STUDENTRAPPORT

NTNU

KANDIDATNUMMER(E)/NAVN:			
10120			
DATO:	FAGKODE:	STUDIUM:	ANT SIDER/BILAG:
13.12.22	IDATT1001	Bachelor i ingeniørfag, data - Trondheim	13

FAGLÆRER(E) :

Muhammad Ali Norozi
Arne Gerhard Styve
Kiran Bylappa Raja
Surya Bahadur Kathayat

TITTEL :

Warehouse management system for Smarhus AS

SAMMENDRAG:

Dette prosjektet innebar å lage et "Warehouse management system" for det fiktive selskapet Smarthus AS. Dette er et program som lagrer og endrer på informasjon om et varehus og produktene det har på lager. Programmet skal ha et brukergrensesnitt som ansatte kan bruke for å utføre endringer på systemet. Denne rapporten dokumenterer gjennomføringen av prosjektet.

INNHOLD

1 SAMMENDRAG

I denne rapporten blir prosessen og resultatene av gjennomføringen av prosjektet som utgjør mappevurderingen til Programmering 1. Prosjektet bestod av å lage et program for å administrere et varehus for det fiktive selskapet Smarthus As. Fra et akademisk standpunkt, så er hensikten med oppgaven at studenten skal produsere et program som følger teoretiske prinsipper om robusthet til kode og gjennomføring av et programmeringsprosjekt. Denne rapporten dokumenterer hvilke valg som har blitt gjort i henhold til tolkninger, avgrensinger og utvidelse av kravspesifikasjonene som ble gitt i oppgaveteksten. Rapporten viser også fremgangsmåten for utviklingen av prosjektet, hvilke valg som ble gjort underveis og hvorfor. Den diskuterer resultatet og hvorvidt det oppfyller kravene som har blitt stilt, og gir til slutt en refleksjon om læringsutbytte av prosjektet.

2 TERMINOLOGI

IDE	Integrated development environment
VS Code	Visual Studio Code
Plugin	Tilleggsprogram som utvider funksjonaliteten til en applikasjon.
Setter	Metode som endrer på verdien til et felt.
Getter	Metode som henter verdien til et felt.
Javadoc	Standardformat for å beskrive Java-kode.
JDK	Java development kit
Exception/unntak	En feilmelding som sier noe om hva som har gått galt. Denne kan bli plukket opp av de som kaller på metoden som utløser den, og de kan bruke den for å håndtere hva som skjer i et slikt tilfelle.
Dyp-kopi	Kopi av et objekt hvor all data til det originale objektet har blitt kopiert og lagret et annet sted i minnet.
Grunn-kopi	Kopi av et objekt som peker til samme minnet som det originale. Begge objektene kan endre på det samme minnet, som ofte ikke er ønskelig.
Versjonskontroll-program	Programvare for å lagre en kodebase slik at en kan gå tilbake til tidligere punkter i utviklingen og som brukes for å lagre koden i skyen.
Kjøretid	Hvor mange operasjoner, i verste tilfellet, som trengs for å utføre en oppgave.
Enum	Kort for “enumerator”. En datatype som lar en representere forhåndsbestemte kategorier.
API	Application programming interface
Argument	Verdien som blir gitt til en metode når en kaller på den
Metode	En samling med kode i en klasse som utfører en funksjon. Vi sier at metoden blir “kalt på” når noen bruker den.
Felt	Enten en variabel eller en metode som tilhører en klasse
String/streng	Datatype som holder på tekst

3 INNLEDNING – PROBLEMSTILLING

3.1 Bakgrunn/Formål og problemstilling

Dette prosjektet innebar å lage et "Warehouse management system" for det fiktive selskapet "Smarthus AS". Dette er et program som oppbevarer og endrer på informasjon om et varehus og produktene det har på lager. Programmet skal ha et brukergrensesnitt som ansatte kan bruke for å utføre endringer på systemet. Programmet består altså av tre forskjellige deler: et element som beskriver en vare og håndterer informasjonen til varen, et element som beskriver et inventar som har en samling av varer og funksjoner for å behandle varene og brukergrensesnittet, som tillater en ansatt å bruke funksjonaliteten til inventaret.

