RAPPORT MAPPEVURDERING

NTNU

KANDIDATNUMMER:			
10019			
DATO: 13.12.22	FAG/FAGKODE: PROGRAMMERING 1, IDATA1001	STUDIUM/STUDIESTED: DATAINGENIØR, NTNU TRONDHEIM	ANTALL SIDER: 21

FAGLÆRER(E) :

MOHAMMED ALI NOROZI
ARNE GERHARD STYVE
KIRAN BYLAPPA RAJA
GEIR OVE ROSVOLD
SURYA BAHDUR KATHAYAT

TITTEL :
RAPPORT MAPPEVURDERING, PROGRAMMERING 1

SAMMENDRAG:

Denne rapporten er besvarelsen til en obligatorisk innlevering i faget IDATT1001 Programmering 1, høsten 2022. Hensikten med rapporten er å forklare et program, prinsipper som er brukt i tilknytning til det, fremgangsmåte, teori, utfordringer og løsninger.

Oppgaven går ut på å lage et WMS til en bedrift, der varer skal registreres og organiseres gjennom et brukerstyrt grensesnitt.

Rapporten vil forklare hva slags fremgangsmåter som er blitt brukt for å løse oppgaven, resultater og konklusjoner, samt komme med tilhørende teori og drøfting rundt disse.

INNHOLD

1 SAMMENDRAG

Denne rapporten er besvarelsen til en obligatorisk innlevering i faget IDATT1001 Programmering 1, høsten 2022. Hensikten med rapporten er å forklare et program, prinsipper som er brukt i tilknytning til det, fremgangsmåte, teori, utfordringer og løsninger.

Oppgaven går ut på å lage et WMS til en bedrift, der varer skal registreres og organiseres gjennom et brukerstyrt grensesnitt.

Rapporten vil forklare hva slags fremgangsmåter som er blitt brukt for å løse oppgaven, resultater og konklusjoner, samt komme med tilhørende teori og drøfting rundt disse.

2 TERMINOLOGI

UML	Unified Modelling Language
OOP	Object Oriented Programming
ENUM	Enumerations
IDE	Integrated development environment
ISP	Interface Segregation Principle
SRP	Single Responsibility Principle
WMS	Warehouse Management System
Regex	Regular Expression

3 INTRODUKSJON/PROBLEMSTILLING

3.1 Bakgrunn/Problemstilling

"Smarthus AS" er et varehus som lagrer og distribuerer/selger varer knyttet til bygg. De ønsker å utvikle en programvare/WMS (Warehouse management system) som skal brukes til lagerstyring, av deres ansatte. Oppgaven er å programmere et system som oppfyller kundenes kravliste og til syvende og sist vil bestå av et tekstbasert brukergrensesnitt og et register som lagrer informasjon.

Selskapet har krav til programmets utseende, funksjonalitet med mer. Her er de viktigste. Selve oppgavebeskrivelsene legges til som vedlegg til dette dokumentet.

Varene skal registreres med 10 variabler gitt fra kunden. Alle varer skal registreres innenfor en kategori, med bruk av kategorinummer, som skal administreres av Enums. Programmet må ha visse funksjoner som slett element, søk etter element med mer. Alle navnene som gis til klasser, metoder, variabler, parametere osv. må være klare og beskrivende samt følge en bestemt kodestil uten regelbrudd med "Checkstyle-plugin" ved levering. Designet skal være robust og følge prinsipper som modularisering, kobling, kohesjon, ansvarsdrevet design etc. Til slutt skal

programmet presenteres på en måte som er brukervennlig nok for operatøren å håndtere, som i vårt tilfelle er en ansatt ved lagerhuset.

3.2 Begrensninger

Arbeidsgiver satte ikke mange grenser for oppgaven, og la derfor god plass til personlige tilpasninger som kan skape mer funksjonalitet og forbedre programmet ytterligere.

Imidlertid ble de fleste datatypene til verdiene i varen gitt, for eksempel skal prisen angis som et heltall, selv om det kan være mer fornuftig at det er en dobbel. En annen begrensning var at kategorien måtte registreres etter kategorinummeret, og ikke selve kategorien.

Foruten oppgaven gitt fra kunden, ble oppgaven begrenset av pensum. Det var meningen at mer kunne vært gjort for å forbedre programmet, men det var ikke mulig siden pensumet ikke har fått det til.

3.3 Begreper/Ordliste

Begrep (Norsk)	**Begrep (Engelsk)**	**Betydning/beskrivelse**
Blank	Blank	*Tom, ikke fylt inn.*
Merke	Brand	*Navnet til produsenten av varen.*
Kategori	Category	*Deler varer inn i spesifikke fellestrekk.*
Endre	Change	*Å bytte om noe fra en ting til noe annet.*
Farge	Color	*Eks. er rød, gul, hvit*
Inneholder	Contains	*Å bestå av*
Slett	Delete	*Fjerne noe (for godt).*
Beskrivelse	Description	*Et avsnitt som forklarer noe som omgår den gjeldene varen*
Desimal tall	Decimal number	*Et tall som består av et komma (,) eller punktum (.).*
Rabatt	Discount	*En verdi som bestemmer hvor mye billigere du kan kjøpe en gjenstand.*
Dører	Doors	*En gjenstand som dekker et hull i byggets åpning som er ment for passasje.*
Avslutt	Exit	*Når man vil legge ned eller ikke være i programmet mer*
Finn	Find	*Lete etter noe for så å oppdage det.*

Begrep (Norsk)	**Begrep (Engelsk)**	**Betydning/beskrivelse**
Gulv laminat	Floor laminate	*Et byggemateriale som brukes på gulv. Oftest laget av tre.*
Høyde	Height	*Loddredd avstand på gjenstanden.*
Heltall	Integer	*Et tall som ikke består av desimaler.*
Vare	item	*En gjenstand, som selges.*
Lengde	Lenght	*Vannrett avstand på gjenstanden.*
Trevirke	Lumber	*Materiale som er laget av tre, eks. spesifikt lister*
Meny	Menu	*Programvalg*
Navn	name	*Det noe heter*
Nummer	number	*En rekke med tall, som i dette tilfelle også består av bokstaver.*
Forskjellig	Different	*Når noe er ulikt noe annet er det forskjellig.*
Pris	Price	*Sum av verdi i valuta for noe.*
Register eller registrere	Register	*Register –holder på en rekke med varer.* *Registrere – legge til noe nytt, i register.*
På lager	Stock	*Antall varer i/på lager.*
Sorter	Sort	*Når noe blir systematisert på et vis.*
Streng	String	*Streng av tekst. Kan både være enkeltord, linjer eller avsnitt.*
Total	Total	*En sammenlagt sum.*
Verdi	Value	*I denne sammenheng, sum i valuta.*
Vekt	Weight	*Hvor tungt noe er.*
vinduer	Windows	*Lysåpning I en bygnings vegg.*

