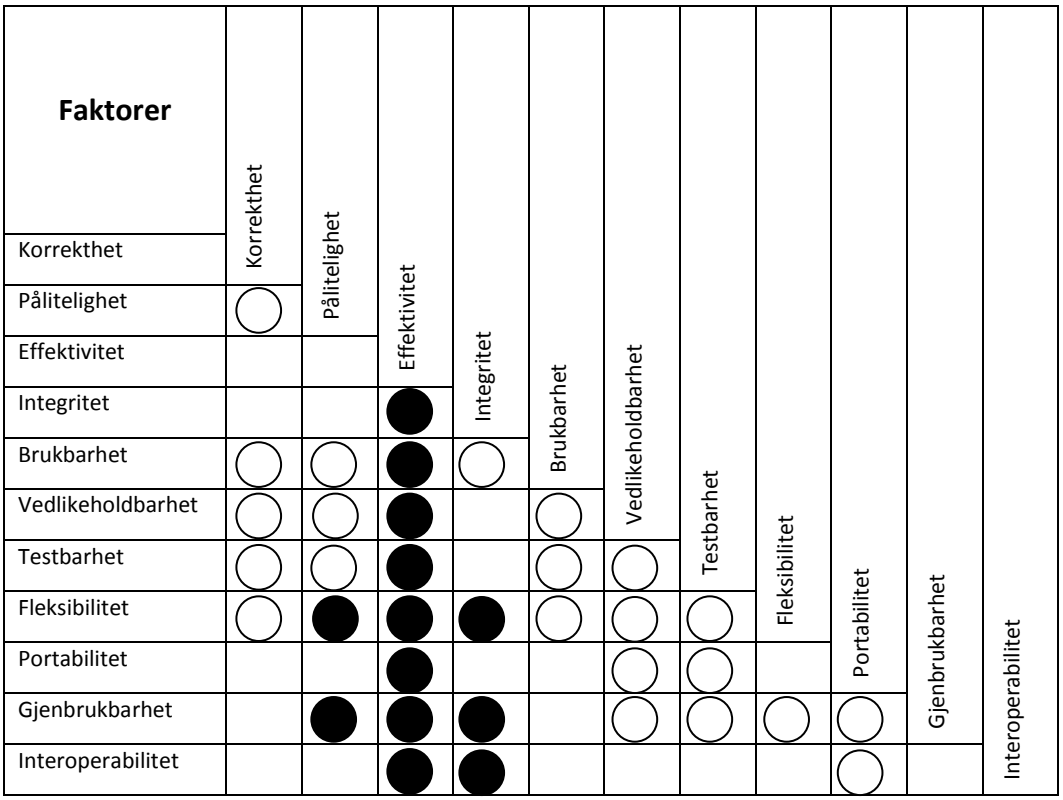
Konseptet rundt kvalitetsfaktorer for programvare oppsto sent på 1970-tallet [20]. Det finnes flere definisjoner av disse faktorene, og de vil også typisk høre til under et eller flere stadier i livssyklusen til et programvareprodukt. McCall [20] definerer elleve kvalitetsfaktorer. Disse er gjengitt i tabell 1.

Tabell 1: Definisjon av kvalitetsfaktorer for programvare

Definisjon av kvalitetsfaktorer for programvare:	
Korrekthet (Correctness)	Hvorvidt et program tilfredsstiller en spesifikasjon og oppfyller en brukers hovedmål.
Pålitelighet (Reliability)	Hvorvidt et program kan forventes å kunne gjøre planlagt funksjonalitet med tilstrekkelig presisjon.
Effektivitet (Efficiency)	Ressurser et program krever for å utføre en funksjon/operasjon.
Integritet (Integrity)	Hvorvidt uautoriserte brukere sin tilgang til programvaren og data kan kontrolleres.
Brukbarhet (Usability)	Innsats som kreves for å lære å bruke, samt forberede input og tolke output fra programvaren.
Vedlikeholdbarhet (Maintainability)	Innsats som kreves for å lokalisere og rette opp en feil i et operasjonelt program.
Testbarhet (Testability)	Innsats som kreves for å forsikre at programvaren presterer etter det som er forventet funksjonalitet.
Fleksibilitet (Flexibility)	Innsats som kreves for å modifisere et operasjonelt program.
Portabilitet (Portability)	Innsats som kreves for å overføre programvare fra en maskinvarekonfigurasjon og/eller et programvaremiljø til et annet.
Gjenbrukbarhet (Reusability)	Hvorvidt programvaren kan brukes med/i andre applikasjoner.
Interoperabilitet (Interoperability)	Innsats som kreves for å koble et system sammen med et annet.

Kvalitetsfaktorene innvirker på hverandre og en oversikt over dette er vist nedenfor, i tabell 2 [20]. Tabellen viser at dersom en faktor er til stede er det ofte sannsynlig at en eller flere andre faktorer vil være enklere å oppnå. Dette er vist med de lyse sirklene. Motsatt kan også tilstedeværelsen av en faktor i noen tilfeller gjøre det mer kostbart eller vanskeligere å oppnå andre faktorer. De mørke sirklene indikerer disse kombinasjonene.

Tabell 2: Forhold mellom kvalitetsfaktorer for programvare



2.3.2 Nivåer for måling

For variabler i matematikk og statistikk er det definert fire nivåer for å gjøre målinger. Disse nivåene er utviklet av Stevens [21] og kan sees i tabell 3 nedenfor. Klassifikasjonen beskriver hva slags informasjon som kan leses ut i fra tall som er tilegnet objekter. Det kan utføres ulike matematiske operasjoner på disse variablene avhengig av hvilket nivå de er målt på [21]:

Tabell 3: Stevens nivåer for målinger

Skala:	**Beskrivelse:**	**Eksempel:**
Nominell	Objekter klassifiseres og klassene skilles med nummer. Størrelsen på tallene sier ingenting om at en av klassene er større eller mindre enn en annen, men angir kun at de er forskjellige. Resultatene kan ikke adderes eller subtraheres, og heller ikke sammenlignes på større eller mindre enn.	Bilnummer, øyefarge, kjønn, personnummer
Ordinal	Objekter tilordnes nummer for å rangere entitetene som måles. Nummeret gjenspeiler mengden som dette objektet innehar. Forskjellen mellom numrene indikerer likevel ikke noe om forskjellen i mengden til attributtene. Man kan sammenligne på større og mindre enn og også måle likhet og ulikhet.	Hardhet for materialer, militære ordener

	Likevel gir det ingen mening å addere eller subtrahere målingene da forholdet mellom variablene ikke er konstant.	
Intervall	På denne skalaen kan objektene ordnes i rekkefølge, og de kan også tilegnes nummer som beskriver forskjellene mellom verdiene de representerer. Nullpunktet på skalaen er vilkårlig, og man kan derfor også bruke negative verdier. Målingene kan adderes og subtraheres, men man kan ikke utføre divisjon og multiplikasjon.	Kalendertid, Fahrenheit og Celsius temperaturskala
Ratio	Denne skalaen har alle egenskapene til intervall skala, men i tillegg er absolutt nullpunkt definert. Alle matematiske operasjoner kan derfor utføres på verdiene.	Høyde, vekt, tid, effektivitet for et program eller en programvareutvikler.

2.4 Eksisterende eksperimenter

For å gjennomføre en test for å undersøke dyktighet og ferdigheter blant programvareutviklere er det spesielt ett eksperiment som ofte brukes. Dette eksperimentet innebærer å designe og utvikle en kaffemaskin, og kalles *The Coffee Machine Design Problem*. Det initielle problemet blir beskrevet slik:

You and I are contractors who just won a bid to design a custom vending machine for the employees of Acme Fijet Works to use. Arnold, the owner of Acme Fijet Works, like the common software designer, eschews standard solutions. He wants his own, custom design. He is, however, a cheapskate. Arnold tells us he wants a simple machine. All he wants is a machine that serves coffee for 35 cents, with or without sugar and creamer. That's all. He expects us to be able to put this little machine together quickly and for little cost. We get together and decide there will be a coin slot and coin return, coin return button, and four other buttons: black, white, black with sugar, and white with sugar [22].

Dette problemet har blitt brukt i eksperimenter for å kartlegge hvordan programvareutviklere med varierende erfaring velger å designe objektorienterte systemer. Eksperimentet tar også for seg hvor mye tid utviklerne bruker for å sette seg inn i og vedlikeholde systemer. Design av ansvar og samarbeid mellom klassene i et system kan gjøres på flere måter, og to av de vanligste fremgangsmåtene for dette er delegert kontrollstil og sentralisert kontrollstil. Delegert kontroll vil si at ansvar er fordelt utover flere klasser i applikasjonen. Sentralisert kontroll innebærer at applikasjonen har noen få, store kontrollklasser som koordinerer et sett med relativt enkle klasser [22]. Forsøk viser at de mest erfarne utviklerne bruker mye kortere tid på å vedlikeholde en applikasjon med delegert kontrollstil enn en applikasjon med sentralisert kontrollstil. Samtidig ser man at mer uerfarne utviklere har store problemer med å forstå hvordan applikasjoner med delegert kontrollstil er bygget opp, og de presterer mye bedre dersom de kan forholde seg til en sentralisert kontrollstil. Dette kan dermed føre til at erfarne utviklere designer applikasjoner og systemer som mindre erfarne utviklere ikke klarer å vedlikeholde [22].

Resultatene fra eksperimentet til [22] underbygger en påstand om at uerfarne programvareutviklere som gruppe har problemer med å forstå delegert kontrollstil i applikasjoner. Dette trenger ikke nødvendigvis å være et problem som gjelder for absolutt alle enkeltindividene som faller innenfor denne gruppen, men det er en generell slutning som kan trekkes ut i fra resultatene. I et eksperiment som dette er det derimot vanskelig å trekke ut informasjon om en enkelt programvareutvikler. Eksperimenter som er rettet mot grupper er sjelden godt egnet når vi vil ha informasjon om individer. For å si noe om en enkelt programvareutvikler sine prestasjoner må vi derfor ta i bruk andre virkemidler.

Dette kontrollerte eksperimentet har blitt brukt i forskning. Sett fra et forskningsmessig perspektiv er det ofte mest interessant å kunne trekke slutninger og se sammenhenger mellom store mengder data, og på denne måten kunne si noe om helheten. Motsatt vil det både i industrien og ved utdanningsinstitusjoner typisk være mer interessant å se på hver enkelt observasjon for å kunne si noe om hvert enkelt individ, enten for å fastsette en karakter eller for å kartlegge arbeidsmetoder og tidsbruk. Til tross for at prototypeimplementasjonen som skal utvikles i denne oppgaven skal brukes innenfor forskning er det likevel individene og deres respektive skårer vi ønsker å sette fokus på. Konklusjonen blir derfor at et eksperiment som det som er beskrevet her gjerne kan brukes som en testlet også i vårt rammeverk, men at vi da må bestemme oss for om det er den sentraliserte eller delegerte kontrollstilen vi vil skal være den foretrukne fremgangsmåten for implementasjonen og som følgelig skal gi mest poeng. Resultatet av vurderingen vil da være sterkt avhengig av hvilken kontrollstil vi har valgt å gi best skår for. Det kunne derfor i denne situasjonen ha vært interessant å implementere to ulike verdisystemer for å skårsette disse deloppgavene, ett som vektlegger sentralisert kontrollstil og ett som vektlegger delegert kontrollstil. Dette ville ha kunnet gitt store variasjoner i skårene til hver enkelt person.

2.5 Eksisterende rammeverk

Den tidligste referansen jeg har funnet til rammeverk for bruk av automatisk evaluering av programmeringsoppgaver er fra 1998 [23]. Her beskrives et online rammeverk som har blitt benyttet av et universitet i Ontario, Canada. Artikkelforfatterne beskriver hvordan de har brukt denne metoden for å evaluere programmeringsoppgaver i Java de tre siste årene, altså helt tilbake til 1995. Ideen bak denne typen rammeverk ble imidlertid første gang presentert av Hollingsworth i 1960 [24].

I denne seksjonen vil jeg først presentere rammeverkene jeg har sett nærmere på og beskrive hva og hvem de brukes til og hva de måler. Videre vil jeg forklare forskjellen på statisk og dynamisk evaluering som alle rammeverkene benytter seg av. Jeg vil deretter klassifisere målingsteknikkene som er brukt, og videre følger en oversikt over hvordan rammeverkene har valgt å operasjonalisere kvalitetsfaktorene sine. Jeg gir også en grundigere oversikt over hvilke anvendelsesområder slike typer rammeverk har. Til slutt snakker jeg noe om hva jeg mener er svakheter ved disse rammeverkene og ser på hva som kan gjøres bedre.

For å få en oversikt over hvilke typer rammeverk som eksisterer og hva slags funksjonalitet det finnes støtte for har jeg tatt for meg et utvalg. Rammeverkene jeg har sett på vises i tabell 4 og kartlegger de viktigste opplysningene om de forskjellige rammeverkene med respektive

kildehenvisninger. De ulike rammeverkene brukes i ulike situasjoner og er beregnet for ulike typer mennesker. Ut i fra disse faktorene er det også forskjellig hva som måles for å vurdere programmeringsoppgavene.

Tabell 4: Rammeverk for automatisk evaluering

Rammeverk	**Brukes til:**	**Måler:**	**Anvendelses-område:**
Online Judge [5]	Programmeringskonkurranser.	Korrekthet, validitet, robusthet og effektivitet.	Studenter
Web-CAT [25]	Programmeringskurs. Setter karakter på både kode og tester.	Kompletthet og korrekthet, samt stil og kvalitet på koden	Studenter
JKarelRobot [8]	Programmeringskurs. Mye fokus på pedagogikk og gradvis tilegning av kunnskap.	Korrekthet.	Uerfarne studenter
Watcher [9]	Overvåker og dokumenterer programmererens oppførsel. Registrerer bevegelse av musen, tastetrykk, scrolling og endringer i dokumentet.	Oppførsel og programmeringsteknikk hos utvikleren.	Industri og forskning
ELP – Environment for Learning to Program [24]	Programmeringskurs.	Korrekthet og programmeringspraksis. Bruker statisk og dynamisk analyse, strukturell likhet og subjektiv vurdering.	Studenter
SESE – Simula Experiment Support Environment [26]	Nøye kontrollerte eksperimenter som skal være så realistiske som mulig.	Real-time monitorering.	Industri og forskning
Topcoder [7]	Programmeringskurs. Legger vekt på algoritmer og problemløsning.	Korrekthet og tidsbruk.	Studenter
Lab-practicum [27]	Programmeringskurs. Studentene utvikler og tester et komplett program innenfor en makstid.	Korrekthet og subjektiv vurdering.	Studenter
HoGG – Homework Generation Grading [6]	Programmeringskurs. Bruker enhetstesting, regulære uttrykk og Java Reflection Classes.	Korrekthet. Subjektiv vurdering dersom studenter klager på resultatet.	Studenter

Fullstendig oversikt over disse rammeverkene finnes som vedlegg (*Vedlegg 1: Kartlegging av utvalgte rammeverk for automatisk evaluering av programmeringsoppgaver*).

Ikke alle rammeverkene gir en egen skår som skal si noe om dyktigheten til subjektene, men felles for alle er at de har bestemte kriterier å vurdere etter. Noen rammeverk bedømmer kun om et svar er godkjent eller ikke godkjent. Her er ikke det ikke lagt vekt på det å kunne si noe om nivå av dyktighet hos en enkeltperson, men fokuset er heller på læring og utvikling.

2.5.1 Statisk og dynamisk evaluering

For automatisk å evaluere programmeringskode bruker rammeverkene enten statisk eller dynamisk evaluering, eller begge samtidig. Statisk evaluering er en prosess der man analyserer kildekode uten å eksekvere selve programmet. Dynamisk evaluering er en prosess der man kjører et program gjennom et sett med data. Førstnevnte brukes derfor til å finne problemer i koden som potensielle feil, unødvendig kompleksitet og områder som krever mye vedlikehold. Statisk evaluering kan igjen deles inn i to hovedområder: strukturell analyse (design av et program) og semantisk analyse (brukes til optimalisering og verifikasjon av programmer). Dynamisk evaluering brukes til å oppdage kjørefeil ved et program og å evaluere korrektheten av et program [24].

2.5.2 Klassifisering av teknikker for måling

Tabell 5 viser en klassifisering av de viktigste evalueringsteknikkene som er brukt i rammeverkene jeg har sett på. Teknikkene er klassifisert etter om det benyttes statisk eller dynamisk evaluering, samt hvilket av Stevens nivåer for måling (som er beskrevet i seksjon 2.3.2) som brukes.

Tabell 5: Klassifisering av ulike evalueringskriterier funnet i eksisterende rammeverk

Hva som måles	**Beskrivelse**	**Evalueringsmetode**	**Skala**
Korrekthet (samt robusthet)	Koden skal gi korrekt output for all mulig lovlig input som brukes.	Dynamisk	Ordinal
Effektivitet	Måles ofte som ressursbruk, som for eksempel hvor mye prosessorkraft og minne som kreves av programmet.	Dynamisk	Ratio
Vedlikeholdbarhet	Selvforklarende variabelnavn, kommentarer, innrykk, kompleksitet og øvrig kodestandard.	Statisk	Ordinal
Tidsbruk	Hvor mye tid som faktisk brukes på en oppgave.	Statisk	Ratio
Programmererens effektivitet	Hvor mye tid som brukes på ulike deler av programmeringsprosessen (scrolling, browsing, tastetrykk, museklikk).	Statisk	Ratio
Strukturell likhet	Kode transformeres til pseudokode og	Statisk	Ordinal

	sammenlignes med en modell.		
Validitet	Koden testes mot et gitt sett input-verdier.	Dynamisk	Ordinal

2.5.3 Operasjonalisering av kvalitetsfaktorer

Begreper som beskriver kvaliteten på programvare kan tolkes på forskjellige måter og ha flere betydninger. I denne seksjonen sees det derfor på hvordan de ulike rammeverkene operasjonaliserer kvalitetsfaktorene de har valgt å vektlegge. Denne oversikten kan sees i tabell 6. Kvalitetsfaktorene kan sees på som å være verdisystemet som rammeverket vurderer etter.