I oppgaven ble det oppgitt at Smarthus AS ønsker at et produkt skal holde på følgende informasjon:

1. Varenummer – består av bokstaver og tall
2. Beskrivelse – en tekst som beskriver kort om varen
3. Pris – Heltall
4. Merkenavn – en tekst som inneholder merke (Hunton, Pergo, Egger osv)
5. Vekt – i kilogram, som et desimaltall
6. Lengde - som et desimaltall
7. Høyde - som et desimaltall
8. Farge – beskrevet som tekst
9. Antall på lager - antall varer på lager. Skal aldri være mindre enn 0.
10. Kategori - et tall som representerer kategori av varen. Bruk følgende: (1) Gulvlaminater, (2) Vinduer (3) Dører og (4) Trelast

Det ble også oppgitt av brukergrensesnittet skulle implementere følgende funksjonalitet:

1. Skrive ut all varer på lageret
2. Søke etter en gitt vare basert på Varenummer og/eller Beskrivelse
3. Legge en ny vare til registeret. Her skal all informasjon fra 1-10 felter (gitt over) innhentes fra bruker input.
4. Øke varebeholdningen til eksisterende vare. M.a.o. du har en vare med et gitt antall på lager (f.eks. 10 stk laminatgulv). Du mottar så en ny forsyning av laminatgulv som så skal registreres inn på lageret (f.eks. 20 stk).
5. Ta ut varer fra varebeholdningen (eksisterende vare). M.a.o. du har en vare med et gitt antall på lageret (f.eks. 20 stk laminatgulv). Du tar så ut 5 stk fra lageret.
6. Slette en vare fra varelageret (fordi den for eksempel er utgått eller ikke i produksjon lenger). M.a.o. du skal ikke lenger ha varen "Laminatgulv" i butikken din lenger. NB! Ikke det samme som å sette antall varer til 0.
7. Endre rabatt, pris og/eller varebeskrivelse for en vare

Vi stod valgfritt til å bestemme hvordan brukergrensesnittet skulle ta innputt fra brukeren. Det ble også oppgitt at vi stod fri til å legge til og endre på funksjonalitet hvis vi synes det var hensiktsmessig eller ville forbedre brukeropplevelsen.

En avgrensning som er gjort av forfatter er følgende: I beskrivelsen av hvilke funksjonalitet brukergrensesnittet skal ha, så står det på punkt 7: "Endre rabatt, pris og/eller varebeskrivelse for en vare." I kravene for vare-klassen, så er rabatt ikke tilstede som noe den skal inneholde. Å endre rabatt er derfor ikke blitt inkludert som noe en kan gjøre i brukergrensesnittet. Et annet argument for hvorfor en rabattfunksjon ikke burde implementeres, er at å gi rabatt er noe en

gjør i henhold til salg. Siden dette er et program for håndtering av et varehus, så er håndtering av rabatt noe som ikke hører hjemme i programmet.

Sentralt i problemstillingen er at vi skal følge prinsippene for utvikling og design av kode vi har lært om i løpet av semesteret. Disse blir diskutert i seksjon 4.

3.2 Avgrensninger

Ingen avgrensninger ble gitt i oppgaven.

3.3 Begreper/Ordliste

Begrep (Norsk)	**Begrep (Engelsk)**	**Betyding/beskrivelse**
Produkt	Product	Inneholder informasjon om en gitt vare.
Inventar	Inventory	Inventaret er samlingen av alle produktene som ligger på lageret, og det har ansvaret for å endre informasjonen til enkelte produkter og lagerbeholdningen som helhet.
Klient	Client	Klienten er ansvarlig for brukergrensesnittet til inventaret. Klienten viser informasjon til brukeren om hvilke valgmuligheter de har og informasjon om produktene. Den tar også innputt fra brukeren for å utføre operasjonene som inventaret tilbyr

3.4 Rapportens oppbygning

Rapporten omhandler gjennomføringen av utviklingen av et program. Gjennomføringen baserer seg på flere teorier om hvordan programvare bør designes og skrives. Disse teoriene blir lagt fram i seksjon 4, og blir så vist til videre i teksten.
I avsnitt 5 blir det øvrige designet av programmet presentert i form av en tidslinje for utviklingen. I denne tidslinjen tas det med beskrivelser av hvordan designet på ulike tidspunkt i utviklingen ser ut, og begrunnelser for hvorfor det ble gjort på den måten. I dette avsnittet blir også verktøy som ble brukt i utviklingen presentert.

I avsnitt 6 presenteres det endelige produktet, og utdypende beskrivelser om implementasjonen blir gitt. I avsnittet blir også den endelige koden sammenlignet med tidligere versjoner, og grundigere begrunnelser enn de gitt i avsnitt 5 for hvorfor endringene ble gjort blir presentert.

