STUDENTRAPPORT

NTNU

KANDIDATNUMMER(E)/NAVN:			
10087			
DATO:	FAGKODE:	STUDIUM:	ANT SIDER:
31.10.2022	IDATT 1001	Bachelor i ingeniørfag, systemutvikling.	18

FAGLÆRER(E) :

Muhammad Ali Norozi

Surya Kathayat

TITTEL :

Applikasjon for Smarthus AS.

SAMMENDRAG:

Rapporten er skrevet på bakgrunn av en mappevurdering i Programmering 1, ved institutt for datateknologi og informatikk, ved Norges tekniske-naturvitenskapelige universitet. Mappevurderingen omhandler et prosjektarbeid hvor hensikten er utviklingen av en applikasjon.

Problemstillingen til prosjektet omhandler utviklingen av en programvare, som skal brukes for håndtering av et varehus sitt varelager. Oppdragsgiveren Smarthus AS, er et varehus som forsyner bygg-industrien med varer av treverk, og de trenger en programvare for administrering av lagerbeholdningen. Løsningen skal være bestående av et tekstbasert brukergrensesnitt og et vareregister som holder på alle varene.

Sluttresultatet er bestående av en entitet-, en register- og en enum-klasse, samt et tekstbasert brukergrensesnitt. Applikasjonen tilfredsstiller alle krav om funksjonalitet fra oppdragsgiver, samt prinsipper om god programmering.

INNHOLD

1 SAMMENDRAG

Rapporten er skrevet på bakgrunn av en mappevurdering i Programmering 1, ved institutt for datateknologi og informatikk, ved Norges tekniske-naturvitenskapelige universitet. Mappevurderingen omhandler et prosjektarbeid hvor hensikten er utviklingen av en applikasjon.

Problemstillingen til prosjektet omhandler utviklingen av en applikasjon som skal benyttes for håndtering av et varehus sitt varelager. Kunden, Smarthus AS er et varehus som forsyner bygg-industrien med varer av treverk, og trenger en programvare for administrering av lagerbeholdningen. Løsningen skal være bestående av et tekstbasert brukergrensesnitt og et register som holder på alle varene.

Under utviklingen av applikasjonen, har det vært fokus på prinsipper som modularisering, responsibility-driven design, kobling, kohesjon, dry code og refaktorering, samt lage et robust program. Når vi jobber med et stort program, er modularisering et nyttig prinsipp å ta i bruk. I tillegg ønsker vi at programmets klasser skal oppnå prinsippene om responsibility-driven design, løs kobling og høy kohesjon. Videre ønsker vi også å utvikle den mest optimale løsningen, og derfor er det viktig med refaktorering og dry code. Til slutt er det viktig at programmet er robust, slik at det ikke kan angis feil verdier som kan medføre til problemer.

Prosjektet ble innledet ved å utvikle en entitet-klasse som representerer en vare. Etter ønske fra Smarthus AS, fikk klassen ti objektvariabler som skulle representere varen. Når entitet klassen var på plass, var neste steg å lage et vareregister. Denne klassen holder på varene, samt utfører ønskede arbeidsoppgaver på dem. Det ble lagd metoder for å gjennomføre de ønskede arbeidsoppgavene, i tillegg noen ekstra metoder for å tilføye mer funksjonalitet til applikasjonen. For å teste register-klassen, ble det brukt enhetstesting ved hjelp av JUnit. Etter hvert som entitet- og register-klassen var på plass, ble det utviklet et tekstbasert brukergrensesnitt. Brukergrensesnittet er et menystyrt program, som benytter seg av klassen Scanner for å hente inn informasjon fra bruker. Underveis i prosjektet ble det gjort

endringer/forbedringer for å optimalisere applikasjonen. Det ble tilføyet hjelpemetoder, en enum-klasse og et mer brukervennlig brukergrensesnitt.

Sluttresultatet er bestående av fire klasser, som samarbeider for å gjennomføre den overordnende hensikten med applikasjonen. Den er bestående av en entitet-, register- og en enum-klasse, samt et tekstbasert brukergrensesnitt. Applikasjonen i sin helhet tilfredsstiller alle krav om funksjonalitet fra oppdragsgiver, samt prinsipper om god programmering.

Prosjektet har vært en lærerik prosess med et stort læringsutbytte i ulike sammenhenger. Blant annet i henhold til prinsippet om refaktorering, som er et viktig prinsipp å ta med seg videre i seinere programmeringsprosjekter. I tillegg har det også vært lærerikt i sammenheng med bruken av enums, stream og enhetstesting med JUnit, som ikke har blitt anvendt tidligere. Selve problemstillingen og oppgavens oppbygning har også vært lærerik, ettersom den enkelt skal illustrere det virkelige liv. Til tross for at jeg er fornøyd med eget resultat av løsning, er det alltid rom for forbedringer. Med mer tid kunne applikasjonen vært mer optimal med et grafisk brukergrensesnitt, eller med flere metoder i register-klassen slik at applikasjonen kan tilby mer funksjonalitet til brukeren.

2 TERMINOLOGI

GUI	Graphical user interface.
OOP	Object Oriented Programming.
UML	Unified Modeling Language.

3 INNLEDNING – PROBLEMSTILLING

3.1 Bakgrunn/Formål og problemstilling

Denne rapporten er et resultat av en mappevurdering i Programmering 1 ved institutt for datateknologi og informatikk, ved Norges tekniske-naturvitenskapelige universitet. Mappevurderingen omhandler utvikling av en applikasjon.