Tabell 6: Evalueringsmetoder og operasjonalisering

Kvalitetsfaktorer:	**Operasjonalisert ved:**	**Støttes av:**
Funksjonell korrekthet	JUnit	Topcoder, Web-CAT
	ANTLR og Abstract Syntax Trees representert vha XML (Strukturell likhet)	ELP
	Enhetstesting, regulære uttrykk og Java Reflection Classes	HoGG Project
	Subjektiv vurdering	HoGG Project, Lab Practicum
Kompletthet for tester	Clover	Web-CAT
Programmeringspraksis	Eclipse plug-in	Watcher
	Statisk analyse	ELP
Tidsbruk		Topcoder
Kodestil	Checkstyle	Web-CAT
Optimalitet på koden	PMD	Web-CAT

Som vi ser av tabellen ovenfor er det stor spredning av hvilke metoder og verktøy som brukes for å operasjonalisere de samme kvalitetsfaktorene. To rammeverk som hevder å vurdere etter samme faktorer kan derfor resultere i å gi ulike skårer for nøyaktig samme prestasjon.

2.5.4 Anvendelsesområder for rammeverkene

Som vi ser av tabellen ovenfor er det stor spredning av hvilke metoder og verktøy som brukes for å operasjonalisere de samme kvalitetsfaktorene. To rammeverk som hevder å vurdere etter samme faktorer kan derfor resultere i å gi ulike skårer for nøyaktig samme prestasjon.

2.5.5 Anvendelsesområder for rammeverkene

Som vi så av tabell 4 er det mulig å skille mellom de ulike rammeverkene ved å se på hvilke områdene de brukes innenfor. Disse områdene kan i hovedsak deles inn i to kategorier:

- Utdanningsinstitusjoner (inkludert programmeringskonkurranser)
- Industri og forskning

Rammeverkene som benyttes innefor de forskjellige anvendelsesområdene vil ha ulike brukere, og det vil være ulike hensikter som ligger til grunn for bruken av rammeverket. Det finnes også et skille mellom hvilke aspekter ved programvareutviklingen rammeverkene velger å legge vekt på. For å kunne si noe om dyktigheten til en person er det ikke bare viktig å se på hva som presteres, men også på prosessen. Hvilke ytre faktorer som er til stede under utviklingen kan også ha innvirkning på helheten. For å utføre målinger innenfor programvareutvikling kan vi derfor se på tre ulike faktorer [10, s 29]:

- Prosess
- Produkt
- Ressurser

Prosess vil i denne sammenheng si den tiden og de aktivitetene som trengs for at en utvikler skal kunne løse en oppgave. Gjennom denne prosessen produserer programmereren et ferdig produkt i form av et kodesegment eller et program. Denne koden har ulike egenskaper i form av kvalitet og funksjoner som kan testes og evalueres. For å komme frem til et produkt trenger utvikleren også ressurser, for eksempel i form av personell, maskinvare eller programvare. Tabell 7 viser rammeverkenes fordeling mellom bruksområder sett i sammenheng med hvilke aspekter ved programvareutviklingen det legges vekt på for å kartlegge programmeringsferdigheter.

Tabell 7: Sammenhenger mellom bruksområder for rammeverk og hva som måles

Navn på rammeverk	**Bruksområde:**		**Hva som måles:**		
	Utdanning	**Industri og forskning**	**Produkt**	**Prosess**	**Ressurser**
Online Judge	X		X		
Web-CAT	X		X		
JKarelRobot	X		X		
Watcher		X		X	
ELP	X		X		
Topcoder	X		X	X	
Lab practicum	X		X		
HoGG	X		X		

Med ett unntak har jeg sett at rammeverk som brukes av utdanningsinstitusjoner og studenter i stor grad måler og evaluerer etter faktorer som funksjonell korrekthet, effektivitet og lignende. Dette er aspekter og egenskaper ved det ferdige produktet. To av rammeverkene måler faktorer

relatert til prosess. Ingen av rammeverkene måler ressurser, som for eksempel eventuelle hjelpemidler som er lov å bruke under testen.

Rammeverkene som er myntet på studenter er opptatt av korrekthet og robusthet, samt aspekter som minne- og prosessorbruk. Hovedmålet med vurderingene er at studentene skal få rask og effektiv tilbakemelding, noe som sparer utdanningsinstitusjonen for både tid og penger. Det er også viktig for skolene at studentene lærer seg å utvikle effektiv kode som ikke tar for mye ressurser på maskinen som skal kjøre og teste koden. Hensikten med rammeverkene er at studentene selv skal kunne forbedre løsningene sine for å bli bedre programmerere. Fokuset ligger altså på å evaluere det ferdige produktet (koden) for at studenten skal utvikle seg og bli bedre for sin egen del. De aller fleste rammeverkene som er utviklet for studenter benytter formativ vurdering. Formativ vurdering vil si at en person kan levere flere løsninger for samme oppgave dersom han/hun ikke er fornøyd med sitt første bidrag. Dette er en vurderingsform som er designet for å forbedre kunnskaper og ferdigheter [28].

I de industrielle og forskningsrelaterte rammeverkene er man på den annen side mer opptatt av å analysere ressursbruk i programmeringsprosessen, og hovedfokuset ligger på hvor mye tid som brukes på de ulike aktivitetene. Det er viktig å kartlegge hva som kjennetegner god programmeringspraksis for at de som er gode skal kunne dele arbeidsteknikker med de som er mindre gode. Det måles derfor hva en programmerer bruker mest og minst tid på når han eller hun løser en oppgave. [26] ser også på muligheten til at programmererne skal kunne gi tilbakemelding underveis angående egne tankemønstre og arbeidsprosesser. Gitt det store fokuset på tidsbesparelse og kartlegging av arbeidsmåter for å estimere tidsbruk kan det virke som om det tas for gitt at utviklerne allerede produserer korrekt og effektiv kode. Her legges det mest vekt på at de ansatte skal kunne lære av hverandres positive og negative arbeidsmetoder for at bedriften totalt sett skal kunne bli bedre. Rammeverkene som brukes i forskning og industri bruker hovedsaklig summativ vurdering. Summativ vurdering betyr at utvikleren kun får ett forsøk for hver oppgave som gis i testen. Dette er en vurderingsform som er designet for å vurdere ferdighetsnivået til utviklere [28].

2.5.6 Forbedringsområder ved eksisterende rammeverk

Som beskrevet i introduksjonen til denne oppgaven er rammeverk for automatisk evaluering av programmeringsferdigheter mye brukt og kan være et effektivt virkemiddel både ved utdanningsinstitusjoner, i industrien og i forskning. Men det er viktig å tenke nøye igjennom hva som ligger til grunn for skårene vi får fra slik programvare. Det er visse aspekter som ikke alltid tas hensyn til som kan innvirke på hvilke subjekter som oppfattes som gode og hvilke som oppfattes som mindre gode. Som de viktigste forbedringsområdene for eksisterende rammeverk vil jeg derfor trekke frem:

Manglende skille mellom evaluering og skår: Ved at det ikke gjøres et klart skille mellom evalueringer og skårer er det vanskelig å se hvilke egenskaper ved koden som vektlegges i en vurdering. Et subjekt som vurderes som dyktig av ett rammeverk er derfor ikke garantert å få samme resultat fra et annet rammeverk. Det legges ikke fokus på at det finnes flere måter å beregne en skår på, og at resultatene er sterkt avhengig av hvilken strategi som velges. I stedet for å måle dyktigheten til et subjekt måles det derfor heller hvor dyktig en person er til å programmere i henhold til de kravene som vektlegges i vurderingen.

Vurdering av produkt og/eller prosess: Et annet aspekt ved disse rammeverkene er at det ikke alltid legges vekt på at det kan være hensiktsmessig å gjøre en vurdering av både prosess og produkt for å se på ferdighetene til et subjekt. Å vurdere produkt eller prosess er også veldig knyttet til anvendelsesområde for et rammeverk. Det sees heller ikke på hvordan ressurser kan ha innvirkning på testresultatene.

Kapittel 4 gir en oversikt over hvilke krav jeg mener bør stilles til et rammeverk for automatisk evaluering av programmeringsferdigheter for å ta tak i noen av disse områdene.

2.6 Persistens i applikasjoner

De aller fleste applikasjoner trenger å lagre data som kan hentes frem igjen neste gang man bruker applikasjonen. De lagrede dataene kalles persistente data og er essensielle for at vi skal kunne analysere opplysninger. Det mest vanlige er å lagre data i en relasjonsdatabase, men det finnes i tillegg objektdatabaser, samt kombinasjoner og varianter av disse [29, s 5]. I denne seksjonen vil jeg først redegjøre for hva en relasjonsdatabase er, og deretter hvilke utfordringer som dukker opp når vi skal integrere en database i en objektorientert applikasjon. Videre gis det en forklaring på hva objekt/relasjonsmapping er. Det gis også en innføring i hva forskjellen på persistente og transiente objekter er, noe som er viktig å forstå deler av resultatene og analysen i denne oppgaven.

2.6.1 Relasjonsdatabaser

Den mest vanlige typen database er relasjonsdatabasen som hadde sin opprinnelse på begynnelsen av 1990-tallet, samtidig som objektorientert programmering oppsto. Modellen bak, relasjonsmodellen, kan derimot dateres tilbake til 1969 og ble først foreslått av Edgar Codd [30]. En relasjonsdatabase bruker en serie med tabeller for å organisere data. Dataene i databasen kan assosieres med hverandre ved å sette opp relasjoner mellom de ulike tabellene. Data som kan lagres i ett eneste objekt vil ofte måtte lagres på kryss av flere av disse tabellene [31]. Relasjonsdatabaser er en fleksibel og robust måte å håndtere data på. Som følge av det komplette og konsistente teoretiske grunnlaget de er bygget på er de med på å garantere høy integritet og levetid på dataene [29, s 6].

2.6.2 Persistens i objektorienterte applikasjoner

Innenfor persistens har vi to hovedparadigmer: relasjonsmodellen og objektmodellen. Når vi snakker om persistens i objektorienterte applikasjoner dukker problemet med forskjellen mellom disse to paradigmene opp. Relasjonsmodellen gir oss en strukturert representasjon av dataene og lar oss sortere, aggregere, manipulere og søke i informasjonen. En objektorientert applikasjon har sin egen domenemodell for disse dataene, og forretningslogikken gjør operasjoner på objektene som er definert i denne modellen fremfor å manipulere de enkelte dataene som leveres fra relasjonsdatabasen. Dataene fra en relasjonsdatabase må derfor manuelt konverteres til objekter før forretningslogikken i applikasjonen kan nyttegjøre seg av objektorienterte konsepter som arv og polymorfisme [29, s 6]. På samme måte må objektene konverteres til enklere verdier før dataene kan lagres i relasjonsdatabasen. Gapet mellom disse to modellene er roten til mange ulike problemer som granularitetsproblemet, subtypeproblemet, identietsproblemet og problemer angående relasjoner mellom entiteter og

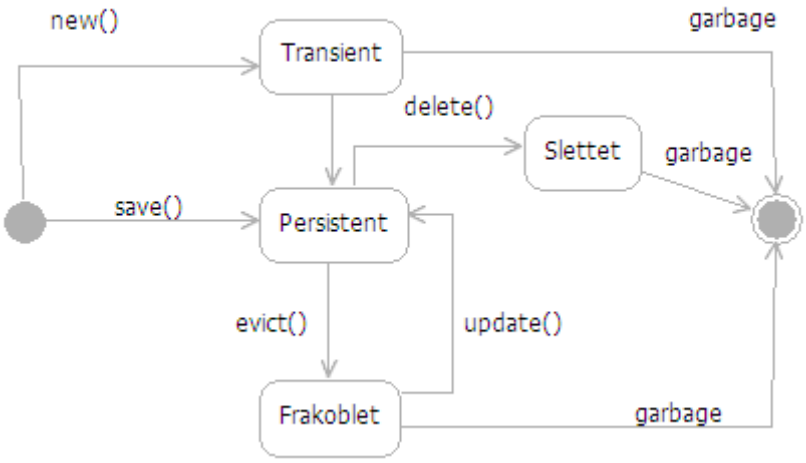
datanavigasjon [29, s 10-19]. Jeg vil ikke gå nærmere inn på disse her, men det å finne løsninger på disse problemene kan være ekstremt tidkrevende [29, s 19]. Et forslag for å løse problemene har vært å utvikle rene objektdatabaser. Disse har av ulike grunner ikke vunnet store andeler i det tradisjonelle databasemarkedet. Det er derfor blitt utviklet rammeverk som skal håndtere gapet mellom relasjons- og objektmodellen slik at den enkelte programmerer ikke skal trenge å bruke unødvendig tid og krefter på dette.

2.6.3 Objekt- Relasjonsmapping

Teknikken for å brolegge gapet mellom objekt- og relasjonsmodellen kalles objekt-relasjonsmapping (O/R-mapping) og skal tilby et sømløst grensesnitt mellom applikasjonen og databasen gjennom en virtuell objektdatabase. Hensikten med O/R-mapping er å spare tid, forenkle utvikling, øke ytelse og skalerbarhet og minimere utfordringer relatert til arkitektur [31]. Det finnes mange ulike verktøy for O/R-mapping, både fra kommersielle aktører og også som åpen kildekode. Eksempler er: Hibernate, Enterprise Objects Framework (EOF), Apache Cayenne, Java Data Objects (JDO), Enterprise Java Beans 3.0 (EJB 3.0). Hibernate er det mest brukte rammeverket innenfor Javamiljøet [31].

2.6.4 Persistentskontekst og livssyklus for persistente objekter

Hibernate er et verktøy for O/R-mapping som bruker transparent persistens. Transparent persistens vil si at applikasjonen er totalt uvitende om objektene den arbeider med representerer en persistent tilstand eller om de bare finnes i minnet [29, s 384]. For at dette skal kunne fungerer i praksis er applikasjonen nødt til å ta hensyn til tilstanden og livssyklusen til objektene. Figur 4 beskriver denne livssyklusen [29, s 386]:



Figur 4: Livssyklus for persistente objekter

De fleste Java-applikasjoner inneholder en blanding av persistente og transiente objekter. Et persistent objekt vil si at tilstanden til objektet er lagret et sted og kan hentes opp igjen senere for å gjenskapes i den samme tilstanden. Et transient objekt derimot har en levetid begrenset til levetiden til den prosessen som opprettet objektet. Objektet er instansiert, men assosieres ikke med en tabellrad i en database. Det vil dermed gå tapt dersom det ikke refereres til fra et annet objekt [29, s 386]. Tilstanden ” frakoblet” betyr at objektet er lagret i databasen, men at

koblingen til databasen er lukket. Vi har derfor en referanse til objektet, men endringer vi gjør lagres ikke før vi oppretter en ny kobling til databasen og oppdaterer databaseradene.

En kobling til databasen kan sees på som det vi kaller persistenskontekst. Persistenskonteksten er et mellomlager av entitetsinstanser. I en applikasjon som bruker Hibernate har en databasesesjon en intern persistenskontekst [29, s 388]. Ved å jobbe med objekter innenfor denne persistenskonteksten kan man for eksempel slippe å tenke på identitetsproblemet (nevnt i seksjon 2.6.2) som oppstår når vi kombinerer objekt- og relasjonsmodellen. Man kan også slippe å tenke på å lagre et objekt etter at man har gjort endringer på det fordi persistenskonteksten automatisk sjekker om noen av verdiene er endret. Når databasesesjonen termineres avsluttes også persistenskonteksten. Objektet er dermed ikke et persistent objekt lenger, men et frakoblet objekt uten kobling til databasen. Både transiente og frakoblede objekter går tapt når prosessen de er opprettet innefor terminerer og objektene fjernes da av Garbage Collector i Java. Det samme skjer med persistente objekter som manuelt slettes fra databasen.

3 Utviklingsprosess

Arbeidet med denne oppgaven kan deles inn i seks deler:

- Kartlegging av eksisterende litteratur og arbeid innenfor automatisk evaluering av programmeringsferdigheter
- Stabilisering av eksisterende funksjonalitet i JCAT
- Definisjon av spesifikke krav til hvordan rammeverket bør håndtere aspekter som evalueringer, skårer og verdisystemer, samt funksjonelle- og ikke-funksjonelle krav til prototypeimplementasjonen
- Implementering av støtte for overnevnt krav i JCAT
- Implementering av støtte for persistente data i JCAT
- Evaluering av eget arbeid

Arbeidet med de ulike delene har i hovedsak foregått i den overnevnte rekkefølgen, men noen oppgaver har blitt utført i parallell.