I avsnitt 7 diskuteres gjennomføringen av oppgaven, i hvor stor grad produktet oppfyller kravene som ble gitt, svakheter ved designet og hvorvidt resultatet og gjennomføringen stemmer overens med teorien som blir gitt i avsnitt 4.

I avsnitt 8 trekkes de viktigste punktene fra de tidligere avsnittene og erfaringer fra prosjektet sammen i en konklusjon som en kan ta med seg videre i senere prosjekter.

4 BAKGRUNN - TEORETISK GRUNNLAG

I oppgaveteksten blir det referert til flere prinsipper om programmering og kode som vi skal følge i utviklingen av prosjektet. Disse er: cohesion, coupling, modularitet, lagdelt arkitektur, responsibility driven design robust og fail-safe kode og graceful termination.

Cohesion er prinsippet om at funksjonalitet som ligner skal være samlet [3]. I objektorientert programmering, så lager en abstraksjoner som representerer deler av funksjonaliteten for programmet. Som eksempel, så bruker vi begrepene som er gitt i seksjon 3. Programmet har tre deler som kan bli delt inn i hver sin abstraksjon: varen, inventaret og klienten. Hvis en skal følge prinsippet om cohesion, så skal funksjonaliteten som er knyttet til hver av disse delene ligge i klassen som representerer konseptet.

Coupling handler om hvor tett funksjonaliteten til forskjellige deler av programmet er knyttet sammen [4]. Vi deler coupling i to versjoner: høy coupling og lav coupling. Hvis en implementasjon har høy coupling, så er funksjonaliteten til en del av programmet i stor grad avhengig til funksjonaliteten til en annen del av programmet. Dette medfører at hvis en gjør endringer i en del av programmet, så er sannsynligheten for at en må gjøre endringer i en annen stor. Dette gjør koden vanskeligere å teste, siden det er utydelig hvor enkelte prosesser skjer, og det gjør at det blir en større jobb hvis en skal endre funksjonaliteten til koden. Hvis et program har lav coupling, så er funksjonaliteten til de forskjellige delene adskilt. Dette fører til inversen av effekten av høy coupling: koden er enkligere å teste, siden det er tydelig hvor prosessene blir utført, og det gjør koden lettere å endre, siden endringer i en del av koden i stor grad ikke påvirker andre deler av koden.

Modularitet er prinsippet om at en deler opp funksjonaliteten til et program i adskilte deler som utfører en mindre del av den totale funksjonaliteten [2]. Dette kan gjøre det lettere å implementere koden, siden en deler problemet i mindre deler som en løser hver for seg.

Lagdelt arkitektur er en måte å organisere kommunikasjonen mellom ulike deler av koden [1]. Hvis en følger lagdelt arkitektur, så kommuniserer en del av programmet kun med koden som representerer laget over seg og det under seg. Dette forsikrer at en ikke får uventede konsekvenser ved at en flere deler av koden endrer på data som de begge er avhengige av.

Responsibility driven design er prinsippet om å skjule hvordan funksjonaliteten i en del av et program er gjennomført for de som bruker denne delen av koden [5]. Hver del av koden implementerer en kontrakt som de som bruker den må følge. Så lenge kontrakten er oppfylt, så vil de som bruker funksjonaliteten få resultatene slik de er beskrevet i kontrakten.

Robust, fail-safe kode og graceful termination er prinsipper om å lage kode som ikke feiler [1]. En gjør dette ved å implementere kode som forsikrer at verdier er gyldige ovenfor det de skal bruke, og som kan håndtere eventuelle feil som kan oppstå i programmet. Enkelte feil som skjer i et program kan føre til at det kræsjer, og robust kode skal forsikre at programmet håndterer disse feilene på en måte som ikke kræsjer programmet.

5 METODE – DESIGN

Det har blitt benyttet flere verktøy i sammenheng med å utvikle programmet: VS Code ble brukt som IDE, og Git ble brukt som versjonskontrollprogram. GitHub ble brukt for skylagring av koden. Bruken av Git og GitHub gjør at en kan jobbe på koden fra flere PCer, og det sikrer at koden er

trygt lagret i tilfelle PCene skulle bli ødelagt eller informasjonen på de blir slettet av en eller annen grunn. Et av kravene til koden er at en skal følge en kodestil. For dette prosjektet har Google sin blitt brukt. VS Code har en plugin for CheckStyle som gjør det enkelt å sjekke at koden følger stilen.