3.4 Rapportens oppbygning

Rapporten består av et kort sammendrag og en terminologi del, for å gi en oversikt over innholdet i rapporten, før den går videre til å introdusere problemstillingen og bakgrunnen for denne. Videre blir det gitt ett innblikk i teorien som ligger bak oppgavens løsning. Deretter følger en metode del som beskriver hvordan oppgaven ble utført, før resultatene presenteres

og drøftes i en egen del. Til slutt er det konklusjon og erfaringer, samt en liste over kilder som er brukt og vedlegg.

4 BAKGRUNN - TEORETISK GRUNNLAG

Under utviklingen av applikasjonen har det vært fokus på en del teori og prinsipper. Hensikten med å buke disse prinsippene er å lage en stabil, sikker og feilfri programvare med god struktur, som er lett å vedlikeholde/forbedre. Teorier som har blitt tatt i bruk for å oppnå dette har vært å oppnå robust- og «responsibility driven»- design, samt ta i bruk prinsipper som modularisering, «coupling» og «cohesion»,

4.1 Robust design:

Et robust design vil si et design som reduserer følsomheten for variasjonskilder istedenfor å kontrollere kildene deres. Altså et «failproof» design som er vanskelig å tukle med, som tilslutt vil gi et program som har liten følsomhet for variasjoner i ukontrollerbare faktorer[1].

Debugging er en ofte brukt metode i lærerboken [4. 633] som brukes for å finne feil i programmet, gjennom å gå stegvis over koden for å se hvor problemet oppstår.

En annen form for debugging er å benytte seg av print statements som en legger til i koden for å se om noe har blitt gjennomført eller ikke. Ved å gjøre denne typen debugging kan man se hvor programmet gikk feil basert på outputen programmet gir.

En annen metode lærerboken benytter seg av [4. 633s] er testing gjennom et brukergrensesnitt. Her kan man teste om klassene og metodene fungerer som de skal, ved å prøve seg fram å «forsøke» å ødelegge programmet, for så å oppdage svake sider ved programmet, og dermed utbedre det.

Enhetstesting er også en form for testing som kan bidra til en robust kode. Boken henviser seg ofte til JUnit, som er en type enhetstesting som gjør det lett for programmerere å organisere en enhetstest.

Å kaste «exceptions» ved hjelp av «try catch» eller «if» statements er god måte å gjøre koden robust på, på den måten at man kan gi tilbakemeldinger på hva som kan ha godt galt i et program om dette skjer. Dette vil både hjelpe under testing fasen for å se hvordan programmet fungerer, men er også mer brukervennlig da bruker kan få en forståelig feilmelding de kan gjøre noe med.

Uten «exceptions» hadde vi også ikke hatt noen robust kode, men en kode som klikker og kaster ut brukeren av programmet. Dette er noe vi ikke vil ha. Det vi sikter etter er «graceful termination» hvor bruker blir gitt nye valg med en enkel å forstå feilmelding, i stedet for et program som krasjer og displayer masse uforståelig tekst bruker ikke kan få gjort noe med, samt gjør at bruker ikke får brukt programmet.

4.2 Modularisering:

Modularisering er et viktig prinsipp i OOP, hvor man deler opp et større og mer komplekst problem, inn i mindre og mer forståelige deler.

Dette gjør programmet mer oversiktlig når man koder, i tillegg til at programmet blir mer strukturert, som igjen gjør det lettere for flere programmerere å jobbe sammen på koden. Oppstykkingen gjør det altså lettere å implementere mer kode i ettertid.

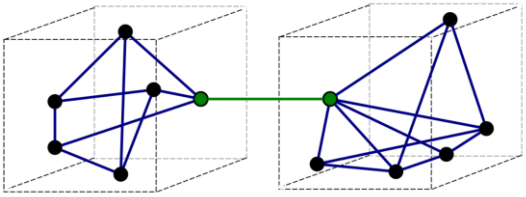
Oppstykkingen i koden lages på en slik måte som gjør at kode er mulig å gjenbruke. Ikke bare gjør dette koden kortere, som igjen bidrar til at den blir mer oversiktlig, men det betyr også at man lettere kan finne oppståtte feil.

Modularisering gjør det også lettere å teste kode, ettersom metodene blir mindre, men mange. Da er det lettere å konkret finne frem til metoden som forårsaker en potensiell feil.

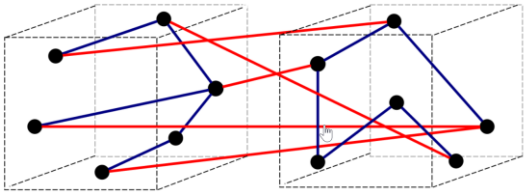
Metoden legger også opp til at kode kan gjenbrukes. Dette kan kutte ned på koden som igjen gjør den mer oversiktlig, men også dersom det oppstår en feil utenfor testingen av hver enkelt metode også. Ettersom problemet da kan oppstå i en metode som blir gjenbrukt, gjør det rette jobben mindre, da man kun trenger å rette på den ene gjenbrukbare metoden, fremfor masse småbiter med kode rundt om i programmet.

4.3 Coupling og cohecion:

«Coupling og cohecion» er ofte begreper som ofte henger sammen, og er viktige konsept i OOP.



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Ifølge læreboken [4 s.203] og faglærer Ali [5] handler «coupling» (kopling) om forholdet mellom klasser og hvor mye de avhenger av hverandre. Altså hvor mye kunnskap et element har om et annet. Med andre ord vil det si at en klasse A ikke er bevisst på eksistensen av en klasse B, og vil derfor ikke ha noe tilknytning til det som skjer i klassen. Dette vil føre til at oppståtte feil ikke vil følge gjennom klasser, emn forbli i sin klasse. Klassene er da selvstendige, noe vi ønsker å oppnå.