Under arbeidet med oppgaven har jeg vært en del av en prosjektgruppe på tre personer. For at flere personer skulle kunne jobbe samtidig med samme kode har det vært behov for et verktøy for versjonskontroll. Vi har valgt å bruke Tortoise SVN [32], som er et åpen kildekode verktøy. Tortoise SVN er en Subversion klient, og er implementert som en utvidelse til Windows.

Ved implementering av ny funksjonalitet har vi forsøkt å følge prinsipper for testdrevet utvikling og har skrevet noen av testene før vi har skrevet selve koden. Arbeidet har blitt planlagt etter en prosjektplan med milepæler, og jeg vil beskrive to viktige deler ved arbeidsprosessen, SCRUM og SMART.

3.1 SCRUM

I utviklingsprosessen har vi jobbet iterativt etter smidige metoder for dokumentasjon og har brukt flere elementer fra SCRUM-metodikken [33]. Vi har hatt relativt korte iterasjoner, sprints, som har blitt avsluttet med en milepæl. I begynnelsen av hver sprint har vi hatt et oppstartsmøte for å planlegge arbeidsoppgaver for den aktuelle milepælen. Vi har også hatt et oppsummeringsmøte etter hver sprint der vi har diskutert fremgangen til prosjektet, samt positive og negative erfaringer. Det har blitt satt konkrete mål for hver milepæl. I tillegg har vi hatt møter flere ganger ukentlig for å informere hverandre om status på arbeidsoppgaver og for å komme med innspill på hverandres arbeid. Ved avslutningen av hver sprint har det blitt ferdigstilt nye versjoner av kode og dokumenter.

Prosjektet ved Simula Research Laboratory som denne oppgaven er en del av består totalt av seks sprints. Levering av prosjektet er i april 2008, og det er en del krav som må støttes i henhold til dette. Grunnlaget for hver sprint er derfor basert på det overordnede kravdokumentet for prosjektet.

3.2 SMART

I utviklingen har vi i tillegg brukt noe som kalles SMART-mål [34]. Disse målene er en måte å beskrive hva det er vi ønsker å oppnå ved å utføre en oppgave og hvordan vi ønsker å utføre den. SMART er et akronym og står for:

- Specific: Hva som skal gjøres og hvorfor og hvordan oppgaven skal utføres

- Measurable: Hvordan det skal måles at man har oppnådd det man skal gjøre
- Attainable: Om det er mulig å oppnå målene som er satt
- Realistic: Om oppgaven er gjennomførbar
- Timely: Oppgaven må ha en tidsfrist

Ved å sette seg ned før man begynner på en oppgave og beskrive hvilke mål man har kan gjøre at man blir mer bevisst på de valgene man tar underveis i arbeidsprosessen. Det kan være vanskelig å vite på forhånd hvor lang tid en oppgave vil ta, men vi har forsøkt å holde tidsfristen og heller være fleksible på kvalitet.

4 Kravspesifikasjon og kravanalyse

Før det gås nærmere inn på utviklingen av prototypen er det nødvendig å beskrive hva som bør tilføres et rammeverk for automatisk evaluering for å kunne gjøre mer presise målinger av dyktighet. Det spesifiseres derfor her et sett med krav spesifikt til evaluering og skåring i et rammeverk. Videre defineres det også funksjonelle og ikke-funksjonelle krav til prototypeimplementasjonen.

4.1 Utgangspunkt for oppgaven: JCAT

Java programming Computerized Assessment Test (JCAT) er programvare for automatisk og manuell evaluering og skåring av programmeringsoppgaver tiltenkt anvendelser innen måling av ferdigheter. Rammeverket har funksjonalitet for å ta inn et svar på en oppgave, evaluere løsningen, og skal på grunnlag av flere delskårer kunne gi et subjekt en skår for en oppgave. JCAT bruker summativ vurdering. Det som skiller JCAT fra øvrige rammeverk for automatisk evaluering er at det gjøres et klart skille mellom evalueringer og skårer. Evalueringene er verdinøytrale observasjoner og sier ingenting om lav eller høy ferdighet. De tillegges ikke verdi før de oversettes til delskårer. Oppgavene som skal evalueres leses inn fra fil. Skårene som genereres er ikke persistente, men går tapt når applikasjonen terminerer. Til å gjøre evalueringer av koden bruker systemet FIT og FitNesse, og det er planlagt støtte for evalueringer med JUnit. Disse tre testrammeverkene brukes også i selve utviklingsprosessen.

Ved oppgavens oppstart var funksjonaliteten som er beskrevet over allerede på plass. Likevel var abstraksjonene rundt skåring og evaluering ikke klare nok, og endel av begrepene var sammenfiltret i systemet. Skåringsreglene var også hardkodet i systemet, slik at skårene ble regnet ut på én bestemt form. Disse reglene var satt opp slik at den eneste måten å regne ut sumskåren for en oppgave på var ved å summere alle delskårene. Rammeverket hadde heller ikke støtte for å kunne gjennomføre tester eller eksperimenter, men kun funsjonalitet for å skåre enkeltoppgaver.

Dokumentasjonen av systemet var mangelfull, og det var tidkrevende å sette seg inn i hvordan de ulike delene av systemet innvirket på hverandre. Det var også en del viktige aspekter som manglet, som for eksempel feilhåndtering, logging og brukergrensesnitt. Rammeverket kunne ikke håndtere store mengder oppgaver og svar på grunn av at dataene ble lagret et filsystem, og det kunne derfor vanskelig skaleres. En annen viktig mangel var funksjonalitet for å lagre skårer. Det var derfor ikke mulig å sammenligne resultater fra ulike subjekter eller oppgaver.

4.1.1 Junit, FIT og FitNesse

JUnit er opprinnelig et rammeverk for enhetstesting av Javakode, og har vært viktig for utbredelsen av testdrevet utvikling [35].

FIT står for Framework for Integrated Test og er et verktøy for å produsere tester. Rammeverket er designet for å støtte akseptansetesting fremfor enhetstesting og er enkelt å bruke, også for ikke-programmerere. FIT skal kunne fungere som en bro mellom programmerere og kunder/brukere og skal gi større innsikt i hvordan programvare virkelig fungerer. FitNesse er en webserver, en wiki og et verktøy for testing av programvare og er bygget på FIT. Brukeren spesifiserer et sett med innverdier, og testene genereres automatisk [36]. I vårt prosjekt bruker vi FitNesse for å evaluere oppgavene som kommer inn i systemet, men også for å teste vår egen kode.

4.2 Spesifikke krav til evaluering og skåring i JCAT

Denne oppgaven søker primært å gi et forslag (i form av en prototype) til hvordan evaluering og skåring skal håndteres i JCAT. I forbindelse med dette er det en del nye prinsipper som bør innføres. Målet er å forsøke å kombinere aspekter fra andre rammeverk med den ekstra funksjonaliteten som bør legges til:

- Det bør eksistere et klart og konsist skille mellom abstraksjonene evaluering og skåring
- Utregningen av skåren bør baseres på et verdisystem
- Verdisystemet bør kunne skiftes ut for å generere nye skårer basert på tidligere evalueringene

Skille mellom abstraksjonene evaluering og skåring: Først og fremst bør det eksistere et klart og konsist skille mellom konseptene evaluering og skår. Evalueringene bør være verdinøytrale observasjoner og bør kunne brukes på forskjellige måter for å generere skårer. Skårene er verdivurderinger som representerer hvordan et subjekt har prestert innenfor det som er formålet med testen.

Utregning av skåren bør baseres på et verdisystem: Det bør brukes et verdisystem for å vektlegge de ulike observasjonene for å regne ut en sumskår. Det finnes dermed et sterkt avhengighetsforhold mellom sumskåren og verdisystemet. Evalueringene er uvitende om dette avhengighetsforholdet. En skår blir unik ut i fra verdisystemet som er lagt til grunn.

Verdisystemet bør kunne skiftes ut for å generere nye skårer basert på tidligere evalueringer: Når vi skårer en oppgave eller gjennomfører en test i sanntid er det hovedsaklig selve skårene vi er ute etter å gjøre persistente. Men et rammeverk for automatisk evaluering av programmeringsoppgaver bør også tilby funksjonalitet for å kunne se på skårene i et historisk perspektiv for å kunne sammenligne de ulike verdisystemene. For å kunne søke i og gjøre nye analyser av tidligere resultatene for ulike subjekter og itemer er det nødvendig å lagre endel tilleggsinformasjon rundt hver enkelt skår:

- **Oppgave:** Beregningen av en skår er først og fremst avhengig av at et subjekt har gitt et svar på en oppgave. Som nevnt i seksjon 2.2 kan et svar fra et subjekt kan være bygget opp på forskjellige måter, enten som fast format eller fritt format. Vi må derfor vite hvilket format det aktuelle svaret er forventet å ha.
- **Karaktersetter:** En oppgave har en eller flere karaktersettere. Karaktersetteren kan enten være en person, eller den kan representere en automatisk skårregel. Begge de ulike typene karaktersetter har sitt eget verdisystem som innvirker på skåren. Når en ny oppgave legges til i systemet må det angis hva slags karaktersetter som skal brukes som standard.
- **Verdisystem:** En menneskelig karaktersetter velger selv (bevisst eller ubevisst) hvilke verdier han eller hun velger å legge vekt på ved en evaluering, og disse valgene reflekteres i skåren. Ved bruk av en automatisk skårregel må det derimot angis hva slags verdisystem som skal brukes som standardoppsett.
- **Evalueringer:** For å beregne skåren trengs det å utføres evalueringer, og det er nødvendig å vite hvilke evalueringer som er brukt som grunnlag for skåren. Karaktersetteren må derfor vite hvilke evalueringer som skal utføres for en oppgave.

Alle disse elementene innvirker på hvilken skår som blir gitt. Det er derfor nødvendig å gjøre alle disse opplysningene persistente for å kunne gjøre analyser av tidligere gitte skårer og sammenligne ulike verdisystemer.

4.3 Andre funksjonelle og ikke-funksjonelle krav til JCAT

For å kunne innfri de spesifikke kravene til evaluering og skåring må JCAT også oppfylle flere andre funksjonelle og ikke-funksjonelle krav. Funksjonelle krav er krav knyttet direkte til oppførselen til systemet, mens ikke-funksjonelle krav gir en mer overordnet karakteristikk av systemet.

4.3.1 Funksjonelle krav

Funksjonelle krav til rammeverket er:

- Støtte for å kunne simulere gjennomføringen av ulike typer tester, både eksperimenter og testlets (dette innbefatter ikke simulering gjennom et brukergrensesnitt)
- Støtte for persistens for å kunne lagre store mengder data
- Støtte for å håndtere historiske data med tanke på pensjonerte oppgaver og tidligere skårer
- Støtte for å kunne foreta nye analyser av tidligere oppgaver basert på hittil ukjente krav

I november 2007 ble det gjennomført et piloteksperiment på Simula Research Laboratory der det ble samlet inn mye testdata som skal kunne brukes av JCAT. Disse resultatene skal integreres i rammeverket, og det trengs derfor funksjonalitet for å håndtere dette datasettet.

Itembanken i JCAT skal på sikt videreutvikles til å håndtere vanskelighetsgrad for hver enkelt oppgave. Det vil derfor være nødvendig å rekalibrere itembanken med jevne mellomrom ettersom hver enkelt oppgave får flere skårer knyttet til seg. Det kan også være aktuelt å ”pensjonere” oppgaver som man ikke lenger ønsker å bruke i tester, men disse oppgavene må fortsatt ligge i systemet for fremtidige analyser og sammenligninger av resultater. Det vil derfor være nødvendig at rammeverket kan håndtere historikk for oppgaver og skårer.

Ved å innføre funksjonalitet for å skifte ut verdisystemet som skåren skal beregnes etter må rammeverket kunne håndtere at et svar fra et subjekt kan ha blitt tilegnet flere ulike skårer. JCAT bør også la administratorer definere egne verdisystemer, og gjøre det mulig å foreta nye analyser av de tidligere lagrede dataene basert på disse nye kravene.

4.3.2 Ikke-funksjonelle krav

Prototypeimplementasjonen må i tillegg oppfylle flere ikke-funksjonelle krav:

- Alle komponenter som integreres i rammeverket må være åpen kildekode
- Systemet skal være enkelt å vedlikeholde (oversiktlig og godt dokumentert kode)
- Systemet skal være enkelt å teste
- Systemet skal være enkelt å videreutvikle og utvide med ny funksjonalitet
- Systemet skal være enkelt å portere til andre plattformer

JCAT er et åpen kildekode verktøy, og det er derfor viktig at alle nye moduler også er åpen kildekode. Rammeverket skal jobbes videre med av de to andre i prosjektet samt eventuelle nye masterstudenter, og

det er derfor også svært viktig at systemet er enkelt å vedlikeholde, teste og videreutvikle. Det er også en fordel at systemet er enkelt å portere, slik at valg som tas nå ikke legger begrensninger på fremtidig bruk og utvikling.

5 Design og implementasjon

For å kunne forstå sammenhengen mellom de ulike delene av JCAT vil jeg først redegjøre for de verktøyene jeg har valgt å integrere i rammeverket, og deretter beskrive designet for systemet. Videre følger detaljer rundt prototypeimplementasjonen før det gis et eksempel på anvendelse av rammeverket.

5.1 Verktøy

For å videreutvikle JCAT og legge til ny, nødvendig funksjonalitet var det flere tekniske behov som dukket opp og det ble behov for å innføre nye verktøy i prosjektet. Først og fremst var det behov for en databaseserver for å kunne opprette skjema og tabeller for å lagre data. Klassene i JCAT bygget opp med mye arv og polymorfisme, noe som kan føre til mye arbeid for å mappe databaseradene til forretningslogikken. Det var derfor også nødvendig å ta et valg på om løsningen skulle implementeres slik at den var tett koblet med databasen med manuell håndtering av resultatsett, eller om det skulle innføres et mellomlag mellom applikasjonen og databasen. Nedenfor beskrives det hvilke verktøy som har blitt integrert.

5.1.1 MySQL Server

MySQL Server [37] er en kjent og stabil databaseserver både for enkeltbrukere og flerbrukere, og trenger lite vedlikehold og administrasjon. Produktet får gode testresultater på de fleste områder i flere tester av open source databaseprosjekter på internett. MySQL er et levende prosjekt med mange aktive bidragsytere, og dette gjør at det vil være mulig å oppgradere programvaren til nyere versjoner senere. I tillegg finnes det en rekke åpen kildekode administrasjonsverktøy som kan installeres for å administrere databaseserveren. Dette gjør at andre enkelt kan videreutvikle prototypeimplementasjonen og gir JCAT et fremtidsrettet perspektiv. Siste versjon av produktet er i skrivende stund MySQL Server 5.0, og det er denne versjonen som er brukt ved implementasjon i denne oppgaven.

5.1.2 Hibernate

Hibernate [38] er et objekt/relasjonsmapping (ORM) verktøy designet for bruk i Javamiljøer. Begrepet ORM refererer til teknikker for å mappe en representasjon av data fra en objektmodell til relasjonsdatamodell med et SQL-skjema [39]. Hibernate håndterer mapping fra Javaobjekter til tabeller i databasen. Det tilbys også teknikker for å lagre og hente ut objekter, samt automatisk generering av databaseskjema. Dette gjør at programmereren slipper aspekter som krever mye tid og kodelinjer, som for eksempel håndtering av JDBC resultatsett, opprettelse av tabeller og relasjoner mellom disse tabellene. Det hevdes at programmereren skal kunne lettes for opptil 95 % av arbeidet som kreves ved tradisjonell persistensprogrammering [39]. Rammeverket har i tillegg full støtte for arv, polymorfisme og andre objektorienterte prinsipper. Det er portabelt til alle SQL-databaser, og man kan enten bruke et eget språk HQL (Hibernate Query Language), standard SQL eller andre SQL-dialekter. Siste versjon av produktet er i skrivende stund Hibernate 3, og det er denne versjonen som er brukt ved implementasjon i denne oppgaven.

5.1.3 MySQL Connector/J

MySQL Connector/J [40] er en Javadriver som konverterer JDBC-kall til nettverksprotokollen som MySQL bruker. Driveren er nødvendig for å koble applikasjonen til databasen. Siste versjon av produktet er i

skrivende stund MySQL Connector/J 5.0, og det er denne versjonen som er brukt ved implementasjon i denne oppgaven.

5.2 Design for JCAT

For å beskrive hvordan JCAT er bygget opp gis det først en oversikt over hvordan ulike konsepter er operasjonalisert. Videre vises det konseptuelle klassediagrammet etterfulgt av klassediagrammer for de respektive delene av systemet, og til slutt vises databasemodellen.

5.2.1 Operasjonalisering av konsepter

Tabell 8 viser en oversikt over hvordan jeg har valgt å operasjonalisere konseptene jeg har brukt hittil i oppgaven. Oversikten kan brukes som oppslag for å forstå diagrammene som følger i de neste seksjonene.