Oppgaven ble gitt til oss i tre deler. Fremgangsmåten for å løse oppgaven har derfor vært å gjøre hver av de tre delene individuelt i stedet for å gjøre alt i ett. Dette forsikrer at oppgaven blir gjennomført slik faglærer tiltenkte det og minsker sannsynligheten for at en gjør feil, siden en løser oppgaven i mindre deler.

Underveis i utviklingen ble det ført notater av prosessen og valgene som ble gjort, og dette ble ført inn i rapporten. Å reflektere på denne måten medførte jevnlige vurderinger av om programmet følgte designprinsippene, og har dermed hjulpet med å sikre robustheten til koden. Notatene var også til hjelp for del 3 av oppgaven, som er "refactoring" av koden.

Som beskrevet i 3.1, så består programmet av tre forskjellige deler: en enkeltvare, samlingen av varene og brukergrensesnittet. Dette gir et bra utgangspunkt for hvordan en kan strukturere klassene som javakode. En kan lage en klasse som representerer et produkt, en som representerer inventaret og en som inneholder brukergrensesnittet. I denne løsningen er disse klassene henholdsvis: "Product", "Inventory" og "Client."

Å lage Product klassen var oppgaven i del 1, så denne ble utviklet først. Siden Product er en informasjonsklasse, så skal den være uavhengig funksjonaliteten til resten av programmet og kan derfor trygt implementeres for seg selv. En skulle også lage en testklient for å sjekke om klassen oppfører seg som ønsket.

I designet av "Inventory" og "Client" er det nødvendig å vurdere hvordan de skal samhandle og å ta hensyn til designprinsipper for å gjøre koden mest mulig robust. Det ble valgt at "Client" kun skulle vise informasjon til brukeren og ta innputt. Client bruker så innputten for å kalle på metoder i "Inventory" som så utfører handlinger på produktene, og gir eventuelt informasjon om produktene tilbake til klienten. "Client" skal altså ikke kunne manipulere produktene direkte, og "Inventory" skal heller ikke kommunisere med brukeren direkte. Utviklingen av disse klassene ble gjennomført fra dette utgangspunktet.

Å utvikle disse to klassene samtidig hadde en gunstig bieffekt: Client kunne brukes for å teste funksjonaliteten til Inventory. Siden funksjonaliteten brukeren skal kunne utføre korresponderer til funksjonaliteten i Inventory, så gir det mening å utvikle denne funksjonaliteten samtidig. Derfor ble metoden for å ta innputt fra brukeren for en spesifikk funksjon i grensesnittet skrevet i tandem med metoden som utfører den handlingen i Inventory. En kunne dermed teste både om innputt ble håndtert på en feilsikker måte i Client, og at denne innputten førte til ønsket aktivitet i Inventory.

I del 3 av prosjektet, skulle en vurdere hvordan koden kunne forbedres i henhold til prinsippene, eller om det er hensiktsmessig å utvide funksjonaliteten til programmet. Å gjøre endringer basert på en slik vurdering heter "refactoring." En ble også bedt om å lage en enum for å representere kategoriene som et produkt kan ha. Denne enumen ble lagt i en egen fil som heter "Category", slik at den kan brukes i alle delene av programmet.

Det ble gjort endringer på to aspekter av koden: den ene var å endre på måten programmet håndterer feil, og den andre var å utvide funksjonaliteten slik at brukeren kan endre på alle

feltene til et produkt. Endringen av feilhåndtering ble gjort på grunnlag av å bedre følge "service oriented design." Grunnen til å endre til at alle feltene til en vare kan endres var brukervennlighet. Når brukeren legger til et produkt, er det mulig at de skriver feil på en eller flere av feltene. Hvis brukeren kun kan endre pris og/eller beskrivelse, så må de slette produktet og lage det på nytt. At brukeren kan endre på hvert felt individuelt minsker arbeidsmengden for å korrigere en feil.

Etter koden var ferdigstilt ble javadoc skrevet for å dokumentere koden, og rapporten ble ferdigstilt. Både public- og private-metoder ble dokumentert med javadoc. Javadocen for public-metodene beskriver APIet til klassen, og javadocen for private-metodene beskriver funksjonaliteten for at eventuelle fremtidige utviklere skal raskere kunne sette seg inn i koden.