Det finnes to typer kopling, lav kopling og høy kopling (tight/loose). Lav kopling er det man ønsker [6, bilde], og antas ofte å være et godt tegn på en god kode. Grunnen til dette er at det fører med god lesbarhet og gjør koden lettere å vedlikeholde (endre/fikse/bedre)

Cohecion (kohesjon) er noe faglærer Ali [5] beskriver som antall ulike oppgaver en enhet i programmet har ansvar for. Man kan si at på en måte er kohesjon et mål på styrken til forholdet til metoder i en klasse.

Det finnes to typer kohesjon, høy kohesjon og lav kohesjon. Her er det høy cohecion man ønsker å oppnå da det gjør det kommer med en del fordeler. Det blir lettere å gjenbruke kode slik at programmet blir kortere og mer oversiktlig. Koden vil også bli lettere å forstå da metoden gjør en mer spesifikk jobb og dermed er lettere å navngi godt, noe som kan være spesielt nyttig i større eller mer komplekse programmer. Dette at metoder kun skal ha en spesifikk oppgave bygger ifølge faglærer Ali på prinsippet SRP. kohesjon gjør også at vedlikehold blir lettere.

Kobling står vanligvis i kontrast til kohesjon. Lav kobling korrelerer ofte med høy kohesjon, og omvendt. Hvis disse prinsippene blir brukt riktig kan dette lettere bidra til en mer robust kode, ettersom det vil være lettere å se feil, som vil føre til færre feil, i tillegg til at feilene er lettere å finne/løse når disse prinsippene blir brukt, ettersom kode gjenbrukes. Den gjenbrukte koden er da lettere å endre feil i, enn å måte lete frem samme feil enkeltvis flere steder i koden, fordi kode ikke har blitt gjenbrukt. Når disse to prinsippene jobber sammen, vil refaktorering også bli gjort betydelig lettere.

4.4 Responsibility driven design:

«Responsibility driven design» er enda et viktig prinsipp innen OOP når det kommer til datasikkerhet. Ifølge faglærer Ali forbedrer designet innkapsling av data, som gjør det vanskelig for uvelkomne å stjele eller endre på sensitiv informasjon. Dette er et meget viktig prinsipp når en jobber med programmer som holder på store mengder sensitiv data som eks banker.

«Deep copy» (dypkopiering) av data er noe man gjør innenfor «responsibility driven design», da man aldri sender ut originaldata, men kopi av dette til bruker. Altså vil man forhindre at to verdier som referer til samme objekt, skal være mulig å endre. På denne måten forhindrer man muligheten for manipulering av data.

Når det kommer til tilgang av sensitiv informasjon, er det viktig å følge prinsipper innen kopling og kohesjon, samt aggregering og komposisjon.

4.5 Aggregering og komposisjon:

Aggregering og komposisjon handler ifølge faglærer Ali [11] om forholdet mellom objekter. I aggregering kan delobjektene leve uavhengig av hverandre, mens i komposisjon har komponentene ansvaret for delobjektets lov og død.

Et eksempel på aggregering er en bil og dets komponenter. En bil kan bli ødelagt, men hjulene på bilen vil fortsatt kunne eksistere.

Et eksempel på komposisjon er en skole med et klasserom. Klasserommet eksisterer bare så lenge skolen eksisterer. Ødelegger man skolen, forsvinner klasserommet også.

4.6 Design prinsipper

Utenom de mer store prinsippene, finnes det også noen mindre, emn også viktige. Noen av de mest generelle vil være:

Abstraksjon er en teknikk for skjule kompleksitet og trekke ut det overordnede og viktigste ifølge lærerboken [7][4 s. 391]. Dette vil si at man utelater informasjon på hvordan en bestemt oppgave utføres, slik at man kan fokusere på hva som skal oppnås uten å måtte tenke på hvordan det skal gjøres. Dette gjør det enklere å lage og vedlikeholde kode for programmereren, og gjør det også enklere å forstå og bruke programmet for bruker. Abstraksjon kan brukes på ulike måter i programmering, for eksempel ved å lage funksjoner eller abstrakte datatyper som skjuler detaljene om hvordan de implementeres.

Det å holde data privat er et viktig lite prinsipp, ettersom brudd på dette vil føre til brudd på det større prinsippet «responsibility design» og sikkerhet. Det å holde data privat forhindrer at hvem som helst kan få tak i informasjonen. Ved å gjøre data privat, kontrollerer programmereren hvordan og om informasjonen skal prosesseres videre i programmet.

Riktig og tydelig navngiving er også viktig, da programmer hadde vært et kaos om ikke. Navn på metoder og klasser skal ha gode og konkrete navn som godt beskriver hva de gjør.

Navn konvensjoner, altså hvorvidt man skriver navnet i camelcase, har hele ordet i blokkbokstaver, første bokstav i blokkbokstaver eller skiller ord i navn med understrek. For eksempel skal konstanter ha hele ordet skrevet i blokkbokstaver, mens klassenavn og metoder skal benytte camelcase. Camelcase er når navnet består av flere ord, og første bokstav i hvert ord er skrevet med en stor bokstav. Dette gjelder klasser. I metoder skal den aller første bokstaven være liten.

Bøyning av ord tilhører også riktig navngiving. Klasser skal for eksempel være substantiv, og metoder verb. Dette kan gjelde ved flere områder som ved lister, så skal objektene listen holder (om den holder objekter) så i entall, og navnet på selve listen i flertall.

Riktig navngiving vil også bruke et gjennomgående likt språk i koden. Dette betyr at man skal prøve å unngå å bruke flere forskjellige ord for det samme i en kode, men det betyr også at man skal følge visse standard regler som at «acsess» (tilgang eller aksess) metoder har «get» før verbet, og mutator metoder har «set».

Det finnes mange flere designprinsipper, men dette er noen av de viktigste.

Det anses generelt som god programmeringspraksis å initialere alle variabler, i stedet for å stole på standardverdier. Dette er fordi det kan skje feil man med sikkerhet kan unngå ved å initialere variablene ifølge

læreboken [s.313]. I tillegg kan den gjøre koden enklere å lese, siden den gjør klart hva starttilstanden til en variabel er.