Problemstillingen omhandler utviklingen av en programvare, som skal brukes for håndtering av et varehus sitt varelager. Oppdragsgiveren Smarthus AS, er et varehus som leverer varer til bygg-industrien, og spesielt varer som laminatgulv, dører, vinduer, lister og andre typer treverk. De trenger en applikasjon for å håndtere varelageret sitt. Løsningen skal være bestående av et tekstbasert brukergrensesnitt og et register som lagrer informasjon. Brukergrensesnittet skal tilby en rekke med funksjoner som en arbeider på lagret skal anvende (se vedlegg for aktivitetsdiagram). Arbeideren skal ha mulighet til å se alle varene som er registrert, søke etter en bestemt vare etter varenummer og/eller beskrivelse, legge til ny vare, øke og redusere beholdning av en vare, slette en vare, og endre rabatt, pris og/eller beskrivelse for en vare.

I tillegg til funksjonaliteten, skal applikasjonen følge en rekke krav. For det første, skal koden følge en bestemt kodestil, og den skal verifiseres med CheckStyle. For det andre, skal klassene, metodene og variablene ha selvforklarende navn på engelsk, som tydelig gjenspeiler funksjonaliteten eller verdien deres. For det tredje, skal programmet tilfredsstille så mye som mulig av designprinsippene coupling, cohesion og responsibility driven design. Til slutt skal applikasjonen ha en brukervennlig og utfyllende brukerinteraksjon, hvor brukeren til enhver tid er fullstendig klar over hva som skjer, og hva som forventes av han/henne.

3.2 Avgrensninger

Det er ikke gitt særdeles mange avgrensninger i oppgaven, utenom hvordan en vare skal bli registrert. Objektvariablene har fått gitt beskrivelser som må følges, som for eksempel at pris og varekategori skal være et heltall, og at varenummer skal være bestående av tall og tekst.

3.3 Begreper/Ordliste

Begrep (Norsk)	**Begrep (Engelsk)**	**Betyding/beskrivelse**
Vare	Item	Et produkt med gitte beskrivelser som varehuset håndterer inn og ut av lager.
Vareregister	Item register	Et register som holder på informasjon om selve varene, samt utfører diverse justeringer på dem.
Brukergrensesnitt	User interface	Applikasjonens interaksjon med bruker.

3.4 Rapportens oppbygning

Rapporten starter med et sammendrag, som kort og enkelt tar for seg de viktigste elementene ved prosjektet. Deretter kommer det en terminologiliste for å oppklare ukjente begreper og symboler for leseren. I det påfølgende kapitlet etter terminologilisten, presenteres rapportens formål og bakgrunn, og videre problemstillingen og kravene til prosjektet som rapporten tar for seg. Like etter blir vi presentert for avgrensingene i oppgaven, og en ordliste med begreper på både norsk og engelsk, samt deres definisjon/beskrivelse. Når alt det er presentert, kommer rapportens bakgrunn og teoretiske grunnlag. Her presenteres teorier og prinsipper brukt under utviklingen av programmet. Deretter starter rapportens hoveddel med kapitlet om metode og design, hvor framgangsmåten av prosjektet blir presentert. Like etter kommer rapportens største del, nemlig selve resultatet. Her presenteres det ferdige resultatet med forklaringer for ulike valg tatt under utviklingen av applikasjonen. Når resultatet er etablert, kommer kapitlet om drøfting. I dette kapitlet blir resultat av oppgaven vurdert opp mot prosjektets bakgrunn og teoretiske grunnlag. Deretter kommer en kort konklusjon med en rekke erfaringer rundt prosjektet. Til slutt kommer referansene brukt under prosjektet og eventuelle vedlegg.

4 BAKGRUNN - TEORETISK GRUNNLAG

Under utviklingen av applikasjonen, har det vært fokus på å utvikle programmet etter gode designprinsipper og kvalitetskriterier. Det har spesielt vært fokus på prinsipper som modularisering, responsibility-driven design, kobling, kohesjon, dry code og refaktorering, samt lage et robust program.

Når vi jobber med å utvikle en programvare, er modularisering et nyttig verktøy å ta i bruk. Modularisering går ut på å dele opp et stort problem i mindre og enklere ledd, slik at det blir enklere å fokusere på hver del for seg selv (Jee, 2021). Leddene kan for eksempel være ulike klasser, som jobber sammen mot et felles mål. Dette er en veldig vanlig teknikk når man jobber med OOP, men nøkkelen til god OOP, er å benytte seg av prinsippet om responsibility-driven design. Prinsippet omhandler tildelingen av veldefinerte ansvarsområder til ulike objekter, som sammen utformer applikasjonsarkitekturen (Wirfs-Brock og Wilkerson, 1989). Dette prinsippet kan brukes til å bestemme hvilke klasser som skal holde på de ulike dataene, og metodene for applikasjonens funksjonalitet.

Når vi skriver et program med flere ledd som skal samarbeide, ønsker vi i følge læreboka løs kobling (Horstmann, 2016, s. 134). Altså vi ønsker lite avhengighet mellom klassen, slik at dersom det skjer endringer i en komponent, vil ikke alle de andre komponentene endres. I likhet med kobling, er kohesjon et viktig prinsipp under utviklingen av en applikasjonen. Det oppnås høy kohesjon når en enhet (klasse, objekt eller metode) kun har en spesifikk arbeidsoppgave/ansvar (*single responsibility principle*) og et logisk navn. Med høy kohesjon blir klasser og metoder lettere å vedlikeholde og mer gjenbrukbare (Cohesion in Java, 2022).