6 RESULTATER

6.1 Øvrige designvalg

Som beskrevet i seksjon 5, så ble det identifisert at programmet kunne deles i tre separate oppgaver, og i koden ble disse til klassene Product, Inventory og Client. Product er en informasjonsklasse, og har derfor kun felter for å holde på informasjon og metoder for å hente og endre på verdiene til disse feltene. Inventory representerer inventaret, og er den eneste klassen som skal holde på og endre Product-objektene som representerer varene i varehuset. Client representerer brukergrensesnittet, og skal kun vise informasjon til brukeren og hente innputt. Disse valgene sikrer også høy cohesion i designet av klassene, siden de kun utfører oppgaver som tilhører deres konseptuelle domene.

6.2 Vareklassen

"Product" er en informasjonsklasse. Den har felter for all informasjonen en vare skulle holde på i følge kravspesifikasjonen. Klassen tilbyr kun metoder for å hente og endre på verdiene til feltene. Datatypene til feltene ble valgt basert på det som stod i oppgaveteksten, siden beskrivelsen av slik dataen skulle lagres kunne bli direkte oversatt til datatyper i Java. I del 3 av prosjektet ble kategori feltet endret fra å bruke et tall til å bruke en enum ettersom det ble eksplisitt skrevet at vi skulle gjøre denne endringen.

Klassen har mutatormetoder ("setters") for alle feltene. Dette er fordi kravet om at brukeren skal kunne endre pris og/eller beskrivelse ble utvidet til at det er mulig å endre på alle feltene. Denne endringen ble gjort på grunnlag av å bedre brukeropplevelsen. Hvis brukeren skriver feil på en av feltene når de legger til en vare, og verdien er gyldig i henhold til typen og verdimengden til feltet, så må brukeren slette varen og skrive inn hele på nytt. Hvis en i stedet kan endre på hvert felt, så minsker det arbeidet brukeren må gjøre for å korrigere feilen.

Alle setterene utfører en sjekk på om verdien de får som argument er gyldig i forhold til verdimengden som feltet kan holde (f.eks. så sjekker setteren for pris-feltet at argumentet som blir gitt er større enn null.) Hvis argumentet ikke er innenfor verdimengden, så utløser metoden unntaket "IllegalArgumentException" med beskjed om at argumentet ikke hadde en gyldig verdi. Dette er med på å gjøre koden feilsikker, ettersom et Product-objekt aldri kan ha ugyldige verdier.

Product oppnår altså høy cohesion ved at den kun har metoder for å hente og endre på informasjon den lager. Den oppnår også loose coupling ved at brukere av klassen kun har tilgang

til informasjonen gjennom definerte metoder. Den er også en robust klasse ved at den sjekker at verdier som blir gitt til setterene som argument ligger i verdidomenet til feltene.

6.3 Inventory

Som beskrevet i 6.1, så ble det bestemt at Inventory er den eneste som skal inneholde og endre på produktene i inventaret. Klassen må altså ha en samling som holder på Product-objekter, og den må ha metoder som en bruker kan nytte seg av for å endre på inventaret. Dette medfører høy cohesion, siden Inventory er den eneste klassen som behandler inventaret, og loose coupling, siden andre klasser kun kan behandle inventaret gjennom metodene Inventory tilbyr.

I oppgaveteksten ble det gitt at vi skal bruke en av samlingene som er tilgjengelige i JDK, og ArrayList, HashMap og HashSet ble gitt som eksempler. Av disse tre ble HashMap og ArrayList vurdert. HashMap har fordelen den har konstant tid for get og put, respektivt å hente og å legge til et objekt i samlingen [6]. Det vil si at det kun tar en operasjon for å hente elementet eller legge til et element i et HashMap.

Måten HashMap oppnår konstant søketid på, er ved at en tildeler hvert element en nøkkel når en legger det til. I dette programmet kunne det f.eks. vært id'en til varen. Når en vil hente et element fra HashMapet, så kaller en get metoden til HashMap objektet, og en gir nøkkelen til elementet som argument til metoden. HashMapet gjør så en behandling på nøkkelen som gir den minneadressen til elementet, og kan dermed hente ut verdien direkte.

Hovedgrunnen til at HashMap ikke er et fornuftig valg for dette programmet er at brukeren skal kunne søke etter både id og beskrivelse. Siden beskrivelsen ligger i Product-objektene, så må en iterere gjennom verdiene i HashMapet for å hente ut beskrivelsen til et produkt og sammenligne den med brukeren sitt innputt. Dette er samme måten som en vil bruke for en ArrayList. Å måtte iterer gjennom verdiene til HashMapet går imot konseptet til denne typen samling, og det gir derfor mening å bruke det i dette programmet. Derfor falt valget på ArrayList for å holde på Product-objektene.