«Interface Segregation Principle» eller ISP, er et prinsipp som sier at bruker ikke trenger å se metoder de ikke bruker. Med andre ord er det bedre å ha flere mindre og mer spesialiserte grensesnitt enn et enkelt stort og generisk, ifølge [8] Prinsippet er ment å forbedre utformingen av objektorienterte systemer ved å fremme løs kobling og høy kohesjon som er meget viktig i OOP.

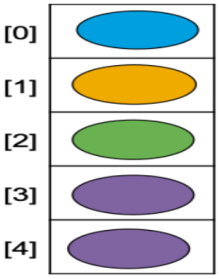
Et prinsipp som ikke direkte er knyttet programmering er likhetsprinsippet. Dette prinsippet går ut på at ting er lettere forståelig om visse mønstre gjentar seg.

4.7 Datastrukturer

«ArrayList», «HashMap» og «Hashset» er alle datastrukturer som fote blitt tatt i bruk. Her har faglærer Ali [10] forklart hva de ulike datatypene er, forskjellen mellom dem og når hva kan lønne seg.

4.7.1 Arraylist

«Arraylist» er en type ordnet liste som lager en samling av elementer i sekvens som de blir lagt inn, kan endre størrelse etter behov, og er en ofte brukt datastruktur brukt i programmering da den er enkel å lage og administrere.



Det at listen tilpasser seg etter behov er en stor fordel i situasjoner hvor man trenger å jobbe med et register f.eks., hvor listen konstant vil være i endring.

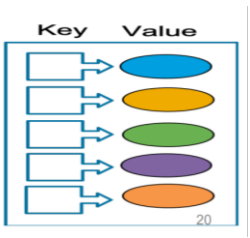
En annen fordel med «arraylist» er at en enkelt kan legge til, fjerne og endre data fordi listen er indeksbasert.

Det er også denne egenskapen at listen er indeksbasert som gjør lett å søke etter og/eller sortere objekter i listen.

«Arraylist» er også en type liste som tillater duplikater, som kan være nyttig til visse funksjoner.

4.7.2 HashMap

«HashMap» er en kart datastruktur som bruker nøkkelpar for å koble verdier. Nøklene hjelper til med å gjøre det enklere å lagre og hente verdier, og funker som identifikasjon da nøklene er unike. Duplikater er derfor ikke lov i nøkler. «HashMap» har heller ikke mulighet til å arve «collection», som blant annet kan gjøre det mulig å sortere.



4.7.3 HashSet

«Hashset» er en uordnet måte å samle objekter/elementer på, ved bruk av «hash»-funksjoner. Dette gir «HashSet» egenskapen til å lete etter elementer i settet, uten å måtte iterere gjennom alle elementene. «Hashset» tillater ikke duplikater, som gjør det enkelt å sjekke om et sett inneholder et bestemt element. «HashSet» kan derimot være litt mer komplisert når det kommer til å hente elementer i en bestemt rekkefølge.

4.7.4 Oppsummert sammenlikning av datastrukturer

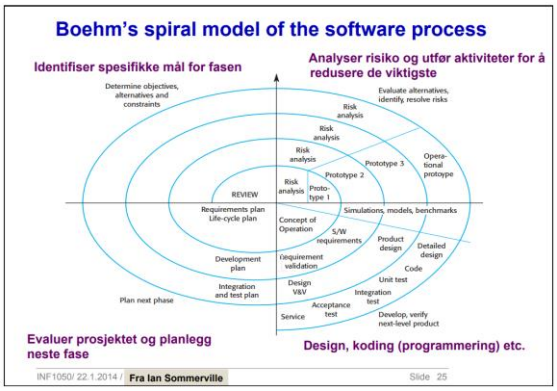
Kort oppsummert avhenger valget mellom «ArrayList», «HashMap» og «HashSet» på hva det skal brukes til. Om du ikke trenger data ordnet, ikke vil ha duplikater, og ikke har elementer med flere enn en verdi, er kanskje «HashSet» valget. Om du har elementer med flere verdier, vil kanskje «HashMap» lønne seg ovenfor «HashSet». Om du derimot jobber med systemer hvor verdier skal være enklest å legge til, fjerne eller endre, inneholder flere verdier i hvert element og hvor du lett vil søke/sortere listen i en rekkefølge, vil «ArrayList» lønne seg fremfor de begge.

4.7.4.1 Itereringsmetoder

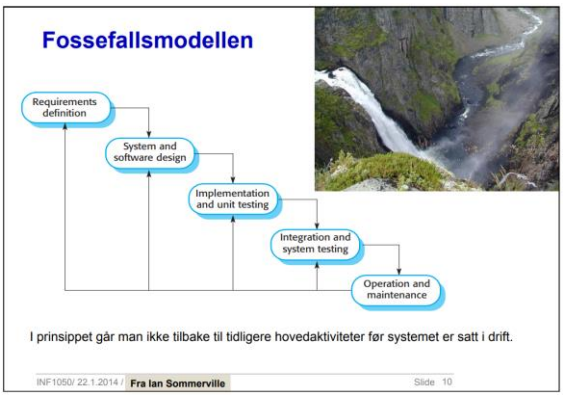
Løkker en en type kontrollstruktur i programmering som gjør at jeg blokk med kode kan kjøres flere ganger. Løkker brukes ofte for å utføre en oppgave på en rekke av data, for eksempel å iterere gjennom elementene i en liste og utføre en oppgave for hvert element. Det finnes for-løkker som går i løkke et bestemt antall ganger basert på avgrensningene eller i «for each» hvor den itererer gjennom likt antall ganger som elementet er langt, og «while-løkker» som går i løkke så lenge parameteren i løkken er «true».

En iterator er et spesialisert objekt som brukes til å iterere over elementene i en datastruktur, som for eksempel en liste eller en mengde. Når du bruker en iterator, kan du hente ut hvert element i datastrukturen en gang, i en bestemt rekkefølge, uten å måtte bekymre deg for hvordan datastrukturen er implementert eller hvordan elementene er lagret.