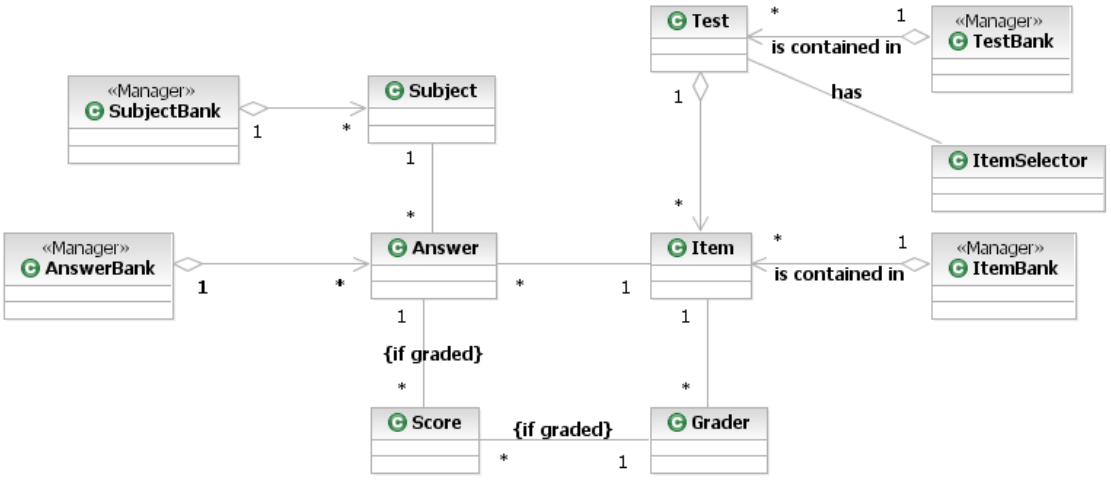
Tabell 8: Operasjonalisering av konsepter i JCAT

Konsept:	**Forklaring:**	**Operasjonalisert i JCAT ved:**
Test	En test kan være bygget opp av en eller flere oppgaver.	Test
Subjekt	En person som tar en test gjennom systemet. Subjektet tildeles en eller flere oppgaver og skal levere svar på oppgaver.	Subject
Oppgave	En oppgave som gis av systemet.	Item
Verdisystem	En strategi for å vektlegge delskårer for å aggregere en total skår.	ValueSystem
Evaluering	En objektiv og verdinøytral evaluering av et kodesegment. Evalueringen kan være statisk eller dynamisk.	Eval
Karaktersetter	Består enten av en skårregel eller en menneskelig karaktersetter. Sier noe om hvordan en oppgave skal bedømmes.	Grader
Skårregel (ScoringMethod)	Har et verdisystem for hvordan delskårer skal vektlegges og regnes sammen til å bli en total skår.	AutomaticScoring
Delskår	Bygger på en evaluering (som er verdinøytral). Ved fastsettelsen av en delskår skjer det en verdivurdering av resultatet av en evaluering.	AutomaticSubScoring
Menneskelig karaktersetter	En person som setter en skår. Vurderingen kan være delvis basert på en automatisk evaluering.	HumanRater
Svar	Den løsningen som et subjekt leverer inn til	Answer

	systemet for evaluering. Et svar kan få en eller flere skårer (basert på ulike verdisystemer).	
Tidsbruk	Den tiden som et subjekt bruker på en oppgave.	Effort
”Oppgavevelger”	Algoritme for å velge neste oppgave for et eksperiment, en CAT (Computer Adaptiv Test) eller et kalibreringseksperiment (tilfeldig rekkefølge på oppgaver).	ItemSelector

5.2.2 Konseptuelt klassediagram for JCAT

For å forstå oppbygningen av JCAT kan vi se på det konseptuelle klassediagrammet som er vist i figur 5:

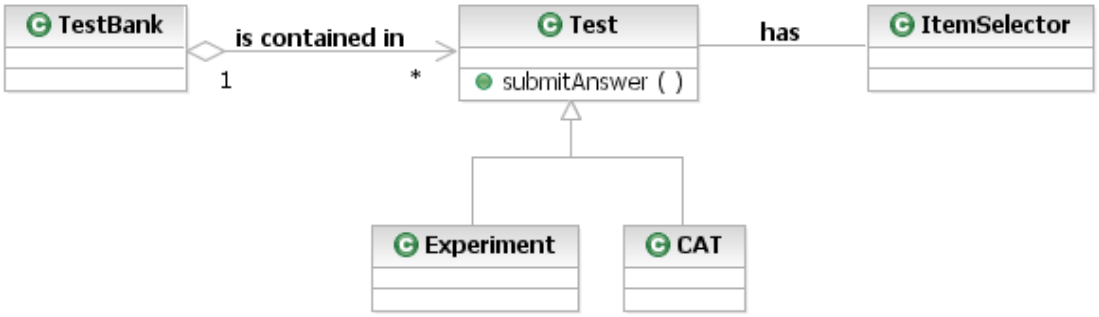


Figur 5: Konseptuelt klassediagram JCAT

Modellen er forenklet for å gi en oversikt over systemet og viser derfor ikke arvehierarkier. For å videre beskrivelse av detaljer vil jeg gå inn på hver enkelt del av systemet. Aller først vil jeg ta for meg Test-hierarkiet. Komplett oversikt over metodekall er ikke tatt med i diagrammene.

5.2.3 Klassediagram for Test-hierarki

En test inngår i en TestBank som er en manager for alle testene. Testen bruker en ItemSelector som brukes for å finne neste Item i testen. Test er implementert som en abstrakt superklasse. Experiment (kontrollert eksperiment) og CAT (Computer Adaptiv Test) er subklasser av Test. Figur 6 viser klassediagram for Test-hierarkiet:

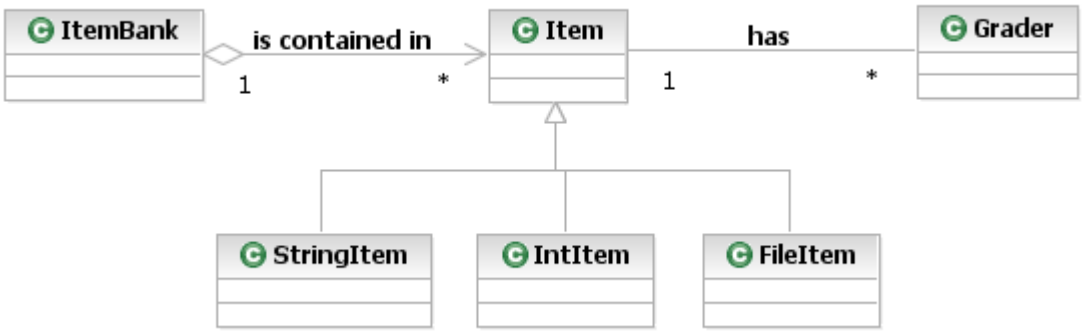


Figur 6: Klassediagram Test-hierarki

Test har en metode submitAnswer() som brukes for å sende et Answer inn i systemet. Answeret evalueres og det beregnes en skår. Denne skåren returneres i et Score-objekt til subjektet. Systemet er bygget slik at en Test kan ha ett eller flere Item som kan være av ulike typer. For å forstå implementasjonen av Test-hierarkiet er det derfor nødvendig å se på hvordan et Item er bygget opp.

5.2.4 Klassediagram for Item-hierarki

Et Item inngår i en ItemBank som fungerer som en manager for alle itemer. Item er en abstrakt superklasse, og har subklassene StringItem, IntItem og FileItem. Ut i fra disse typene kan man beskriver hva slags svar man vil ha tilbake fra subjektet. Figur 7 viser klassediagram for Item-hierarkiet:

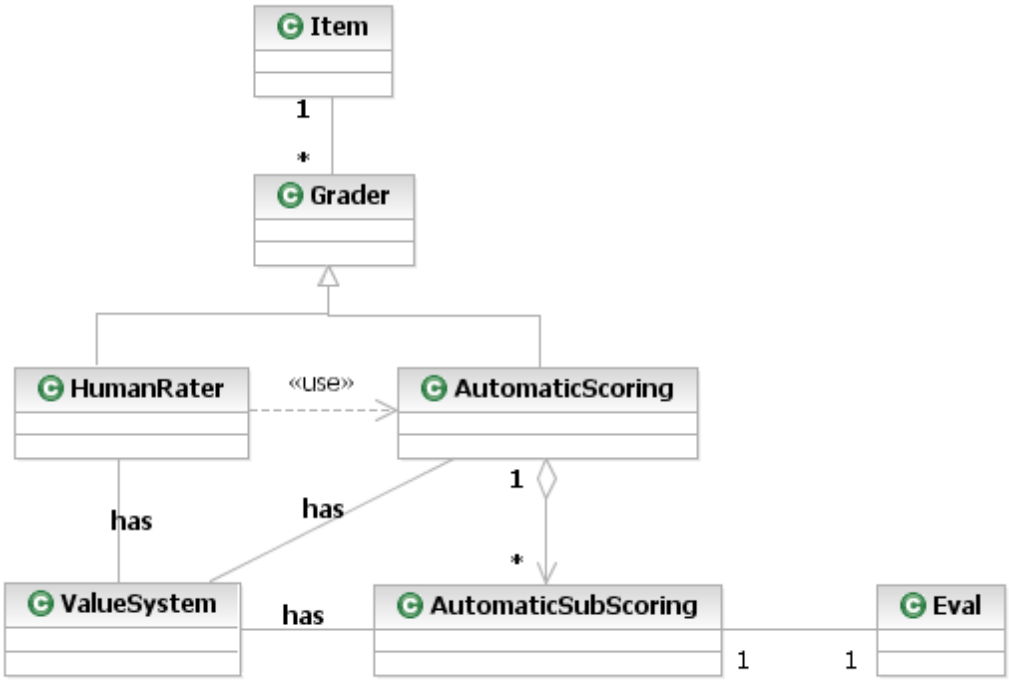


Figur 7: Klassediagram Item-hierarki

Et Item har en Grader som beskriver hvordan svarene for dette Itemet skal skåres.

5.2.5 Klassediagram for Grader-hierarki

En Grader for et Item beskriver hvordan svarene for itemet skal skåres. En oversikt over klassene som hører til under denne abstraksjonen kan sees i figur 8 nedenfor:



Figur 8: Klassediagram Grader-hierarki

Grader er en abstrakt superklasse, og implementeres med klassene HumanRater og AutomaticScoring. En HumanRater kan basere deler av sin vurdering på en AutomaticScoring. AutomaticScoring aggregeres fra én eller flere AutomaticSubScoring, som igjen er basert på en Eval (evaluering). HumanRater, AutomaticScoring og AutomaticSubScoring har alle et ValueSystem.

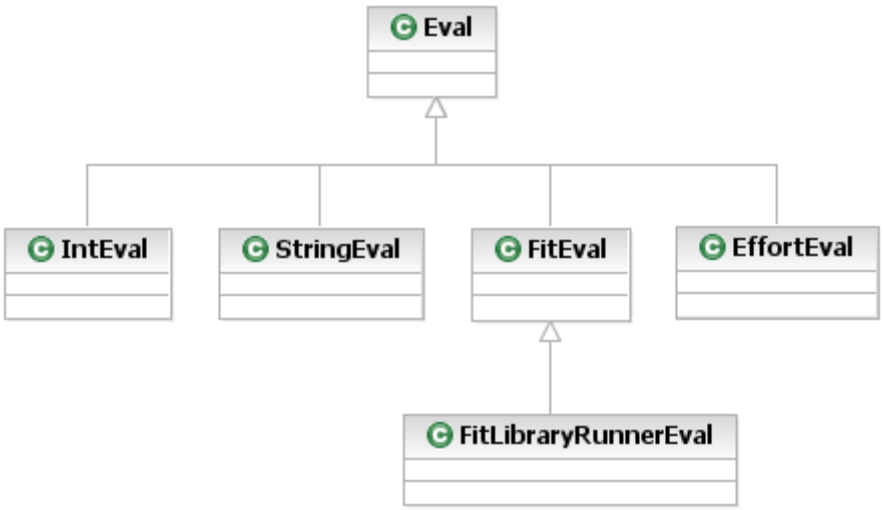
ValueSystem kan igjen ha subklasser. I JCAT finnes det per i dag to ulike verdisystemer som kan brukes, ValueSystemRFE og ValueSystemADD. ValueSystemRFE beskriver hvordan et item som har en Regresjons-, Funksjons- og Effort-evaluering skal skåres. ValueSystemADD adderer alle delskårer for å generere en totalskår.

5.2.6 Klassediagram for Eval-hierarki

For å utføre vurderinger av ulike typer oppgaver er det nødvendig å ha støtte for forskjellige typer evalueringer. Ulike typer evalueringer som støttes av JCAT er:

- String-/Int-evalueringer
- Effort-evalueringer
- FIT-evalueringer

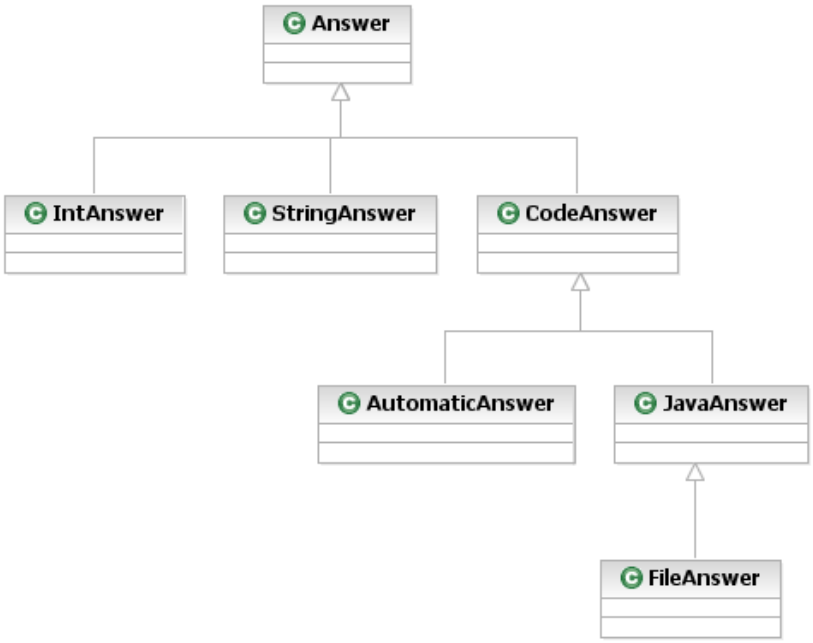
Eval er en abstrakt superklasse som implementeres av klassene StringEval, IntEval, EffortEval og FitEval. De to første utfører en evaluering for å sjekke om en gitt streng eller et heltall matcher det som er angitt som riktig svar. En EffortEval sjekker hvor mye tid som er brukt på å løse en oppgave. FitEval bruker FIT-rammeverket for å utføre en evaluering på koden som er levert av subjektet. Klassediagram for Eval-hierarkiet vises i figur 9:



Figur 9: Klassediagram Eval-hierarki

5.2.7 Klassediagram for Answer-hierarki

Answer er en abstrakt superklasse. IntAnswer, StringAnswer og CodeAnswer arver fra Answer. De to førstnevnte brukes når svaret er et heltall eller en streng. CodeAnswer brukes til å bygge et svar som består av kode (fritt format). Figur 10 viser klassediagram for Answer-hierarkiet:

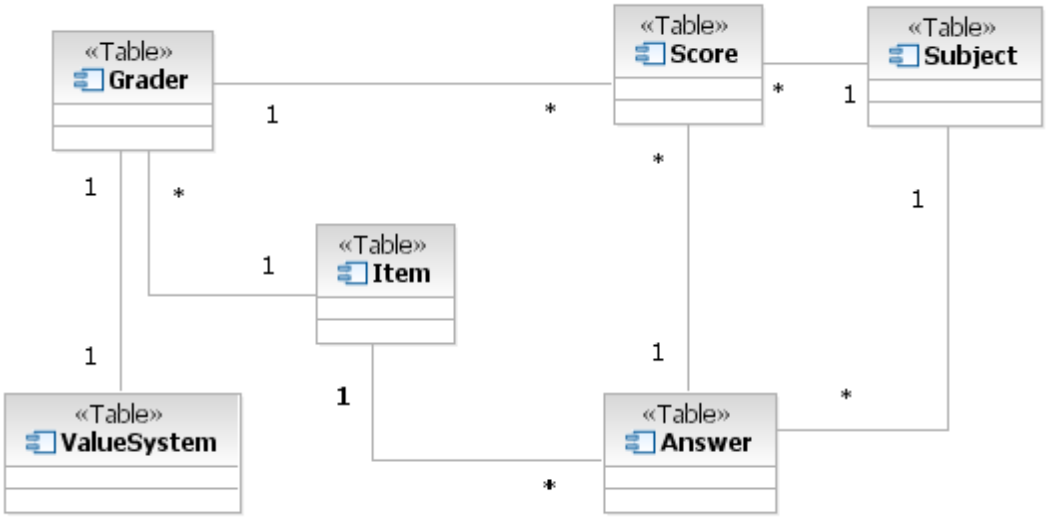


Figur 10: Klassediagram Answer-hierarki

AutomaticAnswer og JavaAnswer arver igjen fra CodeAnswer. Java Answer brukes dersom svaret som skal konstrueres er Javakode. AutomaticAnswer brukes kun for å importere gamle testsvar inn i systemet. Det er her mulig å utvide funksjonalitet med å legge til egne klasser som kan håndtere andre typer programmeringsspråk. FileAnswer arver fra JavaAnswer. FileAnswer har igjen subklasser som implementerer funksjonalitet som er spesifikk for et gitt item, dersom det kreves ytterligere presisjon for å håndtere svaret på korrekt måte.

5.2.8 Databasemodell

Den konseptuelle databasemodell over tabellene og relasjonene i MySQL-databasen kan sees i figur 11:



Figur 11: Konseptuell databasemodell

Modellen er basert på seksjon 4.2 som beskriver hvilke data som må lagres for å håndtere historiske data.