Det er også Inventory som lager Product-objektene som blir lagt til i samlingen. Andre klasser gir kun informasjonen som blir produkt for å lage et Product-objekt til Inventory. Når Inventory returnerer et Product-objekt til andre klasser, så returnerer den en dyp-kopi. Dette sikrer at andre objekter ikke kan endre på produktene som ligger i inventaret direkte, som øker robustheten til programmet. Med dette, så har en komposisjon for håndteringen av Product-objektene.

Siden andre klasser ikke har mulighet til å endre på inventaret direkte, så må de ha en måte å peke på hvilket objekt de vil hente informasjon om eller endre på. Det ble bestemt at id'en til et produkt som ligger i inventaret må være unik, og en kan dermed bruke denne for å peke på et unikt objekt. Metodene til Inventory som behandler et enkelt objekt tar inn en streng med id'en til et produkt, som den bruker for å hente objektet som skal behandles.

6.4 Client

Valget falt på at terminalen skal brukes for å implementere grensesnittet. Grunnen til dette er at vi ikke har lært å lage et grafisk grensesnitt, og fordi Java har god funksjonalitet for å både vise informasjon i terminalen, og for å hente innputt fra den. Grensesnittet er implementert i klassen Client.

Client har tre funksjoner den skal oppfylle. Den første er å vise informasjon til brukeren om handlinger de kan gjøre og produktene i inventaret. Den andre er å kalle metodene i Inventory som korresponderer til handlingen brukeren har valgt å utføre. Den tredje er å ta innputt fra brukeren og konvertere det til et format som kan brukes i disse metodene.

Den første funksjonen oppfyller den ved å vise en meny med de ulike handlingene en bruker kan utføre. Brukeren kan så skrive inn et tall som tilsvarer ett av valgene, og Client kaller så på metoden som korresponderer til dette valget. Hvis en funksjon krever mere innputt fra brukeren, så skriver Client instruksjonene som brukeren må følge i terminale. Den henter så innputten fra brukeren og konverterer eventuelle verdier som skal sendes til metoden i Inventory til et format den kan bruke. Den kaller deretter på den aktuelle metoden i Inventory og skriver resultatet eller en beskjed om at handlingen var ble gjennomført eller var mislykket til terminalen. Etter dette går den tilbake til hovedmenyen, og brukeren kan utføre flere handlinger eller avslutte programmet.

En utfordring med å hente informasjon fra terminalen er å validere at verdiene som brukeren skriver inn er gyldige for feltet, både ovenfor datatypen og om det ligger i dets verdidomene. Dataene en henter fra terminalen kommer i form av tekst. Når brukeren skal skrive inn et tall må en konvertere teksten til en tallverdi. Utfordringen med dette er at det er fullt mulig for brukeren å skrive inn noe annet enn et tall. En må dermed håndtere disse tilfellene. Dette fører til spørsmålet om hvem som er ansvarlig for å håndtere feilene og hvordan dette skal gjøres.

I designet av programmet ble det valgt at det er Client-klassen som er ansvarlig for å håndtere å vise informasjon til brukeren og å hente innputt. Inventory-klassen er dog ansvarlig for å håndtere Product-objektene. Siden Inventory kun er ansvarlig for håndteringen av Product-objekter, så faller det naturlig at brukerne av klassen må gi verdier som kan brukes direkte i et slikt objekt. Det ble derfor valgt at feilhåndteringen av innputt skjer både i Client og Inventory. Denne samhandlingen blir beskrevet i seksjon 6.5.

Client er en klasse som ikke skal brukes av andre klasser, og prinsippet om loose coupling er derfor ikke relevant for denne klassen. I henhold til programmet sitt design, så oppfyller Client høy cohesion ved at den kun viser informasjon til brukeren og behandler innputt. Hvordan den oppnår robusthet i behandlingen av innputt blir diskutert i seksjon 6.5. Den oppnår modularitet ved at hver handling brukeren kan utføre blir gjennomført i sin egen metode.

6.5 Samhandling mellom klassene

Programmet har en lagdelt arkitektur med tre lag: brukergrensesnittet, inventaret og produktene. Det øverste laget er brukergrensesnittet. Brukergrensesnittet kan kun kommunisere med inventaret, som er det midterste laget. Inventaret kan så kommunisere med de individuelle produktene og videreformidle dette tilbake til brukergrensesnittet.