5 METODE



Prosjektet ble stykket opp i 3 deler, hvor hver enkelt del inneholdt sine krav. Disse kravene ble sett på før det ble laget et oversiktlig design av programmet, som videre ble kodet og testet, før en gikk videre på neste del. Videre på neste del gjentok denne prosessen seg, med mulighet for å gå tilbake i koden for å endre eller forbedre den osv. Denne metoden kalles spiralmetoden [3] (som man kan se på bilde) [9], og er metoden jeg for det meste tok for meg i oppgaven.



Oppgavens oppbygning i seg selv minner imidlertid sterkt om Fossefallsmodellen/metoden [3], uavhengig om man ser på oppgaven som en helhet, eller som en tredelt oppgave. Først ble oppgaven gitt, avgrensninger og nødvendig informasjon rundt funksjonalitet blir så sett på, klasser blir designet og metoder laget. Deretter tester man at de fungerer. På dette stadiet dette stadiet har koden oppfylt de nødvendige kravene som er gitt av oppdragsgiver, og man kan begynne å endre eller implementere ting i koden for å gjøre den bedre. Videre slår/kobler man sammen klassene og metodene til et helhetlig program, for så å teste programmet og finpusse det før innlevering.

Jeg vil si at jeg gjorde en kombinasjon av disse, da jeg hadde en fremdriftsplan [Vedlegg 5] etter fossefallsmodellen, men hvor jeg konstant gikk tilbake i tidligere skrevet kode for å rette opp i eller endre noe, noe man i prinsippet ikke skal gjøre så mye av i fossefallsmetoden.

Kun ved å lese oppgavebeskrivelsen, hadde jeg en tankeprosess hvor jeg prøvde å se det store bildet over programmet. Hvor mange klasser dette kunne trenge, hva de enkelte skulle inneholde av metoder, og hvordan disse metodene kunne endt opp med å se ut. Da oppgaven ble gitt lagde jeg meg en oversikt over hva som måtte med i programmet og hva som var av begrensninger. Deretter skrev jeg koden, og finpusset denne. Dette ble gjort for hver del, hvor jeg underveis gikk tilbake til den gamle koden og endret på ting som, som regel, fulgte med følgefeil. Et eksempel så baserte jeg meg på «index» (indeks) for å finne «items» (varene) i starten, og gikk fort tilbake for å endre på dette da jeg så at det å basere seg på selve varen og varenummer ville gitt mer mening. På dette stadiet er programmet i grunn ferdig. Her begynte jeg å legge til ting som setter et eget preg på koden og som jeg mener at ville bedret koden, samt at jeg så gjennom det som allerede var skrevet av kode for å se om noe kunne vært bedre. Opp til dette punktet hadde jeg samlet notater om

forbedringer, men også sett over en gang til med synet til en ansatt som skal bruke programmet. Til slutt ble programmet slått sammen, og et brukergrensesnitt ble laget.

Som hjelpemidler ble intelleJ brukt som plattform, ettersom programmet i seg selv gir gode tilbakemeldinger. I tillegg til dette ble «Checkstyle» og sonarlint brukt for å hjelpe til med å rette opp de mer tekniske feilene. Internett ble brukt en del for å finne ut av teori som kunne hjelpe meg med mulige løsninger. Undervisning, medelever og studentassistenter var også til stor hjelp, men også enkeltpersoner uten programmeringserfaring kom til nytte da jeg ikke hadde mer input å få om koden og trengte noen til finne feil i koden ved å prøve å ødelegge den, for å se om det er noe jeg har oversett.

6 DRØFTING OG RESULTATER

6.1 Valg av datatyper

6.1.1 Klasse Item:

Datatypene som ble brukt var «int», «double», og «string».

Kategori er satt som en «int» etter ønske av oppdragsgiver. Bruker vil imidlertid ikke se kategorinummer når de ønsker å se et item, da en egen ENUM klasse opprettes for å sørge for dette.

```
private final int category;
4 usages
private int inStock;
6 usages
private int price;
6 usages
private int discount;
4 usages
private final double weight;
4 usages
private final double length;
4 usages
private final double height;
4 usages
private final String color;
5 usages
private String itemName;
5 usages
private String itemNumber;
5 usages
private String brand;
5 usages
private String description;
```

Antall varer på lager er satt med datatypen «int». Dette er et tall som aldri vil inneholde bokstaver og datatypen er derfor ikke «string». En vare er heller aldri oppdelt. En stol kan bestå av flere deler, men det går ikke å si at det finnes to og en halv stol. Dermed burde datatypen være «int».

Pris er også satt som «int». Pris kan være et desimaltall, men er satt som et heltall etter kundens ønsker.

Discount har datatypen «int» ettersom det er det vanligste. Det er sjeldent at man kommer borti at noe har en nedsatt pris med 43,4% på et varelager, noe som gjør at man kan se bort i fra «string».

Vekt, høyde og lengde er har alle fått datatypen «double». Vekt er gitt i tall, men er sjeldent et nøyaktig heltall. Om man skulle valgt datatypen «int» her ville man fått tilnærmet lik vekt, men ikke helt presis. Presis vekt kan være viktig når det er snakk om store kvantiter, både for betalende kunder som Smarthus AS har, men også når de selv skal frakte varer, ettersom små feilmarginger kan bli store ved mange nok items. Derfor er datatypen «Double».

«Item name», «brand» og «description» er gitt som "String". Dette er fordi de alle kan inneholde en kombinasjon av tall, tegn og bkstaver.

«Item number» bruker også datatypen «string». Grunnen til dette er at oppdragsgiver fra Smarthus AS har presisert at nummeret skulle bestå av både tall og bokstaver.

6.1.2 Klasse ItemRegister:

I «Itemregister» klassen var datatypen [4.7 Datatyper] som ble valgt «arraylist» for å holde på varers. Grunnen til dette er at jeg ville ha muligheten til å sortere listen, men hovedsakelig fordi «Arraylist» er indeksbasert og kan derfor bruke dette til å finne et spesifikt element.

«HashSet» ble ikke valget da den har mulighet for å legge til og fjerne elementer, men ikke hente ut enkeltelementer uten å vite posisjonen til elementet, og er vanskeligere.

«HashMap» kan lett finne igjen og trekke ut enkeltelementer, i tillegg til at den er mer behagelig for maskinen da den ikke må iterere [Ref. 4.8 Itereringsmetoder] gjennom alle elementer i en list, som den må med «arraylist». Jeg gjorde allikevel valget om å ta «arraylist» ettersom jeg mener at et varehus er avhengig av å kunne sortere varene sine, noe som er komplisert med HashMap.