5.3 Implementasjon av persistens med Hibernate

5.3.1 Bruk av designmønstre

Designmønstre kan sees på som en formell metode for å dokumentere en strategi for å løse et gitt problem [41]. Mønstrene fungerer som en mal for hvordan man skal designe og strukturere klassene i et system for å håndtere spesifikke problemområder.

For å implementere persistens i JCAT er det brukt Facade Pattern[42]. Det innebærer at det kun er én klasse, PersistenceProvider.java, som tilbyr persistensfunksjonalitet til de ovenforliggende klassene. De andre klassene kan kalle metoder for lagring og uthenting av objekter, men trenger ikke å bry seg om underliggende detaljer. Applikasjonen trenger heller ikke ha ansvar for å vite om et objekt er lest inn fra minnet, fra databasen eller fra en fil.

For alle administratorklassene (ItemBank, AnswerBank, og så videre) er det brukt Singleton Pattern [43]. Denne strategien er også implementert for HibernateUtil og PersistenceProvider.

5.3.2 Hibernate konfigurasjonsfil

For at Hibernate skal kunne kjøre på maskinen er det satt opp en konfigurasjonsfil i XML. Denne filen bør ligge i src-katalogen til prosjektet for at Hibernate skal kunne finne den ved kjøring. I figur 12 sees en noe forenklet fremstilling av filen hibernate.cfg.xml.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<!-- Database connection settings -->
<property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property
name="connection.url">jdbc:mysql://127.0.0.1/JCAT_db</property>
<property name="connection.username">jcat_user</property>
<property name="connection.password">jcat</property>
<!-- SQL dialect -->
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Mapping -->

<mapping resource="no/simula/jcat/app/item/Item.hbm.xml"/>

<mapping resource="no/simula/jcat/app/scoring/Grader.hbm.xml"/>

<mapping resource="no/simula/jcat/app/scoring/ValueSystem.hbm.xml"/>

<mapping
resource="no/simula/jcat/app/scoring/AutomaticSubScoring.hbm.xml"/>

<mapping resource="no/simula/jcat/app/answer/Answer.hbm.xml"/>

<mapping resource="no/simula/jcat/app/eval/Eval.hbm.xml"/>

<mapping resource="no/simula/jcat/app/answer/Score.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

Figur 12: Hibernate.cfg.xml

Øverst finner vi standard header for alle mapping- og konfigurasjonsfiler for Hibernate. Under ”Database connection settings” finner vi informasjon om hvilken driver og database som skal brukes, samt innloggingsnavn og passord. Under ”SQL Dialect” er det angitt hvilken SQL-dialekt som skal brukes. Denne må endres dersom systemet skal porteres over til en annen database. Under ”Mapping” må man angi alle filene som skal tolkes av Hibernate under mapping av applikasjonskoden til databasen. Neste seksjon tar for seg en slik mappingfil.

5.3.3 Hibernate mappingfiler

Når man bruker Hibernate for å mappe Javaobjekter til en relasjonsdatabase kan man velge to ulike strategier for å binde objektene og databasen sammen. Enten kan man sette opp mappingfiler for alle klassene som skal gjøres persistente, eller man kan velge å annotere klassene med Hibernate Annotations [29, s 33]. I denne oppgaven er det valgt å bruke mappingfiler.