Å skrive en gjenbrukbar klasse kan være en utfordring når funskjonaliteten til klassen er tett knyttet til formålet til programmet. I dette tilfellet, så er alle de offentlige metodene i Inventory laget for å oppfylle kravene om hva en bruker skal kunne gjøre gjennom brukergrensesnittet. En sentral problemstilling som har vært gjennomgående i utviklingen er hvordan Inventory skal bli implementert slik funksjonaliteten ikke er avhengig av hvordan Client funker.

Som nevnt i 6.4, så ble feilhåndtering fordelt på både Client og Inventory. Klassene er dog ansvarlig for forskjellige typer feilhåndtering. Slik Inventory er implementert, så må en bruker gi et Product-objekt eller verdier som kan settes direkte inn i et Product-objekt. Siden Client tar inn tekst som innputt, så må konverteringen til tallverdier skje i denne klassen. Feilhåndtering for at brukeren skriver inn en ugyldig type for et felt, f.eks. en streng der det skal være et tall, skjer i Client. Inventory er så ansvarlig for å forsikre at verdiene som blir gitt som argument ligger i verdidomenet til feltet. Med denne oppdelingen, så kan Inventory ta nytte av at Product-klassen har funksjonalitet for å sjekke at verdiene som blir gitt til setterene ligger i verdidomenet til feltet. Utløste unntak blir så fanget opp i Client, siden de der kan brukes til å vise hva som gikk galt til brukeren. Dette medfører høy cohesion i denne delen av programmet, og sikrer graceful termination.

For at en bruker av Inventory skal kunne utføre handlinger på varene i inventaret, må de supplere id'en til varen i metodekallet. I kravspesifikasjonene, så skal brukeren kunne søke etter id og/eller beskrivelse. For å øke brukervennligheten til programmet, så ble det implementert at brukeren ikke måtte skrive inn hele id'en eller beskrivelsen, og søket på begge skjer i samme funksjon i brukergrensesnittet. Søket i selve koden er dog delt i to forskjellige metoder som hver for seg håndterer søket etter id og beskrivelse.

I klienten skriver brukeren enten en del av eller hele id'en for å søke etter den, eller ord som forekommer i beskrivelsen. Metoden som søker etter id forsøker så å matche innputten fra starten av id'en. Hvis id'en er "AB12", så vil "A", "AB", "AB1" og "AB12" matche, men "aB" eller "B12" vil ikke. Hvis flere id'er matcher, så blir alle returnert. Hvis innputten ikke matcher en id, så blir det gitt videre til metoden som søker etter beskrivelse. Den vil returnere produkt(ene) som har flest ord som forekommer i innputten. En dyp-kopi av produkt(ene) blir så returnert til klienten som presenterer de i en meny der brukeren kan velge hvilket produkt som skal brukes videre. Denne søkefunksjonen blir brukt for alle metodene der ett produkt skal behandles. Dette forsikrer at produktet som behandles alltid eksisterer.

6.6 Refactoring

"Refactoring" ble ikke gjort i noen stor grad. Måten klassene og metodene ble implementert på i utgangspunktet tilfredsstilte i stor grad funksjonaliteten som skulle implementeres, og det var ingen innlysende endringer som kunne bli gjort for å forbedre funksjonaliteten.

I del 3 av prosjektet, så ble en eksplisitt bedt om å endre koden fra å representere kategori-feltet i Product fra å bruke en tallverdi til en enum. Utenom å måtte skrive koden for selve enumen, medførte ikke dette ikke store endringer i resten av programmet, som er et tegn på at den er bra skrevet. Hvis Smarthus AS har lyst til å implementere flere kategorier, så må en legge til de nye kategoriene i enumen og utvide metoden "parseCategory" for å håndtere konverteringen fra en streng til den nye kategorien. En må så simpelthen legge til valgmuligheter for de nye kategoriene i brukergrensesnittet. Ingen andre endringer av koden trengs, noe som er et tegn på at implementasjonen i programmet er solid.

6.7 Dokumentasjon

Dokumentasjon av koden er en sentral del av prosjektet. Javadoc har blitt skrevet for alle metoder. Det er kun i en metode at det er skrevet kommentarer i selve koden. Dette er et tegn på at koden har blitt skrevet på en selvdokumenterende måte, som er ett av de øvrige kravene som koden skal oppfylle.