«Item register» ble det også laget en metode som skulle finne et objekt basert på beskrivelse, ved å sammenlikne stikkord fra et søk med ord i beskrivelser. Med hensyn til det vi vet om disse datatypene [4.7 Datatyper] var det vurdert at «ArrayList» hadde brukt unødvendig mye krefter av maskinen på å iterere gjennom lange avsnitt for å sammenlikne et par ord. «HashSet» kan gjøre dette på en måte som er mye behageligere for maskinen. Dette betyr derfor også at det å sortere elementene, ble mer utfordrende.

Register klassen tok i bruk komposisjon for å dyp kopiere et objekt. Årsaken til at komposisjon ble valgt fremfor aggregering er mange. I registeroppgaver som denne, vil komposisjon gi bedre kontroll over delobjektene da man lettere kan håndtere endringer med dette. I tillegg til dette er det ikke mulig å ta i bruk dypkopiering dersom vi hadde sendt inn et helt fullført objekt, som er hovedårsaken til at det ble sett bort i fra aggregering da dypkopiering kan være viktig i en oppgave hvor klassen skal håndtere mye data som ikke skal kunne bli endret ukontrollert av bruker.

6.2 Mutator-metoder

I klassen «Item» ble det gjort mulig å endre på de fleste verdier utenom kategori, vekt, lengde, høyde og farge, ettersom disse verdiene ikke vil endre seg. Eks. et skap kan få gitt et nytt navn, merke og beskrivelse, pris, rabatt og varer på lager er noe som kontant er under endring og item nummer er noe som veldig sjeldent endres på, men som en sjelden gang allikevel. Vekten på det samme skapet derimot vil være det samme, samme gjelder mål. Om disse endres er det ikke lenger det samme

skapet. Farge derimot er teknisk sett mulig å endre på ved å male over varen, og kunne derfor blitt satt som et mutabelt felt, men som så sjeldent skjer i varehus, at jeg i dette tilfelle vurderer det som unødvendig. Verdiene som ikke er mulige å endre på er immutable verdier, og har derfor ikke mutator/setter metoder. De resterende verdiene er mutable og har derfor alle hver sin mutator metode i item.

I itemregister klassen var alle «Change» metodene mutator metoder, i tillegg til «delete item» metoden og «increase»-, «descreace»- og «set-Instock».

Alle «change»-metodene og «set-Stock» metoden brukte bare mutator-metodene som allerede eksisterte i Item klassen. Grunnen til at register gjorde disse operasjonene, istedenfor å gjøre det direkte med Item klassen er for å opprettholde en robust kode ved å ikke gi rå data til brukergrensesnittet [Ref. 4.1 Robusthet], og for å følge prinsippene kopling og kohesjon av grunner man kan lese lenger opp i rapporten[Ref. 4.3 kopling og kohesjon].

I «increaseStock» og «descreaceStock» mutator metoder. «Increase» og «descrease» metodene bruker mutator metodene fra «Item klassen», men de gjør også en endring utenom disse.

«Delete Item» er også en mutator metode, men denne ble laget i Register klassen. Den henter data fra Item for å sammenlikne, men selve operasjonen som fjerner objektet fra listen skjer i Register.

6.3 Klasser

Klassene som har blitt laget er «Category», «Item», «ItemRegister» og «Main». Navnet på klassene ble gitt slik for å tydelig reflektere hva de ulike klassene holder på av verdier og/eller skal gjøre etter designprinsipp om navngiving [Ref. 4.6 Designprinsipper].

Kategori er Enum klassen og holder på navnet og heltallsverdien som skal representere kategori. Denne Enum klassen har i oppgave å hjelpe til å organisere kategorier. Klassen inneholder en immutabel variabel av typen heltall med navn «CategoryNumber», samt metodene «getCategory», «getCategoryNumber» og «getCategoryName».

«Item» er klassen som lager et objekt av «item», som består av en rekke ulike verdier [Vedlegg 1]. Disse verdiene kan hentes gjennom «get»-metoder i klassen, i tillegg til at de fleste av disse verdiene kan endres ved hjelp av aksess/set-metoder som også er implementert. Klassen inneholder en konstruktør som er det som faktisk oppretter selve objektet Item ved hjelp av verdiene, i tillegg til en dyp kopi konstruktør som skal hjelpe til med å bevare programmets sikkerhet ved bruk av prinsippet «responsibility driven design». Disse konstruktørene har alle krav for hvordan de ulike verdiene skal være når et nytt item blir opprettet, for å gi et mer brukervennlig, men også robust program. «Get-discountedPrice» er også tatt med i item klassen, ettersom pris om det skulle være gammel eller ny, tilhører objektet direkte. En «to string»-metode ligger også i klassen for å lettere kunne hente ut og vise alle verdiene til et Item.

«Itemregister» er klassen som lagrer alle objektene på et sted. Det er gjennom denne klassen «main» vil snakke, for å få tilgang på verdier i item klassen for å opprettholde god kohesjon og kopling [Ref. 4.3 kohesjon og kopling], samt opprettholde et robust design [Ref. 4.1 Robust design]. Denne klassen er hvor alle operasjonene vil skje gjennom ettersom dette er ansvaret/rollen til registeret [Ref. 4.3 cohecion og coupling].

Main klassen, er brukergrensesnittet/selve programmet. Når programmet kjøres, er det «main» som vises for bruker. Main bruker metoder og variabler fra registeret, eller gjennom registeret for å få bruke metoder og variabler fra klassen Item [Ref. 4.1 og 4.3]. Denne klassen har kun en rolle som et vindu for bruker å se koden, eller for programmerere å teste den.

6.4 Prosess

Prosessen gjennom opprettelsen av koden startet med å følge en metode [Ref. 5 Metode]. Etter hvert som programmet ble større og vanskeligere, falt denne metoden litt vekk, da det konstant ble gjort endringer i koden. For hver endring som ble gjort, kunne man lære seg 5 nye ting, når man kun så etter en. Dette første til at en ble litt overveldet av ideer, og gjerne skulle ha gjort mer om tiden hadde strekt til. I tillegg til at kreativiteten ble bedre, ble også evnen til å refaktorere bedre. Det er vanskelig å sette fingeren på nøyaktig hva som ble endret, da jeg bestemte meg for å slette hele koden og starte på nytt, da jeg innså at den kan forbedres mye med all den nye kunnskapen jeg fikk tilegnet meg gjennom å kode. En kan derfor kanskje si at hele prosessen stort sett baserte seg på refakorering, kunnskapstilnærming og masse prøv og feil.