Det er en selvfølge å ta i betraktning at vi ønsker en mest mulig optimal applikasjon, og derfor er det viktig med refaktorering. Intensjonen bak refaktorering i utviklingen av software, er å forbedre programmet uten å endre selve funksjonaliteten (Be A Better Dev, 2020). I tillegg ønsker vi dry code, altså vi ønsker ikke å repetere samme kode flere steder (Muldrow, 2020). Med prinsippet om dry code, blir programmet kortere og lettere å vedlikeholde. Til slutt er det viktig å være bevist på å utvikle et robust program, slik at det ikke kan angis feil verdier som kan medføre til problemer.

5 METODE – DESIGN

Utviklingen av applikasjonen har i størst grad vært selvsetning arbeid, med unntak av tilbakemeldinger fra læringsassistener. Prosjektet startet med å analysere problemstillingen, med prinsippene fra kapitlet om det teoretiske grunnlaget i bakhodet. Etter å ha funnet gode kandidater for klasser som skal representere applikasjonsstrukturen, startet selv utviskingen av programmet.

Det første steget, var å utvikle en entitet klasse som skal representerer en vare (se vedlegg for klassediagram). Etter ønske fra Smarthus AS, fikk klassen ti objektvariabler. Varen skulle registreres med følgende informasjon, varenummer, beskrivelse, pris, merkenavn, vekt, lengde, høyde, farge, antall på lager og kategori. Deretter ble det laget en konstruktør for objektet av klassen med de nevnte variablene. I tillegg ble det laget en dyp kopi konstruktør, dersom det skulle være nødvending å returnere en kopi av objektet seinere i utviklingen av programmet. Ettersom alle objektvariablene settes som private, og er dermed ikke dirkete tilgengelige utenfor klassen, lages det aksesor-metoder for alle variablene og mutator-metoder for noen av variablene. Videre overstyres den innbygde toString metoden, og blir omskrevet til å vise varenes informasjon. Helt til slutt overstyres også de innebygde metodene for equals og hashCode, slik at vi kan sammenligne objekter etter gitte betingelser seinere i utviklingen av applikasjonen. For å teste entitet klassen, ble det lagd et enkelt testprogram med klassemetoden *public static void main(String[] args){}*. Etter å ha lagd entitet-klassen, ble det gikk tilbakemelding fra læringsassistenter for kvalitetssikring og veiledning av kode.

Når entitet klassen var på plass, var neste steg å lage et vareregister (se vedlegg for klassediagram). Denne klassen holder på varene, samt utføre ønskede arbeidsoppgaver på dem. For å holde på varene brukes det en ArrayList. Dette valget ble tatt på bakgrunn av at ArrayList er indeksbasert, og dermed lettere å sortere. Et varehus kan ønske å ha varene sine organisert etter kategori, slik at de får mer struktur på lager. Dersom varehuset ikke ønsker varene sine sortert, hadde HashMap vært et enklere valg. Ved å bruke HashMap slipper man å bruke løkker for å hente objekter, og kan i stedet referere til nøkler. Nøklene i dette tilfellet kunne vært varenumrene, og verdiene kunne vært selve objektene (varene). Det samme gjelder for HashSet, som heller ikke er indeksbasert, og vanskelig å sortere. Videre ble det

lagd metoder for å gjennomføre de ønskede arbeidsoppgavene, i tillegg til noen ekstra metoder for å tilføyet mer funksjonalitet til applikasjonen. Det ble tilføyet en metode for å tilbakestille prisen til den originale prisen før rabatt, en metode for å hente alle varer med samme kategori, og en metode for å sjekke hvilke varer som trenger påfyll på lager. For å teste om metodene gjorde det de skulle, og ikke gjorde det de skulle ved feil verdier, ble det brukt enhetstesting ved hjelp av JUnit lært fra nettet (Coding with John, 2022). I liket med utviklingen av entitet-klassen, ble det også gitt tilbakemelding fra læringsassistenter når førsteutkastet av register-klassen var ferdig.

Etter hvert som entitet- og register klassen tok form, var det på tide å utvikle et tekstbasert brukergrensesnittet (se vedlegg for aktivitetsdiagram). Brukergrensesnittet til applikasjonen er et menystyrt program, som benytter seg av klassen Scanner for å hente inn informasjon fra bruker. Ettersom Scanner klassen ikke er helt ideal når du veksler mellom å hente inn ulike datatyper, hentes alle verdier inn som String. Deretter konverteres verdiene over til ønskede datatyper ved hjelp av metodene parseInt og parseDouble. For at main metoden skal være kort og enkel å lese, er brukergrensesnittets funksjonalitet delt opp i flere ulike metoder. Oppdeling av brukergrensesnittet, gjør også at klassen blir lettere å vedlikeholde og mer oversiktlig. For at programmet skal være et menystyrt program, brukes det en while-løkke. Valget av while-løkke i stedet for for-løkke, skyldes av at for-løkker kjøres et forhåndsdefinert antall ganger, mens while-løkker kjører så lenge betingelsen er oppfylt. For at brukeren skal kunne få informasjon de ønsker, brukes det en switch-setning hvor hver case representerer en metode som gjennomfører det ønskede arbeidet. For at switch-setningen skal være enklere å lese, er alle tallene for hver case byttet ut med konstanter. Valget av switch i stedet for if-setninger, skyldes av at Switch gir høyre ytelse i programmet, samt er lettere å lese og vedlikeholde. For å kvalitetssikre brukergrensesnittet, og for å teste for brukervennlighet. Ble det gjennomført en brukertest ved hjelp av tre studenter ved NTNU, som ikke hadde programmeringserfaring. Dette var på bakgrunn av at applikasjonen skal brukes av en person uten kjennskap til hvordan programmet er utviklet, og dermed er det viktig å hente inn informasjon om hvordan de hadde anvendt applikasjonen.