Figur 13 viser et eksempel på en mappingfil for en superklasse med tilhørende subklasser:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="no.simula.jcat.app.item.Item" table="ITEM" abstract="true"
polymorphism="implicit">
<id name="id" type="integer" column="ITEM_ID">
      <generator class="increment"/>
</id>
<property name="itemName">
      <column name="ITEM_NAME" not-null="true" unique="true"/>
</property>
<property name="description" column="ITEM_DESCRIPTION"/>
      <many-to-one name="grader" column="ITEMGRADER"
class="no.simula.jcat.app.scoring.Grader"
                  not-null="true" cascade="save-update" lazy="false"/>
<-- Subclasses -->
<joined-subclass name="no.simula.jcat.app.item.IntItem" table="INT_ITEM">
      <key column="ITEM_ID"/>
</joined-subclass>
<joined-subclass name="no.simula.jcat.app.item.StringItem"   table="STRING_ITEM">
      <key column="ITEM_ID"/>
</joined-subclass>
<joined-subclass name="no.simula.jcat.app.item.FileItem" table="FILE_ITEM">
      <key column="ITEM_ID"/>
</joined-subclass>
</class>
</hibernate-mapping>
```

Figur 13: Item.hbm.xml

Disse mappingfilene brukes også til å generere tabeller i databasen.

5.3.4 Generering av tabeller i databasen

Det er ikke nok med bare mappingfiler for klassene som skal lagres, man trenger naturlig nok også tabeller i databasen. Hibernate har funksjonalitet for automatisk å generere både skjema og tabeller. I filen ”hibernate3.jar” som følger med den siste Hibernate-distribusjonen (Hibernate 3) finnes det en klassefil som heter ”SchemaExport.class”. Ved å sende med alle mappingfilene som argumenter ved kjøring av denne klassen genereres det tabeller i databasen.

Figur 14 nedenfor viser hvilke tabeller som finnes i databasen etter eksportering av skjema:

Figur 14: Tabeller i databasen etter eksportering av skjema

De fleste av tabellene beskriver en respektiv klasse i JCAT. De resterende av tabellene er koblingstabeller som beskriver relasjoner mellom objektene eller tabeller for å uttrykke at et objekt har en liste eller tabell som datatype.

5.3.5 Primærnøkler og generering av id

For å generere primærnøkler i databasen kan man enten velge å tildele en egendefinert nøkkel, eller så kan Hibernate generere nøkkelen selv. For å generere primærnøkler i databasen er det i denne oppgaven valgt å bruke en av Hibernate sine innebygde id-generatorer. Det er flere slike generatorklasser man kan velge

mellom. Klassene har ulike egenskaper, og man bør derfor velge generatorklasse ut i fra hvilke behov applikasjonen har. I denne oppgaven er det brukt generatorklassen ”Native”.

5.3.6 Polymorfisme og arv

JCAT er en applikasjon som er basert på mye arv og polymorfi. For å implementere et arvehierarki med Hibernate er det hovedsaklig fire ulike mappingstrategier som kan velges. Disse er [29, s 192]:

- Tabell per klassehierarki: Gir én tabell i databasen ved at SQL-skjemaet denormaliseres. Det innføres en diskriminatorkolonne som holder informasjon om objekttype.
- Tabell per subklasse: Tabellene for subklassene har assosiasjon til primærnøkkelen i superklassetabellen (relasjonsmodellen blir en en-til-en assosiasjon).
- Tabell per konkrete klasse: Fjerner polymorfisme og arv helt fra SQL-skjemaet.
- Tabell per konkrete klasse med implisitt polymorfisme: Bruker ingen eksplisitt mapping av arv, og har polymorfisk oppførsel under kjøring.

Det er til en viss grad også mulig å implementere varianter og kombinasjoner av disse strategiene. For å implementere arv og polymorfi med Hibernate i JCAT er det her valgt strategien ”Tabell per subklasse”. En fullstendig oversikt over begrensninger for de ulike strategiene finnes i Hibernate Reference Guide [39] som følger med Hibernate-distribusjonen.

5.4 Eksempel på anvendelse

En av hensiktene bak prototypen som er implementert i denne oppgaven er å kunne bytte ut karaktersetter og verdisystem ved skåring av en oppgave. Det er nå mulig å endre karaktersetter og verdisystem når man beregner skår, og derfor kan vi få ut ulike resultater avhengig av hvilket verdisystem som er valgt. Figur 15 viser et eksempel på hvordan en utskrift fra systemet kan se ut etter at et subjekt har fått tre skårer for en oppgave (i dette eksemplet, item 21) basert på ulike verdisystemer.

Subjekt	Verdisystem	Skår_Item21
1	Regression	3
1	Function	5
1	Effort	1

Figur 15: Eksempel på anvendelse av JCAT

Som eksempel i tabellen er det valgt å bruke verdisystemene ”Regression”, ”Function” og ”Effort”. Det er altså disse verdiene som legges særlig vekt på under skåringen av en oppgave og avgjør hvilken skår subjektet får. Tabellen viser hvordan ett og samme subjekt kan få tre ulike skårer basert på tre ulike verdisystemer. Dette er kun en overordnet oversikt, og i selve koden er det brukt andre navn.

6 Evaluering

Ved en implementasjon er det alltid fornuftig å tenke igjennom hvorfor man har tatt de valgene man har tatt og om det er noe man kunne ha gjort annerledes. For å evaluere denne løsningen gjøres det først en analyse av de implementasjonsspesifikke delene av systemet. Det evalueres deretter hvordan løsningen har blitt i henhold til kravene som ble definert i kapittel 4. Videre følger en sammenligning av JCAT med andre rammeverk før det gis noen korte synspunkter på arbeidsprosessen for oppgaven.

6.1 Evaluering av implementasjon

For å evaluere implementasjonen gis det først en analyse av implementasjonen av designmønstrene som er brukt og hvilke som eventuelt kunne ha vært brukt i tillegg. Videre følger en begrunnelse for verktøyene som er valgt å integrere i JCAT, før det analyseres strategier som er valgt for mapping, generering av id, polymorfisme og arv, samt persistente og frakoblede objekter.

6.1.1 Bruk av designmønstre

For å implementere persistens i JCAT er det brukt Facade Pattern. Ved å programmere klassene etter denne abstraksjonen får vi et enkelt grensesnitt fra applikasjonen mot alle typer databasekall. Dette gjør at implementasjonen av persistenslaget enkelt kan skiftes ut eller bygges videre på. Facade Pattern er også en god metode for å bryte opp funksjonaliteten i systemet i mindre biter for å gi lav kompleksitet. Dette kalles separation of concerns (SoC) [44].

Jeg sett på muligheten for å implementert Data-Access Object (DAO) Pattern [29, side 709] for persistens. Jeg følte likevel at dette ikke var nødvendig for å kunne tilby den funksjonaliteten som skulle implementeres. I og med at entitetsklassene allerede var implementert da jeg startet arbeidet har jeg valgt å beholde disse noenlunde som de var og heller ha fokus på å utvikle ny funksjonalitet.

I ettertid av implementeringen har jeg sett at det kunne ha vært hensiktsmessig å bruke Strategy Pattern [29, s 705] for å implementere skåringabstraksjonen. Den implementerte løsningen ligger tett opptil dette mønstret, bortsett fra at det ikke brukes grensesnitt (interface) i tillegg til vanlige klasser. Det kunne kanskje ha vært spart noe tid dersom jeg hadde satt meg inn i dette mønstret før jeg begynte på utviklingen av skåringabstraksjonen.

6.1.2 Valg av verktøy

MySQL Server og Hibernate er som tidligere nevnt basert på åpen kildekode og er mye brukt på verdensbasis. Som en følge av dette er de godt testet, og har relativt lite feil. Det finnes også mange ildsjeler over hele verden som legger mye jobb i å videreutvikle gode åpen kildekode verktøy, og det slippes derfor oppdaterte versjoner med ny funksjonalitet relativt hyppig. Det er i tillegg spesifikke trekk ved hvert enkelt av verktøyene som var avgjørende:

MySQL Server: MySQL Server har fått gode testresultater hos flere nettsteder, og den er enkel å administrere og vedlikeholde. JCAT-databasen kjøres per i dag lokalt på hver enkelt utviklers maskin, men den skal på sikt kunne settes ut på en server slik at alle brukerne kan aksessere de samme dataene. MySQL er et godt valg for å ivareta dette kravet. En annen avgjørende faktor ved at jeg valgt MySQL Server var at jeg hadde kjennskap til rammeverket fra før, både gjennom skole og fritid. Jeg oppfatter MySQL som en ryddig og grei database som ivaretar behov til transaksjonshåndtering, sikkerhet og andre viktige aspekter.

Hibernate: I implementasjonen av JCAT er det mye arv og polymorfisme, og det var derfor tidsbesparende for meg å velge et verktøy som gir støtte for O/R-mapping.

Jeg hadde ingen tidligere erfaring med Hibernate og en viktig grunn til valget mitt er at jeg fikk sjansen til å lære meg en ny teknologi. En utfordring har vært at jeg ikke har hatt noen å spørre om spesifikke spørsmål og detaljer, og løsningen har vært å lese forumposter på internett. Dette har fungert relativt bra, og i og med at det er finnes så mange brukere av Hibernate har det ofte vært andre som har hatt de samme spørsmålene som meg.

6.1.3 Hibernate mappingfiler og annotasjoner

I starten av arbeidet med oppgaven jobbet jeg med flere steg-for-steg guider for Hibernate for å sette meg inn i den grunnleggende funksjonaliteten. I disse guidene var det strategien med mappingfiler som hovedsaklig ble brukt. Imidlertid har jeg sett at det er veldig mange som heller bruker Hibernate Annotations, og at dette er mer utbredt i næringslivet. Hibernate Annotations skal være enklere å bruke enn mappingfiler ved at metadata legges til rett i koden fremfor at man må opprette og vedlikeholde separate filer [29, s 33]. Etter å ha jobbet med mappingfiler i starten av oppgaven har jeg derimot fortsatt å bruke dette fordi jeg synes det var viktigere å legge vekt på ny funksjonalitet fremfor å bruke tid på å migrere koden til å bruke annotasjon. Det kunne likevel ha vært enklere å bruke Hibernate Annotations ved at man da ville ha sluppet å administrere mappingfilene i tillegg til resten av koden.

6.1.4 Primærnøkler og generering av id i databasen

For å generere id-er i databasen har er det valgt generatorklassen ”native”. Denne strategien kan brukes dersom man ikke har spesielle krav til id-generering. ”Native” kan brukes uavhengig av underliggende database og algoritmen velger andre genereringsstrategier ut i fra hvilken database som brukes. Mapping-metadataene er derfor portable til andre databasesystemer uten at det trengs å gjøres endringer [29, s 167].

6.1.5 Polymorfisme og arv

For polymorfisme og arv er det valgt strategien ”Tabell per subklasse” [29, s 203]. Dette har blitt gjort fordi det var denne framgangsmåten som var mest intuitiv å forstå som nybegynner med Hibernate. Strategien gir også enklere spørringer og har ingen begrensninger som sees som problematiske for funksjonalitet som JCAT har eller kan komme til å trenge.

6.1.6 Proxier og lasting av objekter

Som standard bruker Hibernate en strategi som kalles ”lazy loading” for å hente inn objekter og lister fra databasen. Dette betyr at Hibernate kun laster inn de objektene det spesifikt spørres etter [29, 564]. Når vi spør etter et item (for eksempel ved hjelp av id-en i databasen) er det kun dette itemet og ingenting annet som lastes inn i minnet. Men det som blir tilgjengelig i persistenskonteksten er ikke en instans av et itemobjekt, derimot er det en proxy som skal fungere som en plassholder for det aktuelle objektet under kjøringen av programmet. Hibernate sjekker alltid om det er mulig å returnere en proxy i stedet for å kjøre en spørring i databasen. Innlastingen av det virkelige objektet skjer derfor ikke før dette aksesseres for første gang. Den samme strategien brukes for collections, men da kalles plassholderen for en collection wrapper [29, s 564]. Dette er ressursbesparende dersom man jobber med store objekter og lister. Kallene til databasen begrenses til et minimum vet at objektene ikke hentes ut før de skal brukes, men det forutsettes at man jobber med objektene innenfor den samme persistenskonteksten. Problemene oppstår når man

lukker koblingen til databasen og dermed avslutter en persistenskontekst. Når man da aksesserer et objekt utenfor denne konteksten er det ikke lenger mulig for Hibernate å instansiere collectionene og objektene. Man får derfor kjøringsfeil.

I JCAT er objektene relativt små, og ytelse er heller ikke et kritisk problem. Vi har også behov for å jobbe med objektene utenfor persistenskonteksten, altså er objektene frakoblede. For å unngå at Hibernate genererer plassholdere i stedet for å lese inn og instansiere de ekte objektene er det derfor valgt å legge til en parameter i mappingfilene for entitetene. Dette kan sees i mappingfilen i seksjon 5.3.3. Her ser vi parameteren ”lazy=false” er lagt til for mange-til-en-assosiasjonen Grader. Dette gjør at Hibernate bruker strategien ”eager fetching” for uthenting av objekter fra databasen. Objekter lastes dermed inn i minnet med alle entiteter og lister med én gang. Dette reduserer ytelsen, men JCAT er ikke et så stort system at dette har noen praktisk innvirkning på responstiden.

Dersom det ikke hadde vært behov for å jobbe med objektene frakoblet ville det ha vært enklere og mer effektivt å jobbe med objektene etter ”lazy loading”-strategien.

6.2 Validering av krav

Det sees her på hvorvidt den implementerte prototypen tilfredsstiller kravene som ble definert i kapittel 4. Våren 2008 skal det foreligge en ny prototypeimplementasjon av JCAT. Det beskrives derfor også hva som kan gjøres av videre arbeid for den forestående og fremtidige implementasjoner av rammeverket.

6.2.1 Spesifikke krav for evaluering og skåring

Oppsummert fra kapittel 4 er de spesifikke kravene vedrørerende evaluering og skåring: å ha et klart skille mellom evaluering og skår, at utregning av skåren for en oppgave bør være basert på et verdisystem og at dette verdisystemet bør kunne skiftes ut for å kunne generere nye skårer basert på tidligere evalueringer.

Klart skille mellom evaluering og skår: JCAT er nå implementert slik at en evaluering er verdinøytral. Evalueringene fungerer som rene observasjoner av et fenomen der det kun registreres i hvor stor grad koden oppfyller et krav. Resultatet fra evalueringen tolkes deretter for å tillegges en verdi. Dette resultatet kaller vi en delskår.

Utregning av skåren for en oppgave bør være basert på et verdisystem: Delskårene legges sammen og vektlegges på bakgrunn av et verdisystem. Verdisystemet angir hvilke delskårer som skal prioriteres i fastsettelsen av sumskåren eller om alle delskårene skal telle likt. Sumskåren for en oppgave avhenger derfor sterkt av hvilket verdisystem som er benyttet.

Verdisystemet bør kunne skiftes ut for å generere nye skårer: Det er nå mulig å skifte ut verdisystemet for å regne ut skårer for oppgaver på nytt, i etterkant av at en test er utført. Denne skåren vil i de fleste tilfeller bli ulik den første skåren fordi den baserer seg på et annet verdisystem.

Relatert til disse punktene er det hovedsakelig to nye aspekter som dukker opp som noe som trenger videre arbeid:

Fleksible verdisystemer for delskår: Det brukes kun fleksible verdisystemer ved utregning av sumskåren for en oppgave. Dette betyr at det ikke er mulig å skifte ut verdisystem for å fastsette en delskår. Delskårene vil derfor være påvirket av kriteriene som brukes for å tillegge resultatene fra evalueringene verdi. Dette virker igjen inn på totalskåren og gjør at målingene kan inneholde feil.

Sumskår og fleksible verdisystemer for test: Resultatet av å utføre en test i JCAT er et sett med flere skårer, én for hver enkelt oppgave. Mer hensiktsmessig ville det ha vært dersom et subjekt hadde fått én totalskår for hele testen. Dette er ikke implementert per i dag, men bør være relativt enkelt å legge til teknisk. I denne sammenheng er det imidlertid viktig å tenke igjennom hvordan denne testskåren bør beregnes. Vi får da det samme problemet som med skårene og delskårene, om sumskår fra hver enkelt oppgave skal telle likt eller om noen skårer skal noen vektlegges mer enn andre. Det kan derfor være aktuelt å implementere fleksible verdisystemer for å beregne testskårene. Det er planlagt å legge til støtte for Polytomous Rasch modell og klassisk test-teori i senere versjoner av JCAT.

Definisjon av flere verdisystemer: Det er hittil kun to ulike verdisystemer som er registrert i databasen. Dette er tilstrekkelig for å kunne teste funksjonaliteten som er implementert, men for å kunne få fornuftige resultater er vi nødt for å definere flere karaktersettere og verdisystemer som kan brukes til å beregne mer realistiske skårer.

6.2.2 Funksjonelle krav

De viktigste funksjonelle kravene til rammeverket er at det har støtte for persistens, at man kan simulere gjennomføringen av en test, at rammeverket kan håndtere historiske data, samt at det kan gjøres analyser av tidligere oppgaver basert på hittil ukjente krav.

Støtte for persistens: JCAT har i dag funksjonalitet for å kunne lagre oppgaver, svar og skårer i en database. I tillegg lagres det informasjon om hvordan svarene skal skåres avhengig av hvilken karaktersetter/verdisystem som er valgt. Kravet er derfor innfridd, men databasefunksjonaliteten er foreløpig ikke fullstendig testet.

Simulere gjennomføringen av en test: JCAT har nå støtte for at et fiktivt subjekt skal kunne gjennomføre et eksperiment. Imidlertid er det foreløpig ikke støtte for andre typer tester som testlet og tester med CAT-struktur.

Deloppgavene i en testlet er i bunn og grunn like de andre oppgavene i systemet. Disse kan derfor lagres på lik linje med øvrige oppgaver, og selve testleten kan da innholde en kjøreplan over hvilke oppgaver som skal inkluderes. Problemet med testleter er at grunnlaget for å beregne sluttresultatet av testleten ofte kun baseres på siste response. Dette vil si at et subjekt kan bruke mye tid på de foregående deloppgavene i testleten og levere gode løsninger på disse, men så løper tiden ut slik at han/hun ikke rekker å levere tilfredsstillende kode på siste deloppgave. Dersom det leveres utilstrekkelig eller ikke kjørbar kode for siste deloppgave vil subjektet få hele testleten underkjent, uavhengig av kvaliteten på løsningene på de foregående deloppgavene. En løsning på dette kan være å ta vare på alle responsene som subjektet har gitt. Med denne tilnærmingen vil vi kunne gå tilbake på de tidligere responsene til subjektet og se på de foregående løsningene. Skåren på testleten kan dermed fastsettes avhengig av kvaliteten på den siste deloppgaven som er besvart kombinert med antall ubesvarte og underkjente oppgaver. Vi vil med dette oppnå en større nyanse i fordelingen av skårene, og det forkastes ikke brukbare data. Støtte for testleter i JCAT er i skrivende stund under utvikling av andre involverte i prosjektet.

Det har også blitt gjort noe arbeid for å implementere funksjonalitet for å dynamisk velge neste oppgave i en test. Dette legger grunnlag for å kunne implementere støtte for tester med CAT (Computer Adaptive Testing)-struktur der rekkefølgen på oppgavene i testen er avhengig av hvilke løsninger et subjekt har gitt på de foregående oppgavene. For å oppnå dette må det også legges til rette for kalibrering av oppgavene i itembanken. Adaptiv testing er basert på at alle oppgavene har fått tildelt en vanskelighetsgrad. Dette er noe

som ikke er inkludert i denne prototypeimplementasjonen, men som er planlagt i nær fremtid av andre deltakere i prosjektet. Vanskelighetsgraden for en oppgave kan anslås for eksempel ved å regne ut gjennomsnittet av oppnådde skårer for en gitt oppgave.

Håndtering av historiske data: Ved gjennomføringen av en test får subjektet en skår for hver oppgave. Disse skårene lagres i databasen sammen med opplysninger om hvilken oppgave, svar og karaktersetter de hører sammen med. Det er ikke utviklet funksjonalitet for at et subjekt skal kunne få én totalskår for hele testen, og det er kun sumskårer for hver enkelt oppgave som lagres. Et problem ved lagringen av disse oppgaveskårene er at det ikke er lagt inn begrensninger for skårtabellen, slik at skårer med samme id for oppgave, svar og karaktersetter kan lagres som to forskjellige rader. Når vi kjører testene flere ganger etter hverandre blir det derfor duplikate verdier i databasen. Det er heller ikke lagt til rette for å kunne ”pensjonere” en oppgave uten at den slettes fra databasen. Det trenges altså å utvikles en strategi for å kunne håndtere flere versjoner av dataene i databasen.

Analyser av tidligere oppgaver basert på hittil ukjente krav: Ved å ha støtte for persistens i JCAT åpner vi for å kunne analysere data fra en test og regne ut skårer i etterkant av at testen er avsluttet. De dataene som lagres sammen med skåren gjør at vi har oversikt over hvilken oppgave og hvilket svar og verdisystem skåren er basert på. Det er også mulig å legge til nye verdisystemer og karaktersettere som kan ha spesielle krav til hvordan skåren for en oppgave skal regnes ut.

6.2.3 Ikke-funksjonelle krav

De viktigste ikke-funksjonelle kravene til rammeverket er at det er basert på åpen kildekode, at det er enkelt å vedlikeholde, teste, utvide og portere.

Åpen kildekode: Alle verktøyene jeg har valgt å integrere i JCAT er åpen kildekode.

Vedlikeholdbarhet: Under arbeidet med oppgaven har det blitt etterstrevet å skrive ren og godt dokumentert kildekode. Bruk av designmønstre er også med på å gi vedlikeholdbar kode [41], og det har blitt utviklet krav- og designdokument som beskriver rammeverket. Vi har fulgt smidige metoder og har ikke tegnet flere modeller enn det vi føler har vært strengt tatt nødvendig for egen og andres forståelse av systemet, men det er likevel mye mer dokumentasjon nå enn da vi startet opp. Det har blitt laget diagrammer for å vise avhengighet (dependency) mellom klasser, og sekvensdiagram for å viser normal flyt gjennom systemet. Brukstilfeller er beskrevet, og det har blitt satt opp en arkitekturmodell. Denne dokumentasjonen finnes nå samlet i et kravdokument og et designdokument for JCAT. Selv om systemet har blitt mer komplekst vil det dermed være enklere for andre å sette seg inn i systemet nå enn før.

Verktøyene jeg har integrert i JCAT er godt kjente og mye brukte, og det finnes mye dokumentasjon for disse på internett. I tillegg har jeg utarbeidet en installasjonsguide for MySQL Server, Hibernate og Connector J som kan brukes av fremtidige prosjektdeltakere for å installere disse verktøyene for bruk sammen med JCAT. Dokumentasjonen er testet ut på de nåværende prosjektdeltagerene og skal være fullstendig og lettfattelig. Installasjonsguiden er lagt ved denne oppgaven som vedlegg (*Vedlegg 2: MySQL og Hibernate Installation Guide*).

Testbarhet: Ved oppstart av prosjektet var det allerede flere JUnit og FitNesse-tester i systemet. Underveis har vi modifisert disse slik at de tester den nye funksjonaliteten vi har utviklet, samt at vi også har skrevet endel nye tester. Disse testene har blitt brukt hyppig for å kontinuerlig sjekke at systemet fungerer som det

skal. Denne fremgangsmåten er fornuftig både for å sjekke at alt installeres riktig og for å sjekke kontinuerlig at man ikke brekker gammel funksjonalitet ved implementering av ny kode.

Utvidbarhet: Under arbeidet med rammeverket har det blitt diskutert muligheten for å kunne bruke JCAT ikke bare til å måle programmeringsferdigheter, men også til å teste andre typer ferdigheter. Arkitekturen og strukturеringen av klassene i systemet gjør det mulig med relativt enkle grep å skifte ut oppgaver og hvilke typer evalueringer som skal brukes. Det vil dermed være mulig å bruke rammeverket til å vurdere ferdigheter innenfor felter som for eksempel matematikk, rettskrivning og så videre. Støtte for flervalgstester er også en reel mulighet, da funksjonaliteten for dette allerede finnes i systemet i form av streng- og heltallsevalueringer.

Portabilitet: JCAT er programmert i Java (versjon 5.0) og er hovedsaklig beregnet for bruk på Microsoft-plattform. Java er et fleksibelt programmeringsspråk som er kompatibelt med de fleste andre verktøy. Man står derfor relativt åpent på hvilke nye verktøy man ønsker å innføre i prosjektet. De nye verktøyene som har blitt integrert i denne oppgaven er også fleksible og portable. Både Java, MySQL og Hibernate er produkter som jevnlig videreutvikles og oppdateres med ny funksjonalitet som igjen kan integreres i JCAT.

I tillegg til kravene nevnt ovenfor er det andre sider ved et slikt rammeverk som også må tas hensyn til. Det er flere viktige aspekter som ikke er vektlagt i denne oppgaven som kan arbeides videre med i nye prototyper av JCAT for at rammeverket skal kunne brukes for å gjennomføre realistiske tester for ekte subjekter:

Ytelse: Rammeverket er på teststadiet, og fokus har derfor vært på å utvikle ny funksjonalitet fremfor å bruke mye tid på å sikre at applikasjonen har høy ytelse.

Brukergrensesnitt: Rammeverket har per i dag ikke brukergrensesnitt, og testing foregår derfor gjennom utviklingsmiljøet. Det er utviklet FIT-tester som simulerer brukeroppførsel.

Pedagogisk design: For at et rammeverk for automatisk evaluering skal lykkes, er det viktig med et pedagogisk design [2]. På denne måten kan brukerne og de som ønsker å vurdere oppnå den ønskede tilliten som kreves for at rammeverk skal fungere i praksis. Det er også viktig å definere gode og konsise oppgaver [5] slik at det ikke er rom for misforståelser som kan føre til at subjekter får en dårligere poengsum enn de fortjener. Et brukergrensesnitt for JCAT bør derfor være enkelt og lettfattelig for å redusere feilmargin ved testene.