Det ble gjort utallige endringer underveis. Noen av disse var å lage en mer kompleks «toString» da «Discount» kom inn i bildet, da dette ville gjøre systemet mer oversiktlig, ved å ikke vise unødvendig informasjon til bruker, men heller ikke utelate noe viktig.

En annen endring som ble gjort var å endre på navn, da prinsipper som modualisering [Ref. 4.2] ble klarere, og metoder ikke gjorde samme oppgave lenger.

Kopling og kohesjon ble bare viktige og viktigere som programmet ble større, som gjorde at endringer som at «ItemRegister» ble initialisert i en annen klasse enn register som allerede hadde opprettet noe.

Dyp kopi var noe som forsvant ut av tankene under kodingen, og ble derfor noe som ble tatt hensyn til underveis. Denne er kanskje lett å glemme da vi direkte ikke kan teste sikkerheten i praksis, men er ekstremt viktig når det kommer til systemer som holder sensitiv informasjon og generelt for å ha robusthet i programmet [Ref. 4.1 Robust design]

6.4.1 Tilleggsmetoder

Designet på koden og hvilke metoder jeg valgte å implementere utenom det oppdragsgiver ga i beskrivelsen, er sterkt påvirket av erfaring fra å jobbe på lager.

I item klassen ble det ikke gjort mer enn at det ble lagt til en ekstra parameter «navn», ettersom det finnes mange typer dører f.eks og navnet pleier å gjenspeile hva slags dør det er, uten at det skal være nødvendig å lese hele beskrivelsen. Dette er et minimalt inngrep og er lett å fjerne om arbeidstaker skulle ønske det, men utenom dette, gir det flere valg til bruker som mulig vil gjøre det lettere å finne frem blant items.

I item registeret ble det implementert en del ekstra metoder som ikke var krav, men som allikevel kan være viktige.

«Number of different items» ble lagt til som en metode, slik at bruker kan se hvor mange forskjellige typer varer de har inne på lager. Dette kan være viktig med tanke på at det finnes regelverk rundt antall ulike artikler et varehus kan holde på per areal, av sikkerhetsmessige årsaker. Jeg kunne også laget en metode som regner ut hva dette antallet skal være basert på arealet på varehuset, og brukt denne til ulike varehus om det skulle være nødvendig, men ettersom det teknisk sett ikke var nødvendig i utgangspunktet, så er maks antall varer hardkodet, ettersom jeg mener at slik informasjon hadde vært best i holde i en egen klasse «Warehouse» som holder på areal, og dermed begrensning. For å holde oss til de gjeldene metodene, ble dette en tanke som videre kan bli utviklet om ønskelig. I tillegg hadde det vært mulig å legge til hjelpemetoder til denne som finner antall innenfor de ulike kategoriene, som ikke nødvendigvis er like nyttig, men som kan være greit å vite når man ser på hva man ønsker å selge mer/mindre av.

«Total value in stock» er en metode som finner summen av alle varer på lager. Denne er nyttig for regnskapet til bedriften. Metoden er også lagt opp slik at den kan bruker på flere forskjellige lister, dersom man skulle ha flere lager med ulikt antall og typer varer.

Videre er «find» - metodene. Oppgaven var å finne item basert på nummer og/eller beskrivelse, men jeg la opp til flere muligheter ved å legge til at man også kan søke etter navn og kategori. Alle metodene baserer seg på nummer, men tillater bruker å se alle varer innenfor søket sitt først, for så å lettere finne frem til den ønskede varen dersom man ikke skulle kunne item nummeret utenat. På nummer, navn og kategori må man vite ordrett nøyaktig hva de enkelte verdiene er. På beskrivelse er det et unntak. Som regel er beskrivelser mer enn et par ord, og er derfor vanskelig å finne frem ved å skrive inn teksten ordrett. I tillegg til dette er det mye lettere for bruker å skrive feil når setningene blir lenger. Derfor lagde jeg «find by description» litt annerledes. Her skriver du inn stikkord som f.eks. «ytterdør eik produsert Norge» og du vil få alle objektene som har samme ord i beskrivelsene sine. Jeg skulle gjerne også sortert dem etter objektene med flest treff først og færrest nederst, ettersom dette virker mest logisk, men her var det mangel på tid til å

finne ut hvordan jeg bruker «HashSet» på en slik måte at dette er mulig.

«Sort» - metodene er til for å sortere listen med varer etter ulike verdier. Det å generelt sortere varer etter noe, spesielt når det blir flere hundre eller tusen varer på lager, gjør alt mer oversiktlig for bruker. Disse metodene ble bevisst ikke dyp kopiert da alle lister som er mulig å sette inn som parametere, er dypkopiert før de kommer til denne metoden. Om man skulle ønske å dyp kopiere en liste med originaldata, er det ikke mye jobb å gå inn i «sort» metodene og endre på dette.

«Split tekst» ble dannet som en metode for å hjelpe til med sortering, men ble skrevet som en egen metode ettersom den i teorien kan bruker på flere områder, i tillegg til at det står mer i stil med prinsippene om kopling og kohesjon.

«List to string» er en metode som ble lagt til av mer estetiske årsaker. Dette er for at bruker skal få et penere og ryddige display, som gjør alt lettere å lese.

«Deep copy list of items» er strengt talt en mer unødvendig metode da man alltid kan dyp kopi hvert enkelt element i listen ved å iterere gjennom den før den blir sendt videre til bruker. Jeg valgte allikevel å lage den ettersom jeg mener at det blir lettere å kun ha denne itereringen foregå i en metode, sammenliknet med individuelt der det skulle være nødvendig.

Sist har jeg «change» - metodene som endrer på ulike verdier som item bærer, av de verdiene jeg har tillat at skal være mulige å endre på.