Verktøy brukt for å utvikle applikasjonen, har vært Intellij og CheckStyle. Valget av Intellij i stedet for andre IDE som Visual Studio Code, skyldes av at Intellij er mer optimal for Java

programmering. I tillegg til å gi tips til optimalisering av koden, inneholder Intellij funksjoner som gjør det å kode en fungerende applikasjon lettere. Blant annet er det lettere å enhetsteste med JUnit på Intellij, enn på Visual Studio Code. CheckStyle er brukt for å sjekke om programmet følger en god kodestil, og om den er veldokumentert ved hjelp av JavaDoc.

6 RESULTATER

Datatypene valgt for objektvariablene i entitet-klassen beskrevet i kapitlet over, er på bakgrunn av hva slags type informasjon hver enkelt av dem skal holde. Variabelen varenummer for eksempel, skal holde bokstaver og tall, og er dermed satt som en String. Ettersom en String kan både holde bokstaver og tall (som String), i motsetning til int og char som kun kan holde en av delene. I likhet med varenummer, ble variabelen for beskrivelse, merkenavn og farge også satt som String, ettersom de skal holde på ord eller setninger. Vekt, lengde og høyde fikk datatypen double, fordi de kan oppgis i desimaltall. Variabelen for antall varer på lager derimot, fikk datatypen int, fordi logisk sett burde antallet oppgis i heltall. I tillegg ble kategori og pris også satt som int, fordi oppgaven spesifiserer at kategori og pris skal representeres som et heltall.

I likhet med å velge datatypene for objektvariablene i entitet-klassen, måtte det også bli tatt et i valg om det skulle være mulig å endre verdiene deres. De fleste av objektvariablene har ikke mutator-metoder og er satt som final (uforanderlige), ettersom det ikke er logisk for et varehus å endre de fleste av dem. Hvis vi tar merkenavn som et eksempel, er det ikke logisk at en vare endrer merkenavn. Dermed er den satt som uforanderlig. Feltene som er logisk å endre derimot, som beskrivelse, pris og antall på lager, har fått mutator-metoder. Dette skyldes av at et varehus kan finne på å gi en ny beskrivelse av en vare, eventuelt senke eller høyne prisen, og redusere antall på lager når en vare blir solgt, eller øke hvis de får flere inn på lager.

I henhold til prinsippet om refaktorering nevnt i det teoretiske grunnlaget, ble det underveis i prosjektet gjort endringer/forbedringer for å optimalisere applikasjonen (se vedlegg for klassediagram). Den første endringen var å implementere hjelpemetoder i register-klassen.

Ved hjelp av hjelpemetodene ble hovedmetodene kortere (mindre repeterende kode), lettere å lese og fikk kun en spesifikk oppgave. Det ble implementert tre hjelpemetoder, en for å finne en vare etter varenummer, en for å sortere varer etter kategorinavn, og en for å fjerne en vare fra HashMap-en med prisene for varene før rabatt. Videre ble det implementert en egen enum-klasse for å holde på de ulike kategoriene av varer, ettersom kategoriene er et sett med forhåndsbestemte verdier. På grunn av at varekategorien bestemmes av et heltall, ble hver enum konstant assosiert med et heltall og et kategorinavn. For at kategorinavnet skal kunne benyttes i entitet klassen, ble det lagd en metode i enum-klassen som henter kategorinavnet ved hjelp av kategorinummeret. Dersom varehuset ønsker å legge til en ny kategori av varer, må de lage en ny konstant i Enum klassen, og i tillegg endre det tillatte intervallet for kategorinummer i konstruktøren. Ettersom applikasjonen kun opererer med ArrayList for objekter av entitet-klassen, og ikke HashMap eller HashSet, ble den overstyrte metoden for hashCode fjernet fra entitet-klassen. I likhet med entitet- og register-klassen, ble det gjort endringer i brukergrensesnittet. Etter å ha gjennomført brukertesten nevnt tidligere i teksten, ble det implementert noen små endringer for å gjøre brukergrensesnittet mer brukervennlig. Blant annet var ikke alle meldingene ut til bruker selvforklarende, og måtte spesifiseres enda mer. I tillegg måtte feil formatering av informasjon inn fra bruker håndteres.

Den ferdigstilte løsningen for applikasjonen, med kravspesifikasjonene og begreps-kapittelet tatt i betraktning, er bestående av fire klasser med hvert sitt spesifikke ansvarsområde (se vedlegg for sekvensdiagram). Den overordnende hensikten med applikasjonen er delt opp i flere ledd, som sammen jobber mot et felles mål. Den første klassen er en entitet-klasse kalt «Item», og har ansvar for å representere en vare. Denne klassen holder på alle data som en vare skal identifiseres med, etter krav fra oppdragsgiver. Den andre klassen er en enum-klasse, som har ansvar for å holde et sett med kategorier for entitet-klassen. Den tredje klassen er et vareregister kalt «ItemRegister», som har ansvar for å lagre varene og utføre diverse arbeidsoppgaver på dem. Register-klassen tilbyr metoder som oppfyller alle krav om funksjonalitet fra Smarthus AS. Den fjerde klassen er et tekstbasert brukergrensesnitt kalt «Client», og brukes for å kommunisere med brukeren. Brukergrensesnittet benytter seg av metodene fra register-klassen, og ved hjelp fra input fra bruker utfører arbeidsoppgaver i henhold til krav fra oppdragsgiver. Navngivingene av klassene er på bakgrunn på ansvarsområdene til hver enkelt klasse. Entitet klassen er kalt «Item» og ikke for eksempel

«Person», fordi klassens ansvarsområde er å representere en vare. Applikasjonen i sin helhet tilfredsstiller alle krav om funksjonalitet fra oppdragsgiver.