Sandboxing: Når man evaluerer programmeringskode er det viktig å ta høyde for at koden kan inneholde elementer som kan være skadelige for maskinen som utfører evalueringen. Et ukjent program bør derfor ikke kunne operere fritt på testmaskinen. JCAT trenger derfor støtte for å kunne sette opp et avgrenset kjøringsmiljø eller sandbox som isolerer koden fra resten av systemet ved å avgrense rettighetene til programmeringskoden [45]. Dette har blitt diskutert som noe som skal integreres i prosjektet i nær fremtid.

Stimuli for oppgaver: For bruk i en reell situasjon er det nødvendig å utvikle oppgavebeskrivelser, eller stimuli. Hittil har det kun blitt simulert gjennomføring av tester, og følgelig har vi ikke hatt behov for stimuli som forklarer et subjekt hva en oppgave går ut på. For at testresultatene skal være pålitelige er det viktig at stimuli som skal gis til subjektene er utformet slik at det ikke kan oppfattes som tvetydig eller uklart [2]. Det kan være hensiktsmessig å utvikle stimuli ved å bruke XML-tagger for å merke opp ulike deler av oppgavene [2]. Dette er noe som kan være aktuelt å innføre i JCAT.

Flere typer evalueringer: Som beskrevet tidligere i dette kapitlet er de fleste evalueringene i JCAT evalueringer som måler selve produktet. Det kan derfor være en idé å innføre prosessevalueringer som kan si noe om programmererens oppførsel under gjennomføringen av en test.

Brukeradministrering og innlogging: For å identifisere subjektene og kunne kontrollere hvem som har rettigheter til å gjøre hva i systemet er det viktig å implementere støtte for brukeradministrering og innlogging. Rammeverket SESE [26] er i likhet med JCAT utviklet ved Simula Research Laboratory. SESE har støtte for brukeradministrering og har stor fleksibilitet for å sette opp ulike typer eksperimenter med forskjellige oppgaverekkefølger og oppgavemateriale. Det er derfor diskutert muligheten for at JCAT skal kunne integreres med SESE. SESE vil da fungere som front-end, mens JCAT vil være back-end.

6.3 JCAT sammenlignet med andre rammeverk for automatisk evaluering

For å sette en skår på en oppgave tar andre rammeverk i bruk ulike strategier for å bedømme hvilke løsninger som er bedre enn andre løsninger. Som beskrevet i seksjon 2.3.1 er det mange ulike kvalitetsfaktorer som kan brukes for å beskrive hva det vil si å være dyktig som programmerer. I seksjon 2.5.3 ser vi også at det er mange fremgangsmåter for å måle de samme kvalitetsfaktorene. Dette vil si at selv om to rammeverk gir seg ut for å måle de samme kvalitetsfaktorer er det ikke dermed sagt at et subjekt vil skåre likt i begge rammeverkene. Det er derfor ikke mulig å sammenligne skårer fra to forskjellige rammeverk for et subjekt.

Ved skårsetting av oppgaver blir det heller ikke alltid tatt hensyn til at det er flere måter å beregne en skår på. Det forutsettes dermed at den strategien som er valgt for å regne ut skårene er den eneste riktige. Resultatet av testene sier derfor kun noe om hvilke subjekter som er gode til å skrive kode som passer inn med verdisystemet som er brukt i rammeverket.

Ved å bruke fleksible verdisystemer i JCAT kan vi generere ulike skårer basert på ulike kriterier, og resultatet blir at det er spesifikke kvalitetsfaktorer som måles. Disse kvalitetsfaktorene kan gjerne ses som ulike former for dyktighet. Skårene fra de forskjellige verdisystemene kan enten analyseres separat, eller de kan igjen kombineres på ulike måter for å kunne trekke mer nøyaktige og rettferdige slutninger om dyktigheten til en person. Den store forskjellen fra andre rammeverk ligger i at JCAT tilbyr en ytterligere spesifikasjon av hva som måles ved en test.

6.3.1 Testvaliditet i JCAT

Det kan hevdes at JCAT kan gi relativt valide tester totalt sett fordi det er mulig å definere spesifikt hvilke ferdigheter som faktisk måles. Ved at det er mulig å definere verdisystemer som vektlegger spesifikke vurderingskriterier er det i større grad mulig å måle det man ønsker. Det som derimot ikke er tatt hensyn til i denne oppgaven er i hvilken grad de vurderingskriteriene som defineres kan sies å være valide. Vi kan ikke være sikre på at vurderingskriteriene vi har spesifisert er de beste for å måle for eksempel funksjonell korrekthet for løsningen av en programmeringsoppgave. På den annen side kan vi gjøre små endringer i vurderingskriteriene og sammenligne med tidligere skårer for å se hvilke utslag vi får i resultatene. JCAT gjør det med andre ord mulig å gjøre nye analyser for å sjekke validiteten av vurderingskriteriene våre. Dette er en forbedring av funksjonaliteten fra mange av de andre rammeverkene der man ikke har noen mulighet til å sammenligne resultatene på denne måten.

6.3.2 JCAT som standardisert databaseløsning for andre rammeverk

I artiklene jeg har undersøkt som omhandler de øvrige rammeverkene har det vært lite fokus på hva slags databaseløsning som er valgt. Unntaket er rammeverk som HoGG [Morris] som bruker en MySQL database der det lagres kildekode, bytekode og eventuelle feilmeldinger. I tillegg lagres resultater fra kjøring av programmene. Det er uklart hva som er grunnen til mangelen på informasjon om databasefunksjonalitet i de andre artiklene. Det kan være fordi dette ikke blir sett på som et viktig aspekt, eller det kan være fordi det ikke er støtte for dette.

Ifølge [2] er de fleste systemene utviklet internt i institusjonen der det skal brukes, og det finnes derfor foreløpig ingen felles standard for hvordan rammeverkene skal bygges opp eller hvordan oppgavene skal defineres. For å effektivt kunne dele den kunnskapen og de gode metodene som finnes for å evaluere er det derfor viktig at verktøyene har mer støtte for interoperabilitet og portabilitet [2].

De fleste av systemene tilbyr kun vurdering av oppgaver i sanntid der subjektene får et resultat på skjermen uten at det lagres data. JCAT lagrer sine data i en database som kan porteres til de fleste plattformer. Teoretisk sett vil det derfor være mulig å la andre rammeverk koble seg opp mot itembanken til JCAT for å bruke ferdig definerte tester for å beregne sine skårer. Dersom flere ulike rammeverk benytter seg av oppgaver som er kalibrert i samme itembank og fastsetter skårer basert på de samme verdisystemene vil det bli mulig å kunne gjøre fornuftige sammenligninger og aggregeringer av skårer gitt av to forskjellige rammeverk. Dette vil være spesielt nyttig for utdanningsinstitusjoner, da det ofte er store variasjoner i vurderingskriterier mellom ulike skoler og lærere [2].

6.4 Evaluering av arbeidsprosess

Vi har jobbet i en prosjektgruppe på tre personer der alle har drevet både med design og utvikling. I utviklingsprosessen har vi jobbet etter smidige metoder for dokumentasjon og har brukt flere elementer fra SCRUM-metodikken. Dette har fungert bra og har ført til at det har vært fokus på kontinuerlig framdrift i prosjektet.

Å sette opp SMART-målene for en oppgave kunne føles noe tungvint, og det var ofte vanskelig å huske på faktisk å skrive disse ned før man startet på oppgaven. Men jeg merket også at jeg ved å definere disse målene ble mer bevisst på hvorfor jeg skulle utføre en oppgave og det var lettere å kartlegge problemer som kunne oppstå underveis i arbeidet. Jeg tror derfor dette kan være en fornuftig strategi å jobbe etter i et prosjekt.

Min tekniske deltagelse i prosjektet har strukket seg over to milepæler på totalt cirka ni uker, og vi har ferdigstilt programmeringskode og dokumenter i henhold til målene for den aktuelle sprinten.

For å dele programmeringskode innad i prosjektet har vi som nevnt i kapittel 4 brukt verktøy for versjonshåndtering. Dette har jevnt over fungert bra, men det har av og til vært en utfordring å integrere kode dersom flere har jobbet på de samme klassene samtidig. En erfaringen jeg har trukket ut av dette er at jeg sett hvor viktig det er å kunne skrive ren og godt dokumentert kode for å kunne samarbeide godt med andre. Alt i alt føler jeg at jeg også har fått mange nyttige erfaringer med å jobbe i en prosjektgruppe og i å samarbeide med andre i utviklingsprosjekter.

7 Konklusjon og videre arbeid

I denne oppgaven har det blitt utviklet en prototype som på sikt skal kunne brukes til automatisk og manuell evaluering og skåring av programmeringsoppgaver for å vurdere ferdigheter.

En begrensning hos andre rammeverk for automatisk evaluering er at det ikke alltid gjøres et klart skille mellom evaluering og skåring. Skårene viser dermed ferdighetene til en person kun sett fra ett perspektiv. Vi kan derfor bare si noe om hvor dyktig en person er innenfor de spesifikke områdene som rammeverket tester.

Denne oppgaven er den del av et større prosjekt ved Simula Research Laboratory. På grunnlag av målet for dette prosjektet om å kunne vurdere dyktighet kreves det at det tas hensyn til at resultatene fra en vurdering har sterke avhengigheter til vurderingskriteriene. Hvilke vurderingskriterier som legges til grunn for å beregne en skår og hvordan de ulike delskårene vektlegges for å utgjøre en sumskår avgjør hvem som blir oppfattet som dyktige innenfor området som testen har til hensikt å si noe om. Gitt at vi vet hva vi ønsker å måle i en test kan vi ved å skifte ut vurderingskriteriene for å fastsette skåren oppnå detaljerte målinger som viser ulike perspektiver av ferdighetsnivået til en programmerer.

Prototypeimplementasjonen som er utviklet tar sikte på å videreføre aspekter ved eksisterende rammeverk og kombinere dette med funksjonalitet for fleksible verdisystemer. JCAT gjør først og fremst et klart skille mellom skår og evaluering. Evalueringer i JCAT defineres derfor som verdinøytrale observasjoner av kode. Disse observasjonene transformeres deretter til delskårer som beskriver rangeringen av en prestasjon i forhold til en annen. For å beregne hvordan disse delskårene skal vektlegges og aggregeres bruker JCAT verdisystemer. Det tilbys muligheten til å definere egne verdisystemer slik at man kan velge hva som skal vektlegges ved en vurdering. Det er mulig å skifte ut verdisystemene for å kunne generere ulike skårer for en og samme løsning fra et subjekt. På denne måten kan JCAT gi oss mer spesifikk informasjon om hvilke områder et subjekt er dyktig innenfor. Skårene lagres i en database. En oppgave kan skåres på flere ulike måter, og vi kan deretter empirisk sammenligne skårene for å finne det beste alternativet. Det er også mulig å sammenligne feilmarginen i tester ved å analysere resultater fra de forskjellige karaktersetterne.

For å utvikle prototypen er det tatt utgangspunkt i en eksisterende versjon av JCAT som er programmert i Java. Et viktig krav har vært at alle deler av prototypen skal være åpen kildekode. For å lagre dataene brukes MySQL Server, og Hibernate brukes som verktøy for O/R-mapping mellom applikasjonen og databasen.

Løsningen som er utviklet kan simulere gjennomføringen av en test for et fiktivt subjekt. Den kan også håndtere enkle historiske data ved at skårene lagres i databasen og kan beregnes på nytt basert på et annet verdisystem. Rammeverket er tilrettelagt for enkelt å kunne vedlikeholdes, testes, videreutvikles og porteres.

En begrensning for denne oppgaven er at det er nødvendig å definere flere ulike typer verdisystemer og karaktersettere for å kunne teste rammeverket mer inngående. I tillegg er ikke støtten for å håndtere historiske data robust nok. Dette området trenger derfor mer arbeid for at man senere skal kunne implementere støtte for adaptive tester i rammeverket. Det er ikke utviklet funksjonalitet for å gi et subjekt én totalskår for en test, men i stedet blir det gitt en skår for hver enkelt oppgave. For å beregne denne testskåren må det bestemmes en strategi for hvordan utregningen skal skje, om det også her skal legges et verdisystem til grunn for vektleggelsen av oppgaveskårene. Det trengs

også å gjøres mer arbeid innenfor brukergrensesnitt, brukeradministrering, sandboxing og utvikling av oppgavebeskrivelser før rammeverket kan brukes for å teste ekte subjekter. Det kan også være nyttig å implementere støtte for flere ulike typer evalueringer, som for eksempel prosessevalueringer.

En generell egenskap ved rammeverk for automatisk å evaluere programmeringsoppgaver er at de ofte er spesialdesignet av og for den institusjonen de skal benyttes innenfor. Det finnes derfor ingen standard for hvordan de er bygget opp, og dette gjør det vanskelig å sammenligne skårer for et subjekt på tvers av rammeverkene. Teoretisk sett ville det være mulig for andre rammeverk å koble seg opp mot itembanken til JCAT for å få tilgang til ferdig definerte tester, oppgaver og verdisystemer. En slik løsning kunne ha vært spesielt nyttig for utdanningsinstitusjoner, da det ofte er store forskjeller mellom evalueringsmetodene som brukes av ulike skoler og lærere.

Referanser

1 Riku Saikkonen, Lauri Malmi og Ari Korhonen. ”Fully Automatic Assessment of Programming Exercises”. *ACM*, 2001.

2 Kirsti Ala-Mutka. “A Survey of Automated Assessment Approaches for Programming Assignments”. *Computer Science Education,*årg. 15 (2005), nr 2.

3 David Jackson. “A Semi-Automated Approach to Online Assessment”. *ITiCSE*, 2000.

4 Fritz Ruehr og Genevieve Orr. ”Interactive Program Demonstration As a Form of Student Program Assessment”. *CCSE,* 2002.

5 Brenda Cheang, Andy Kurnia, Andrew Lim og Wee-Chong Oon. “On Automated Grading of Programming Assignments in an Academic Institution”. *Computers & Education,* årg. 41 (2003), s 121-131.

6 Derek S. Morris. “Automatic Grading of Student’s Programming Assignments: an Interactive Process and Suite of Programs”. *ASEE/IEEE Frontiers in Education Conference*, 2003.

7 Owen L. Astrachan. “Non-Competitive Programming Contest Problems as the Basis for Just-in-time Teaching”. *ASEE/IEEE Frontiers in Education Conference*, 2004.

8 Duane Buck og David J. Stucki. “JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum”. *SIGCSE,* 2000.

9 John McKeogh og Dr. Chris Exton. “Eclipse Plug-in to Monitor the Programmer Behaviour”. *OOPSLA’04 Eclipse Technology eXchange Workshop*, 2004.

10 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell og Anders Wesslén. *Experimentation in software engineering:An Introduction*. Kluwer Academic Publishers, 1999.

11 N. Fenton og S. L. Pfleeger. *”Software Metrics: A Rigourous & Practical Approach”*. 2. utg. International Thomson Computer Press, 1996.

12 Wikipedia. ”Objectivity (science)”. Tilgang: http://en.wikipedia.org/wiki/Objectivity_(science) [22.11.07].

13 Wikipedia. “Value System”. Tilgang: http://en.wikipedia.org/wiki/Value_system [29.11.2007].

14 Lee J. Cronbach. *“Essentials of Psychological Testing”.* 5. utg, Harper & Row, 1990.

15 Wikipedia. ”Test”. Tilgang: http://en.wikipedia.org/wiki/Test [22.11.07].

16 David Thissen og Howard Wainer. *“Test Scoring”.* Lawrence Erlbaum Associates, 2001.

17 Wikipedia. ”ANOVA” Tilgang: http://en.wikipedia.org/wiki/Anova [08.12.07].

18 Wikipedia. ”Test-retest”. Tilgang: http://en.wikipedia.org/wiki/Test-retest [22.11.07].

19 Wikipedia. ”Classical Test Theory”. Tilgang: http://en.wikipedia.org/wiki/Classical_test_theory [22.11.07].

20 S. L. McCall. “Quality Factors”. *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

21 Gene V. Glass og Julian C. Stanley. ”Statistical methods in Education and Psychology”, 1970. *Instrument Design with Rasch IRT and Data Analysis 1*, Murdoch University, 2006.

22 Erik Arisholm og Dag I. K. Sjøberg. ”Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software”. *IEEE Transactions on Software Engineering*, årg. 30 (2004), nr. 8.

23 Dr. Denise Woit og Dr. David Mason. ”Lessons from On-Line Programming Examinations”. *ITiCSE’98*, 1998.

24 Nghi Truong, Paul Roe og Peter Bancroft. “Static Analysis of Student’s Java Programs”. *Australian Computing Education Conference (ACE)*, 2004.

25 Anthony Allowatt og Stephen Edwards. “IDE Support for Test-Driven Development and Automated Grading in Both Java and C++”. *OOPSLA’05 Eclipse Technology eXchange Workshop,* 2005.

26 Erik Arisholm og Dag I. K. Sjøberg. “A web-based support environment for software engineering experiments”. *Nordic Journal of Computing,* årg. 9 (2002), s 234-247.

27 A. T. Chamillard og Jay K. Joiner. “Using Lab Practica to Evaluate Programming Ability”. *ACM*, 2001.

28 Christopher Douce, David Livingstone og James Orwell. “Automatic Test-Based Assessment of Programming: A Review”. *ACM*, årg. 5, nr 9 (2005).

29 Christian Bauer og Gavin King. “*Java Persistence with Hibernate”.* Manning, 2007.

30 Wikipedia. ”Relational Model”. Tilgang: http://en.wikipedia.org/wiki/Relational_model [22.11.07].

31 Wikipedia. ”Object-relational Mapping”. Tilgang: http://en.wikipedia.org/wiki/Object-relational_mapping [22.11.07].

32 Tortoise SVN, v 1.4.5. Tilgang: http://tortoisesvn.tigris.org/ [07.11.2007].

33 Softhouse Consulting. ”SCRUM in five minutes” [online]. Tilgang: http://www.softhouse.se/Uploades/Scrum_eng_webb.pdf [22.11.07].

34 Arina Nikitina. “SMART Goal Setting” [online]. Tilgang: http://www.goal-setting-guide.com/smart-goals.html [15.11.07].

35 Wikipedia. ”JUnit”. Tilgang: http://en.wikipedia.org/wiki/Junit [22.10.07].

36 Wikipedia. ”FitNesse”. Tilgang: http://en.wikipedia.org/wiki/Fitnesse [22.11.07].

37 MySQL Server,v 5.0. Tilgang: http://www.mysql.com/ [22.11.07].

38 Hibernate, v 3.0. King et al. Tilgang: http://www.hibernate.org/ [22.11.07].

39 Hibernate Reference Guide, v 3.2.5 .Tilgang: http://www.hibernate.org/ [22.11.07].

40 MySQL Connector/J, 5.0. Tilgang: http://www.mysql.com/products/connector/j/ [03.11.2007].

41 Wikipedia. ”Design Pattern”. Tilgang: http://en.wikipedia.org/wiki/Design_pattern [22.11.07].

42 Wikipedia. “Façade Pattern”. Tilgang: http://en.wikipedia.org/wiki/Facade_pattern [22.11.07].

43 David Geary. Javaworld, 2003. “Simply Singleton” [online]. Tilgang: http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html [21.11.07].

44 Wikipedia. ”Separations of Concerns”. Tilgang: http://en.wikipedia.org/wiki/Separation_of_concerns [22.11.07].

45 Wikipedia. ”Sandbox (computer security). Tilgang: http://en.wikipedia.org/wiki/Sandbox_(computer_security) [29.11.2007].

Vedlegg 1: Kartlegging av utvalgte rammeverk for automatisk evaluering av programmeringsoppgaver

Navn på rammeverk	**Bruksområde**	**Evalueringskriterier**	**Verktøy/kommentarer**
Online Judge (Cheang, Kurnia, Lim og Oon)	Teste programmer under programmeringskonkurranser, samt trening til slike konkurranser	Måler korrekthet, validitet, robusthet, effektivitet (prosessorload og minnebruk, men dette må bestemmes ut i fra hvert enkelt program).	Kan ikke evaluere på vedlikeholdbarhet (om koden lett å forstå, beskrivende variabelnavn, kommentarer, innrykk, kodestandard). Dette må derfor vurderes subjectivt av en assistent. Bruker i tillegg CheaterChecker for å oppdage eventuell plagiering
Web-CAT - Web-based Center for Automated Testing (Edwards + Allowatt og Edwards)	Benyttes av studenter. Bygger på Test-driven development. Rammeverk som setter karakter på både kode og tester.	Tester på kompletthet og korrekthet av testene. Sjekker også stil og kvalitet på koden. Det settes deretter sammen en score der alle de tre førstnevnte punktene utgjør samme prosentandel av sluttkarakteren. På grunn av at man tester egen kode, kan man ikke få veldig dårlig resultat på en oppgave og bra på en annen.	JUnit for måle korrekthet, Clover for å måle kompletthet av testene. Checkstyle for statisk analyse for å se på stil, og PMD for å sjekke kodens optimalitet. Bruker Ant til å ta seg av samkjøring av scorene.
JKarelRobot (Buck, Stucki)	Benyttes av uerfarne studenter og er platformuavhengig. Legger mye vekt på pedagogikk, og gradvis tilegning av kunnskap. Viktig å lære uttrykk før man skal skrive et helt program. Single-step gjennom et program, studentene skal forutsi hva programmet vil gjøre for hvert steg.	Sjekker korrekthet, og gir umiddelbar feedback. Bygget på Bloom's Taksonomi for kognitiv utvikling.	

”Watcher” – an Eclipse Plug-in (McKeogh, Exton)	Overvåker og dokumenterer programmererens oppførsel under kodingen. Registrerer bevegelse av musen, tastetrykk, scrolling og endringer i dokumentet. Skal kunne identifisere ulik oppførsel mellom ulike programmerere og programmeringsteknikker. Høyt detaljnivå på informasjonen som lagres slik at man kan gå tilbake for videre analyser.	Tester programmererens effektivitet/programmeringspraksis ved å måle tidsbruk for browsing, scrolling, museklikk og tastetrykk under en sesjon. Områder der koderen har brukt mye tid kan legges vekt på i undervisning. I industrien kan disse områdene være nødvendige å refaktorere eller omstrukturere. Kan også brukes i prosjektanalyse og oppdagelse av feil.	Registreringene lagres i xml-format, og konverteres deretter til Excel-format som muliggjør grafisk fremstilling. Grafer med enten en kodeseksjon eller hele koden. Problem med at perioder der programmereren var inaktiv ga store utslag på resultatene. Likevel kan dette gi et mer realistisk bilde av tiden som brukes.
ELP – Environment for Learning to Program (Truong, Roe, Bancroft)	Benyttes av studenter. Bruker både software engineering metrics and relative comparison for å bedømme kvalitet og gi tilbakemelding	Statisk og dynamisk analyse. Kvalitative og kvantitative analyser. Måler god programmeringspraksis. Korrekthet evalueres på strukturell likhet. Dersom denne feiler helt, foretas det en subjektiv vurdering.	Alle øvelsene er ”Fill in the gap”. Bruker ANTLR for å konstruere Abstract Syntax Tree i XML-format.
SESE – Simula Experiment Support Environment (Arisholm, Sjøberg, Carelius)	Profesjonelle programmerere utførere nøye kontrollerte eksperimenter. Mye vekt på at eksperimentene skal være så realistiske som mulig. Rammeverket skal automatisere mye av logistikken ved eksperimentene.	Real-time monitorering	Mener at ulikheter i kunnskaper og ferdigheter blant profesjonelle utviklere er mye større enn forskjellene blant studenter. Forfatterne ønsker seg mer kvalitativ info om hva som blir tenkt og gjort for å komme fram til de resultatene de gjør. Planlegger å implementere pop-up-dialog som ber brukeren skrive inn hva han/hun har gjort siden de startet på en aktuell oppgave/del.
Topcoder – an online programming contest (Astrachan)	Kurs legger ofte vekt på objektorientering og design. Artikkelforfatterne fremhever viktigheten av algoritmer og problemløsning.	Tester korrekthet ved hjelp av JUnit, og måler tidsbruk.	Legger ikke vekt på kodestil.

Lab-practicum (Chamillard, Joiner)	Benyttes av studenter. Studentene utvikler og tester et komplett program som en løsning på en oppgave i løpet av en makstid. De får tilgang til et syntaks-ark, oversikt over de mest vanlige programmeringsfeilene (med løsninger) samt kursets web-side	Tester på korrekthet. Det utføres deretter en subjektiv vurdering.	Gir ulike oppgaver til studenter som sitter i nærheten av hverandre for å forhindre juks. Utfordring å bestemme om disse ulike oppgavene har samme vanskelighetsgrad. Sammenligner vanskeligheten på oppgavene.
HoGG (Homework Generation Grading) Project (Morris)	Benyttes av studenter	Måler korrekthet ved hjelp av enhetstesting, regulære uttrykk og Java Reflection Classes. Det blir foretatt en subjektiv vurdering dersom studenter klager på resultatene.	Måler på skala fra 1-10 for hver oppgave, lager en total score ved å beregne gjennomsnittet på oppgavene. Alle oppgavene teller like mye. Vektor av bits der hvert element representerer success(1) failure(0). Krav at oppgaven MÅ kompilere.

Vedlegg 2: MySQL og Hibernate Installation Guide

Necessary files

Before starting the installation process, you need to download the files shown in table 1.

Name	Description	Link
MySQL Server 5.0	Open Source database server	http://dev.mysql.com/downloads/mysql/5.0.html [23.08.07]
Hibernate Core 3.2.5.ga	Core functionality for Hibernate	http://www.hibernate.org/344.html [23.08.07]
MySQL Connector/J 5.0	Java driver that converts JDBC calls to the network protocol used by MySql	http://dev.mysql.com/downloads/connector/j/5.0.html [23.08.07]

Table 1: Necessary files

In advance, you must have Eclipse 3.2 installed on your machine. You must also have the JCAT project properly installed.

MySQL Server

Before installing MySQL Server, the port **3306** must be opened, or else MySQL will be blocked by the firewall on your machine.

- For Windows XP: Open **Control panel** -> **Windows Firewall** -> **Exceptions** -> **Add Port**. You should now see a window like the one in figure 1. (You can call the new exception what you like, but it might be a good idea to give it a descriptive name, e.g. "MySql_TCP"). Enter a name and the port number **3306.** Choose "**TCP**" and press "**OK**".

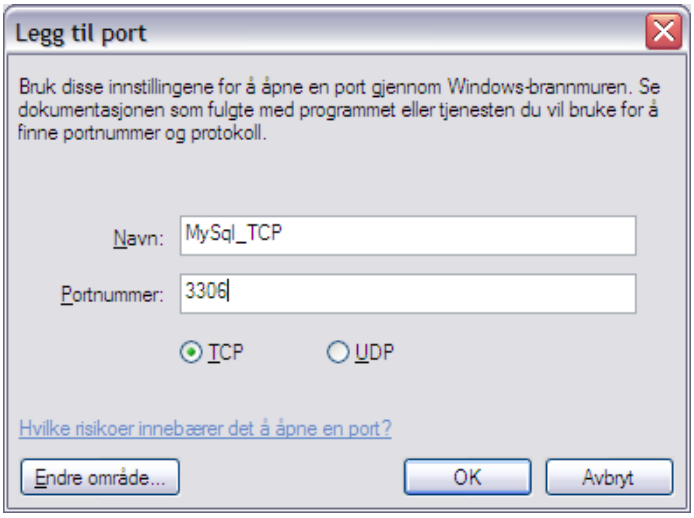


Figure 1: Opening port 3306

Unpack and install the files. After the install process is completed, the **MySQL Instance Config** tool will run automatically. (If it does not start, you can find it in the bin directory under the MySQL Server program files).

Choose "**Standard Configuration**". Mark the two boxes "**Install as Windows Service**" and "**Add bin path**" and press "**Next**". You are then asked to choose a root password. Enter a password of your own choosing, and skip the other entries. Click next until the configuration process is finished.

Open the MySQL Command Line Client. Write the password you chose under the instance configuration of MySQL. After you are logged in (you should get a welcoming message), write the commands from figure 2:

```
CREATE DATABASE jcat_db;

USE jcat_db;

GRANT ALL ON * TO jcat_user IDENTIFIED BY 'jcat';
```

Figure 2: Script for creating database

The database should now be correctly configured.

Hibernate Core

- Unpack the Hibernate Core files.
- Find the directory called "**lib**" and copy its content (the readme files are not necessary) to the lib directory of the JCAT project.
- Locate the file "**hibernate3.jar**" on the top level of the distribution (where you found the "**lib**" directory), and copy also this file to the "**lib**" directory of the JCAT project.

Connector J

- Open the file "**mysql-connector-java-5.0.7**".
- Copy the file "**mysql-connector-java-5.0.7-bin.jar**" and place it in the root directory of the JCAT project.

Add the external archives to Eclipse

- Start Eclipse. Right-click on the JCAT project, choose "**Build Path**" and "**Add External Archives**"
- Locate "**hibernate3.jar**", "**mysql-connector-java-5.0.7-bin.jar**" and all the other files you just added to the JCAT "**lib**" directory and press "**Open**"/"**Åpne**". Eclipse will now add these file as resources.

Your Package Explorer in Eclipse should now look something like the image shown in figure 3 (all the jar could not fit in the image):

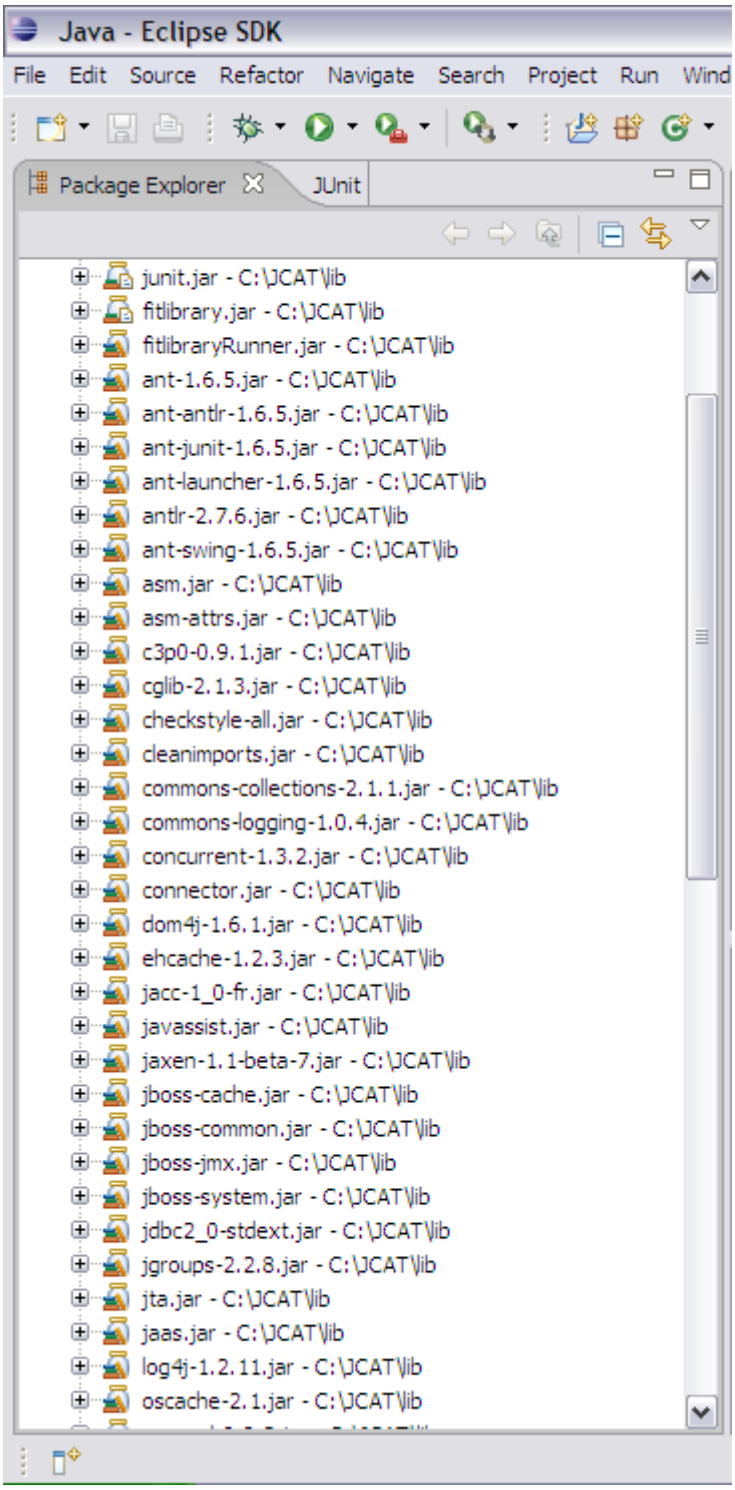


Figure 3: Eclipse Package Explorer

Generate the tables with SchemaExport

Preconditions: The previous install tasks must be completed. It is also important that the file ”**hibernate.properties**” is placed correctly (in the project’s “**scr**” directory).

- Open the jar file ”**hibernate3.jar**” in Eclipse (Package Explorer)

- Open the package ”**org.hibernate.tool.hbm2dll**”. Right-click on the class file “**SchemaExport.class**” and choose ”**Run as**” and “**Run…**”
- Choose the tab ”**Arguments**” and add complete path for the Hibernate mapping files (.hbm.xml) you wish to generate code from (e.g. **C:\JCAT\src\no\simula\itembank\Item.hbm.xml**)

For the code checked in at Milestone 2, the mapping files to add are shown in figure 4:

```
C:\JCAT\src\no\simula\jcat\itembank\Item.hbm.xml
C:\JCAT\src\no\simula\jcat\scoring\ScoringSuper.hbm.xml
C:\JCAT\src\no\simula\jcat\scoring\SubScoringSuper.hbm.xml
C:\JCAT\src\no\simula\jcat\eval\Eval.hbm.xml
```

Figure 4: Required mapping files

Figure 5 shows how the mapping files are to be added before running SchemaExport.class:

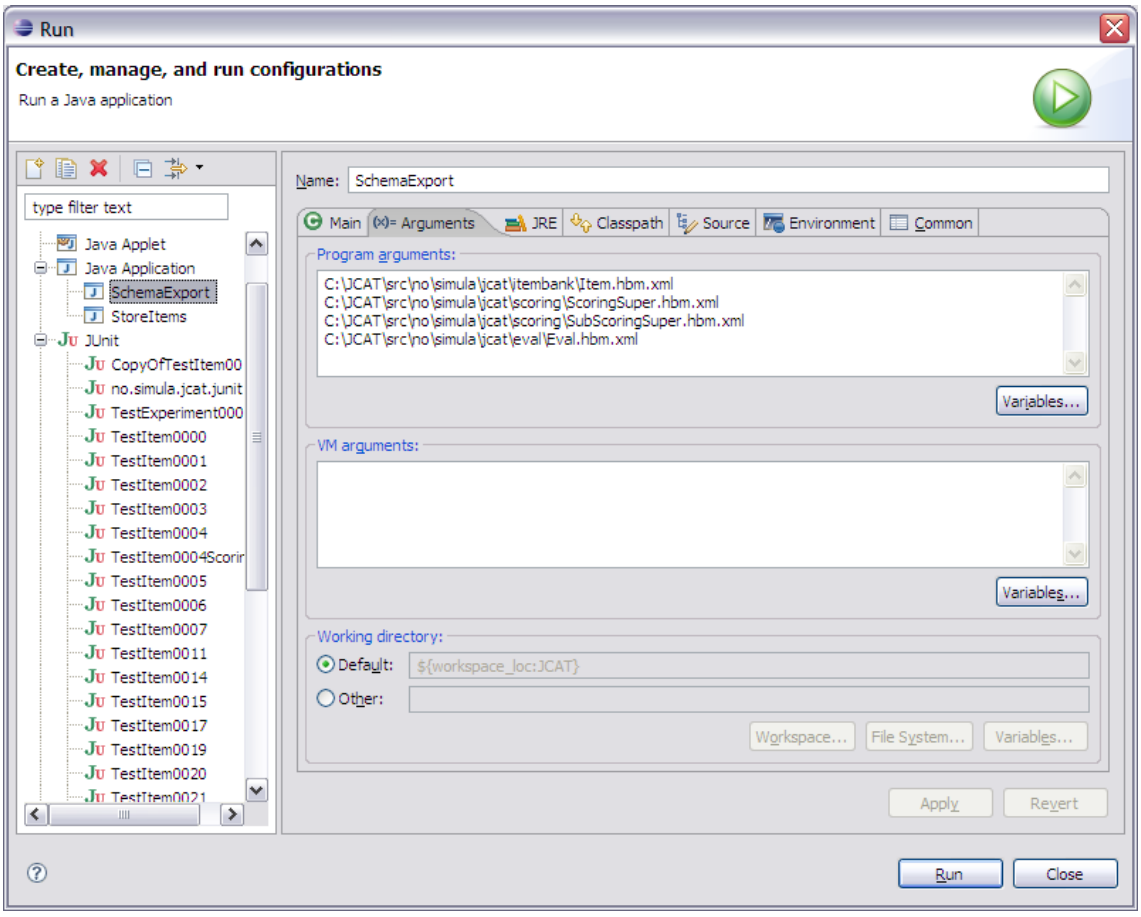


Figure 5: Running SchemaExport in Eclipse

After running SchemaExport, check the MySQL Command Line Client again. Type the command *show tables;* to check if the tables have been generated. Your result should look like the result in figure 6:

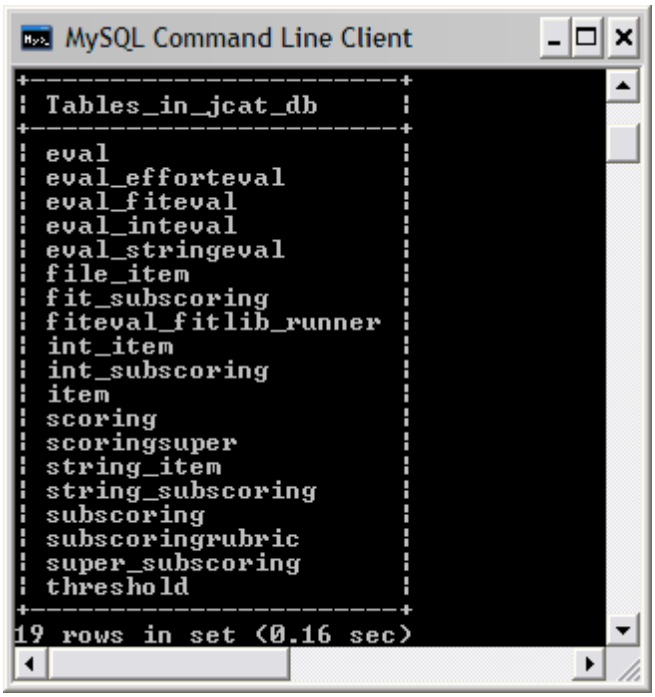


Figure 6: Tables in the database

Store Items in the database

Under the package "**no.simula.jcat.util**" is a Java class named "**StoreItems.java**". Run this class one time as a Java application to fill the database tables with the Items required for testing. Make sure you just run the class once, or else you will get duplicate Items in your database. You are now ready to run the JUnit and Fitnesse tests!

Dealing with errors and inconsistencies

If you encounter an error with the values you insert in your local database or suspect that the tables or values might be corrupt, the best thing to do might be to delete all the tables and run SchemaExport tool and "**StoreItems.java**" again to start fresh. In that case, open a MySQL Command Line Client and copy the commands from figure 7:

```
DROP DATABASE jcat_db;

CREATE DATABASE jcat_db;

USE jcat_db;

GRANT ALL

ON * TO jcat_user IDENTIFIED BY 'jcat';
```

Figure 7: Script for dropping database and regeneration of schema

This script deletes the whole schema and creates the database (and user) from scratch. If you run the SchemaExport tool and store the Items again, you will have a clean database for continued testing.