7 DRØFTING

Den endelige versjonen av programmet har fullført alle kravene som ble gitt i kravspesifikasjonen, og de øvrige kravene for kodestil og dokumentasjon. Den har i tillegg utvidet funksjonaliteten for hva brukeren kan endre på et produkt til at det er mulig å endre alle feltene. Programmet sin implementasjon følger også i stor grad prinsippene om lagdelt arkitektur, cohesion, loose coupling, robust design, graceful termination og feilhåndtering. Dette har blitt argumentert for i seksjon 5 og 6.

Selve utviklingsprosessen kunne ha blitt utført bedre. En stor mangel fra det vi har lært i pensum er bruken av diagrammer. Ingen diagrammer ble laget før eller i løpet av utviklingsprosessen startet, og ingen har heller ikke blitt supplert i rapporten. Forfatter hadde også en svak forståelse av flere av prinsippene som koden skulle følge, og måtte jevnlig referere til kilder for å forsikre

seg at de ble fulgt. Bruken av diagrammer og en grundigere forståelse av prinsippene på forhånd kunne ha ført til en raskere og enklere utviklingsprosess.

Testing av koden løpende gjennom utviklingen har også vært mangelfull. Bortsett fra testklienten som det ble stilt krav om i del 1 av prosjektet, så har ingen konkret testkode blitt skrevet for resten av programmet. Som nevnt i seksjon 5, så var en fordel med den samtidige utviklingen av klienten og inventarklassen at en kunne bruke klienten til å teste for ønsket oppførsel for både klienten og inventarklassen. Dette er dog ikke en like fullverdig måte å teste koden på, og eksplisitt testkode hadde gjort gjennomføringen av oppgaven mere robust i henhold til gode utviklingsprinsipper.

En utfordring har også vært å finne gode kilder for prinsippene. De har i stor grad blitt gjennomgått i forelesninger, noe det ikke er mulig å referere til. Pensumboken dekker heller ikke alle prinsippene. Dermed har forfatter måtte lene seg på kilder som ikke nødvendigvis er av akademisk kvalitet.

8 KONKLUSJON - ERFARING

For å konkludere, så var resultatet av selve produktet tilfredsstillende både ovenfor kravene som ble stilt til programmet og ovenfor teorien som skulle anvendes. All funksjonalitet ble implementert, og det er tydelig vist i foregående seksjoner at bruken av prinsippene har ført til robust kode. Koden er også dokumentert grundig, slik at brukere og utviklere kan lett sette seg inn i hvordan den funker og hva den tilbyr.

Selve gjennomføringen av prosjektet har dog forbedringspotensiale. Som nevnt i seksjon 7, så kunne forberedelsen til utviklingen vært grundigere gjennomført. I senere prosjekter, så vil det være nyttig å lage diagrammer for å få en oversikt over strukturen til programmet før utviklingen begynner. Verdien av å ha en grundig forståelse av teorien en skal anvende i gjennomføringen av prosjektet er også noe som er nyttig å ta med seg videre.

Utvidelsen av funksjonaliteten til å la brukeren kunne endre alle feltene viste seg å være både relativt komplisert og tidkrevende prosess. Denne delen av programmet er den største enkeltenheten og den som tok mest tid til å implementere. Det hadde derfor vært nyttig å lage en skjelettløsning for å analysere om den økte nytteverdien er verdt utviklingstiden det vil medføre. Å gjøre slike vurderinger vil være nyttig i fremtidige prosjekter.

Som vurdering av prosjektet som helhet, så er det et bra arbeid. Det endelige produktet tilfredsstiller kravene som ble gitt i oppgaveteksten, og er løst på en solid måte. Selv om utførelsen ikke var optimal, så har erfaringen av gjennomførelsen ført til refleksjoner om hva som kunne blitt gjort bedre, som har stor nytteverdi. Hovedformålet med en akademisk oppgave som dette er å lære, og dette formålet har i aller største grad blitt oppfylt.

9 REFERANSER

[1] Core Java, Volume 1: Fundamentals, twelfth edition. Horstmann, Cay S.

[2] Modular programming. Wikipedia. Hentet 13.12.2022 fra: https://en.wikipedia.org/wiki/Modular_programming

[3] Cohesion in Java. GeeksforGeeks. Hentet 13.12.2022 fra: https://www.geeksforgeeks.org/cohesion-in-java/

[4] Coupling in Jaava. GeeksforGeeks. Hentet 13.12.22 fra: https://www.geeksforgeeks.org/coupling-in-java/

[5] Responsibility driven design. Wikipedia. Hentet 13.12.22 fra: https://en.wikipedia.org/wiki/Responsibility-driven_design

[6] HashMap. Oracle corporation. Hentet 13.12.22 fra: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html