For å sikre at entitet-klassen er implementert som en robust klasse, blir det flere steder kastet IllegalArgumentException for ugyldige verdier. Dette er både gjort i konstruktøren og i mutator-metoder, for objektvariablene som vil medføre til problemer for varehuset dersom det angis ugyldig verdier. Blant annet, blir det kastet IllegalArgumentException hvis prisen er negativ. Det er ikke logisk at varehuset skal gi kunden penger, når det er de som skal selge en vare. Et annet eksempel er for variabelen som holder antallet på lager. Det er ikke logisk at antallet skal være lavere enn null, og derfor kastes det en IllegalArgumentException hvis antallet er negativt.

I likhet med entitet-klassen, ønsker vi ikke å lage en register-klasse som kan ta inn ugyldige verdier som kan skape problemer for varehuset. Derfor benyttes det også i denne klassen IllegalArgumentException, men også returnering av boolean og null verdier. Ettersom bruken av mange exceptions er kostbart for programmets ytelse, blir det kun kastet exceptions i metoder hvor feil verdi fra bruker kan føre til kritiske feil, mens det blir returnert boolean eller null verdier for mindre kritiske og gjennomgående feil. For eksempel vil det være naturlig å kaste en exception i metoden for å registrere nye vare, dersom man prøver å registrere en vare som allerede eksiterer. I metoden for å fjerne en vare derimot, er det ikke nødvendig med en exception. I stedet kan man returnert en boolean verdi, som forteller bruker om varen de prøver å fjerne faktisk eksisterer og om arbeidet ble gjennomført. En rekke av metodene benytter seg av begge deler, fordi de kan gi to forskjellige feilmeldinger. Dersom vi skulle ha byttet ut exception med returnering av en boolean verdi, ville det vært vanskelig å gi to ulike konkrete feilmeldinger. For at bruker skal kunne tydelig forstå hvorfor det ønskede arbeidet ikke ble gjennomført, må begge deler benyttes.

Feilmeldingene fra entitet- og register-klassen må til slutt håndteres i brukergrensesnittet. For håndtering av feil verdier/informasjon i brukergrensesnittet, brukes det Try-catch for IllegalArgumentException og if-else-setninger for boolean og null verdier. Try-catch er nødvendig for å kunne teste kodene for feil uten å stoppe hele programmet. Dersom en exception forekommer, vil det bli håndtert ved at bruker mottar en feilmelding som spesifikt

forteller hvor feilen ligger. Deretter kan bruker fortsette å bruke programmet, og eventuelt prøve igjen med riktige verdier. Det samme gjelder for bruken av if-else-setninger, som forteller bruker om det de skrev inn var gyldig eller ikke. I tillegg til å håndtere feil fra entitet- og register-klassen, håndteres også NumberFormatException, som forekommer ved feil format av verdi. Dersom bruker taster inn verdier som ikke samsvarer med mønstret for den forventede typen, vil de ved hjelp av en catch få beskjed om at de har puttet in ikke er en gyldige verdi. Deretter vil programmet fortsette, og de kan prøve igjen med riktig format. For å være på den sikre siden, og sørge for at programmet aldri krasjer, inneholder også brukergrensesnittet en catch som håndterer generelt exceptions. Dermed vil programmet kunne fortsette dersom det oppstår et uforventet problem, som ikke ble tatt i betraktning under utviklingen av applikasjonen.

7 DRØFTING

I kapitlet om det teoretiske grunnlaget fra tidligere i teksten, ble det nevnt at modularisering og responsibility-driven design var to fokusområder under utviklingen av applikasjonen. Dersom vi ser på sluttresultatet av løsningen, kan vi se at den når de prinsippene ved å dele opp applikasjonens funksjonalitet i flere klasser med veldefinerte ansvarsområder. I tillegg kan vi også se at prinsippet om modularisering, også har blitt anvendt innad i klassene. Blant annet i brukergrensesnittet, hvor funksjonaliteten til klassen har blitt delt opp i flere selvstendige metoder, med hver sin hensikt.

Når vi benytter oss av modularisering, er det viktig å være bevist på koblingen mellom de ulike leddene. Som tidligere nevnt, ønsker vi løs kobling mellom klassene. Det er ikke mulig å oppnå total selvstendighet mellom klassene, men for å oppnå mest mulig løs kobling mellom entitet- og register-klassen, opprettes det et objekt av entitet-klassen i registeret. Ved hjelp av objektet, hentes og endres private objektvariabler ved hjelp av aksesor- og mutator-metoder. Register-klassen har altså ikke direkte tilgang til entitet klassen, og må benytte seg av et objekt av klassen. Dette gir mindre avhengighet mellom klassene, og dermed løs kobling. I tillegg benyttes det dyp kopiering hver gang en offentlig metode returnerer en referanse datatype, og dermed minimerer avhengighet mellom klassene. Dersom referanse

datatypen hadde blitt returnert uten dyp kopiering, vil det vært mulig å endre den utenfor klassen, noe som skaper en sikkerhets trussel og dårlig kode.

I tillegg til at den ferdigstilte applikasjonen har relativt nådd løs kobling, har den også nådd høy kohesjon. Alle klassene har en tydelig spesifikk hensikt, samtidig som de inneholder metoder og variabler relatert til hensikten. Metodene innad i klassene bidrar til at klassene får høyre kohesjon, ved at de har tydelige logiske navn og kun gjør en oppgave. Spesielt er det hjelpemetodene som ble implementert på bakgrunn av prinsippet om refaktorering, som bidrar til at metodene oppnår prinsippet om *single responsibility* og dry code, noe som bidrar til høy kohesjon. Som tidligere nevnt i sammenheng med kategorier, er det relativ lett å implementere en ny kategori uten å skrive om mye kode. Dette viser til god kodedesign, noe som igjen forsterker prinsippet om høy kohesjon.

I likhet med å ha nådd prinsippene over, er applikasjonen også implementert som et robust program. Det er ikke mulig å angi feil verdier/informasjon som kan medføre til problemer, noe som øker kvaliteten til applikasjonen. I tillegg gjør den programmet mer brukervennlig, ettersom den hjelper brukeren med å kun putte inn verdier som varehuset ønsker.

Sluttresultatet tilfredsstiller alle krav om funksjonalitet fra oppdragsgiver, samt alle krav til selve koden. I henhold til avsnittene over, når løsningen også kriteriene for noen av de viktigste prinsippene for god programmering. Ettersom selve utviklingen av en slik applikasjon ikke var ukjent, siden vi har jobbet med lignende programmer tidligere fra øvingsoppgavene i faget, var det ikke særlig store utfordringer/problemer underveis i utviklingen. Likevel kan teorien bak å lære seg streams, trekkes fram som det mest utfordrende. Dette ble håndtert og anvendt under utviklingen av applikasjonen ved hjelp av opplæringsvideoer på nettet (Amigoscode, 2020). Med alt tatt i betraktning, tilfredsstiller resultatet en optimal applikasjon for varehuset.

8 KONKLUSJON - ERFARING

Prosjektet har vært en lærerik prosess med et stort læringsutbytte i ulike sammenhenger. Blant annet i henhold til prinsippet om refaktorering. Det å endre kode underveis i prosjektet for å

optimalisere den, er en god kilde for egen læring. Hvis vi bruker eksemplet med implementering av hjelpemetodene fra tidligere, er et dette en nyttig erfaring å ta med seg videre i seinere programmeringsprosjekter. I tillegg har det også vært lærerikt i sammenheng med bruken av enums, stream og enhetstesting med JUnit, ettersom at det ikke har vært anvendt i tidligere programmeringsoppgaver. Selve problemstillingen og oppgavens oppbygning har også vært lærerik, ettersom den enkelt skal illustrere det virkelige liv, hvor du kun får en rekke med krav som du selv må finne en løsning for. Dette krever kompetanse, og at utvikleren er løsningsorientert og kreativ.

Til tross for at jeg er fornøyd med sluttproduktet, er det alltid rom for forbedringer. Det er ikke alltid like lett å peke på egne forbedringspotensialer, men applikasjonen kunne kanskje hatt et enda mer brukervennlig brukergrensnitt eller et register med flere funksjoner. Den kunne for eksempel tilbyd endring av språk i brukergrensesnittet, eller flere metoder i register klassen. Dersom det hadde vært mer tid, hadde applikasjonen også vært et hakk bedre med grafisk brukergrensesnitt (GUI). JOptionPane kunne vært en løsning for å bedre illustrere applikasjonen for brukeren. I tillegg vil denne vært et hakk bedre med håndtering av skrivefeil, hvor bruker får et valg om å velge det nærmeste til det han/henne prøvde å skrive.

9 REFERANSER

Amigoscode (2020) *Java Streams Tutorial*. Tilgjengelig fra: https://www.youtube.com/watch?v=Q93JsQ8vcwY&t=589s (Hentet 21.11.22).

Be A Better Dev (2020) *What is Refactoring? (as a software developer).* Tilgjengelig fra: https://www.youtube.com/watch?v=DQJGRV9np40 (Hentet: 05.12.22).

Coding with John (2022) *Java Unit Testing with JUnit – Tutorial – How to Create And Use Unit Tests*. Tilgjengelig fra: https://www.youtube.com/watch?v=vZm0lHciFsQ (Hentet: 03.12.22).

Cohesion in Java. (2022) *GeeksforGeeks*. Tilgjengelig fra: https://www.geeksforgeeks.org/cohesion-in-java/ (Hentet: 05.12.22).

Horstmann, C.S. (2016) *Core Java.* 10. utg. New York: Prentice Hall.

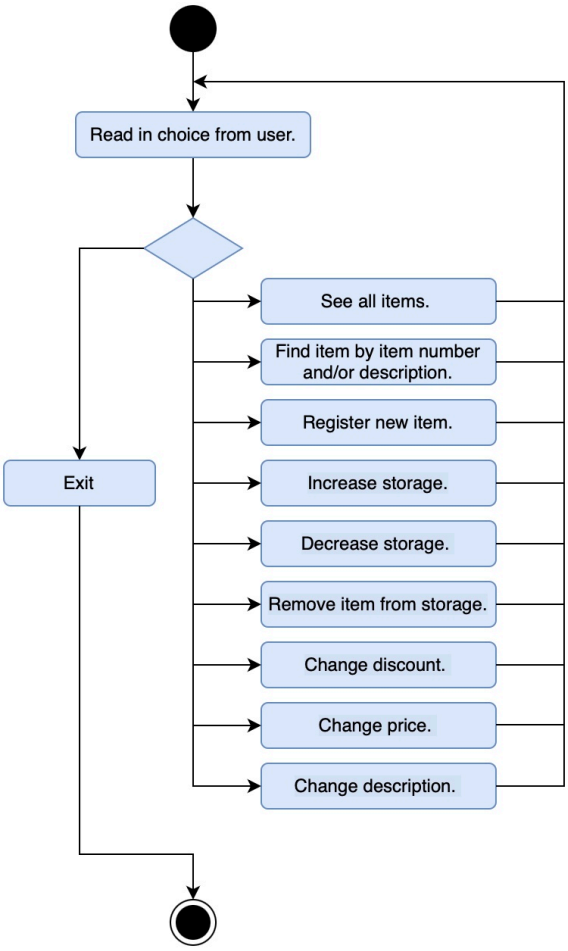
Jee, C. (2021) Modularization in Software Engineering, *Medium*, 10. oktober. Tilgjngelig fra: https://medium.com/@caitlinjeespn/modularization-in-software-engineering-1af52807ceed (Hentet: 01.12.22).

Muldrow, L. (2020) *What is dry development?* Tilgjengelig fra: https://www.digitalocean.com/community/tutorials/what-is-dry-development (Hentet: 08.12.22).

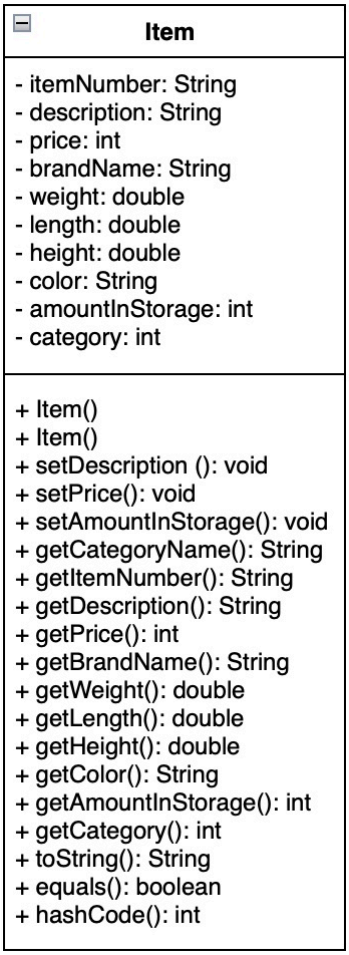
Wirfs-Brock, R og Wilkerson, B (1989) *Object-Oriented Design: A Responsibility-Driven Approach*. Tilgjengelig fra: http://cv.znu.ac.ir/afsharchim/lectures/Responsibility-Driven%20Design.pdf (Hentet: 08.12.22).

10 VEDLEGG

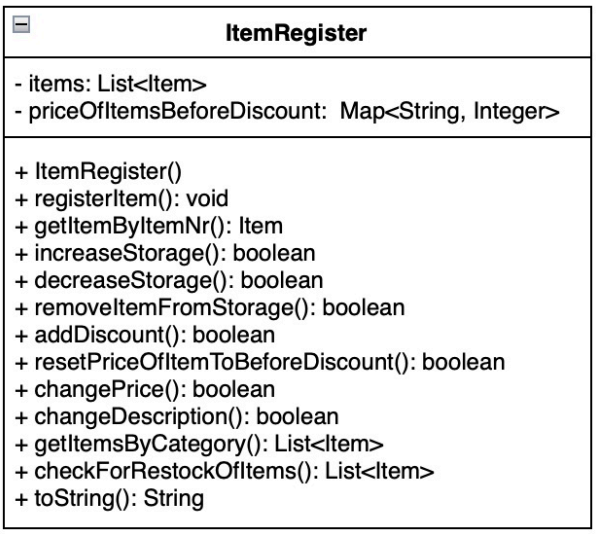
Vedlegg 1: UML aktivitetsdiagram for funksjonalitet for brukergrensesnitt fra oppdragsgiver.



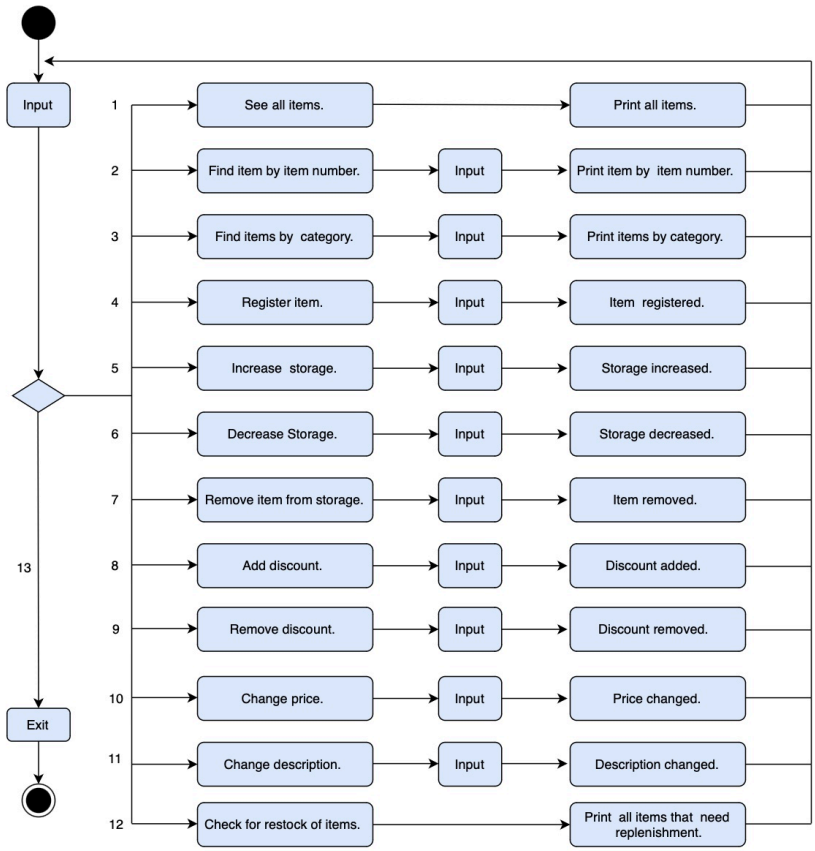
Vedlegg 2: UML klassediagram for førsteutkast av entitet-klasse.



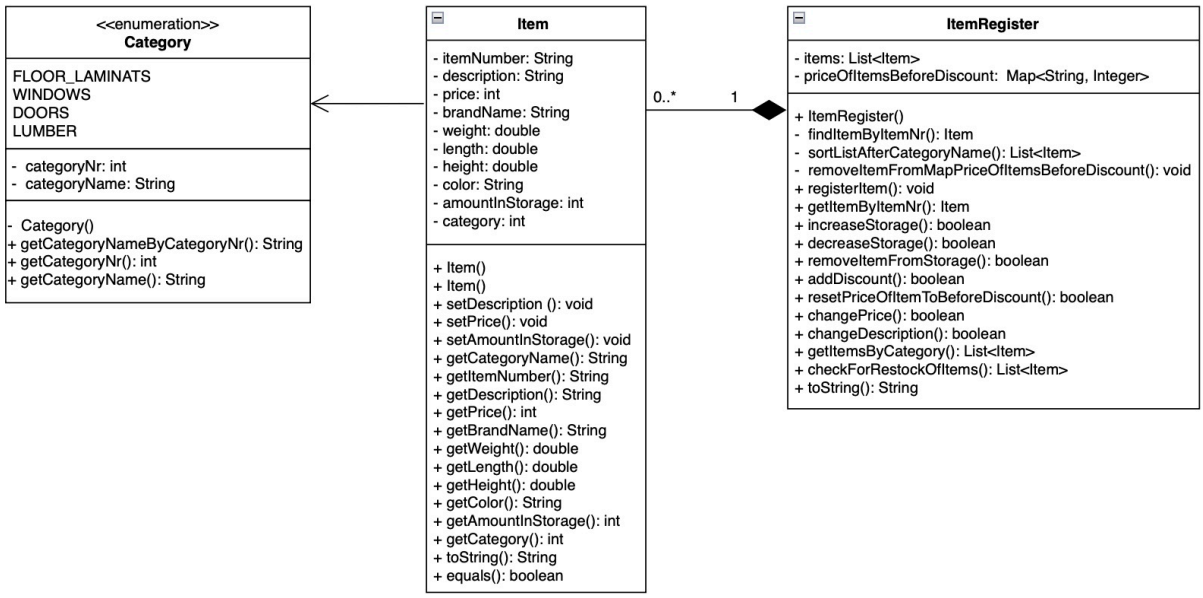
Vedlegg 3: UML klassediagram for register-klasse før refaktorering.



Vedlegg 4: UML aktivitetsdiagram for brukergrensesnittet.

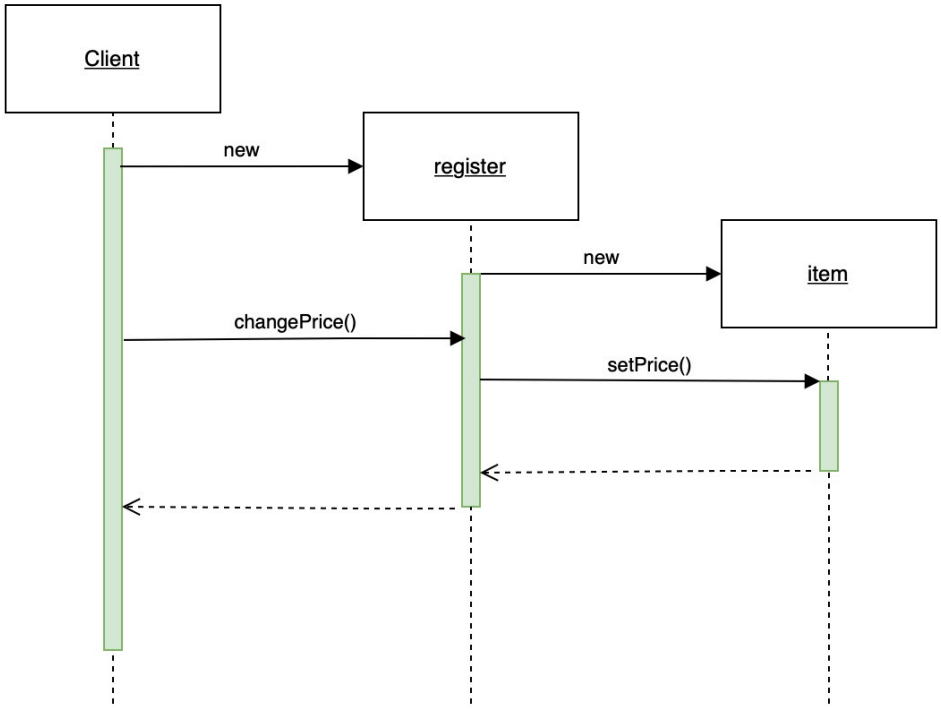


Vedlegg 5: UML klassediagram for sluttresultatet av entitet-, register- og enum-klasse.



Kommentar: Det er komposisjon mellom Item- og ItemRegister-klassen, ettersom den ene ikke kan eksistere uten den andre. I tillegg kan kun en vare tilhøre et vareregister, mens et vareregister kan inneholde flere varer. Hvor mange vareregister en vare kan tilhøre til, kan variere på bakgrunn av hvordan varehuset ønsker å ha lagret sitt organisert. Det mest logiske er at en vare kun kan tilhøre et vareregister.

Vedlegg 6: UML sekvensdiagram.



Kommentar: Viser interaksjonene mellom klassene. Programmet innledes ved at brukergrensesnittet oppretter et objekt av register-klassen, og videre oppretter register-klassen et objekt av entitet-klassen. I sekvensdiagrammet illustreres hvordan interaksjonene mellom klassen hadde foregått, dersom brukeren hadde ønsket å endre prisen på en vare.