6.5 Robusthet

Det har blitt gjort en rekke med tiltak i programmet for å sikre en robust klasse. Ved å følge prinsipper om robusthet [Ref. 4.1 Robust Design], spesielt ved bruk av riktige grunnleggende designprinsipper og prinsipper som kopling og kohesjon.

Ved å bruke debugging og testing under koding av programmet kan man se hvorvidt om programmet fungerer eller ikke, og evt. hvor det stopper, slik at en vet konkret hvor man skal gå tilbake i koden for å gjøre en endring. Debugging gir også en form for oversikt over hvordan Java går gjennom og leser kode, som hjelper til med å se om det har blitt tenkt riktig, når man ser på hvor informasjon egentlig går.

«Exceptions» er også viktig for en robust klasse slik at dersom feil skulle skje, så vil ikke dette påvirke det en prøver å oppnå med koden. Ved å bruke «exceptions» vil ikke programmet stoppe dersom det møter på et forutsatt problem, med å avbryte handlingen for så å gi en feilmelding til bruker med muligheten for å fortsatte. Testing er til stor hjelp til å gjøre koden robust ved å forutse problemer som kan oppstå, ved å prøve seg frem i forsøk om å finne svake sider ved koden.

Kopling og kohesjon har også veldig mye å si for programmets robusthet. Dersom en skulle si at en bruker prinsippene feil, kan f.eks. bruker få direkte tilgang på verdiene log metodene i klassen og kan derfor manipulere originaldata. Dersom dette blir gjort mulig, vil ikke klassen være robust.

Dette henger også med noen mindre grunnleggende prinsipper som bruk av «public» og «private» for lettere kontrollere hva en bruker får tilgang på fra main, samt riktig bruk av mutatorer også for å bevisst kontrollere hva en definerer som tillat i/og mellom klasser og ikke. Dette henger veldig tett sammen med årsaken til at det blir gjort dyp kopier av objekter, nettopp så det er tydelig hva bruker skal ha tilgang på, for å begrense deres evne til å manipulere programmet

6.6 Dokumentasjon

For å oppnå en godt dokumentert kode ble det brukt en del hjelpemidler.

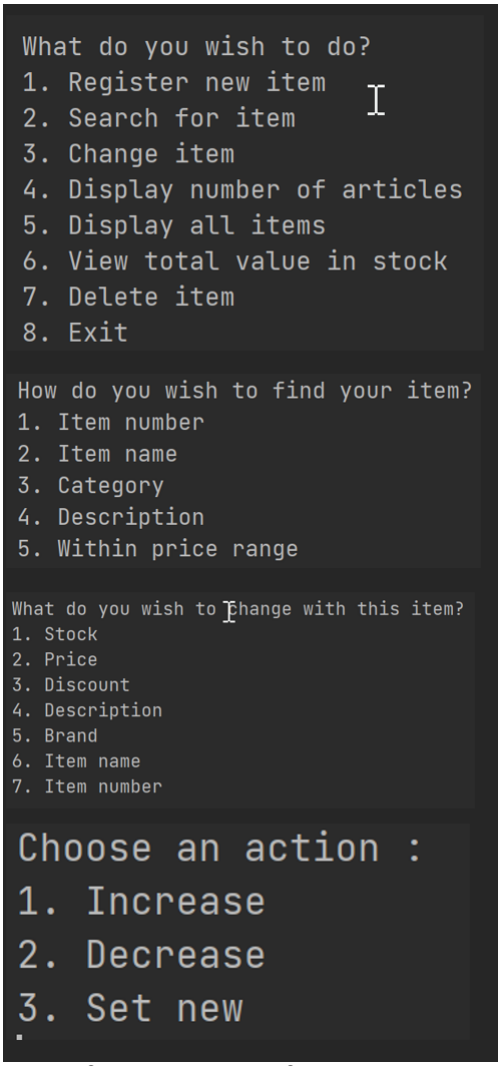
For at selve koden skal være riktig ble «CheckStyle plugin» med «Google checks» stilen brukt, for å sørge for at krav om stil blir etterfulgt. Denne sammen med «Sonarlint plugin» sørget for at mindre tekniske feil, ble ordnet opp.

For å dokumentere hva hver enkelt klasse gjør og holder på, hva hver metode sto ansvarlig for og gjorde, og for å beskrive hva alle de ulike elementene i programmet gjør/betyr generelt, ble det gjort «JavaDoc».

Rapporten i seg selv er også dokumentasjon på hva som har blitt gjort, hvordan og hvorfor støttet opp av teori, samt fremgangsmåte og tabeller, i tillegg til vedlegg som f.eks. Brukerveiledning.

6.7 Brukervennlighet

Brukervennlighet er et viktig punkt da det er bruker som skal bruke programmet. Vi ønsker ikke bare at bruker skal være fornøyde med programmet, men vi vil også at de skal forstå det og kunne bruke det. I koden er brukervennlighet tatt i betraktning på flere områder.



Et tiltak som er tatt i bruk er å gi bruker få, men allikevel mange valg på en måte som ikke overvelder bruker. Dette har blitt gjort ved å tilby en liten meny, som tar deg videre til flere andre menyer, istedenfor å se alt på en gang. Et eksempel på dette kan man se på bilde til høyre som er tatt fra koden, dersom man velger «3. Change item» i meny 1 og «1. Change Stock» i meny 3. Bygger på noe man kan minne om prinsippet modualisering [Ref. 4.2 Modualisering]

Et annen tiltak som er gjort er å oppnå brukervennlighet gjennom likhetsprinsippet [4.6, siste avsnitt]. Dette går ut på at der man finner likhet og møster, vil større forståelde forekomme. Meny 2 på bilde til høyre [Vedlegg 6] bruker dette prinsippet, da dette er den samme menyen bruker vil se hver gang de vil gjøre noe med et objekt. Denne blir brukt når de skal søke etter item, når de skal endre et item og/eller slette et item.

Hva slags feilmeldinger som kommer tilbake til bruker når noe har gått galt er også viktig ved brukervennlighet. Ved å følge prinsipp om robust kode og «graceful termination» [4.1, siste avsnitt], kan vi sørge for at bruker for en behagelig opplevelse av feil, hvor det kommer en tydelig feilmelding som forteller bruker hva den skal gjøre, med muligheten for å gjennomføre det.

Det ble også lagt til flere metoder [5.2.1, Tilleggsmetoder] i programmet som arbeidsgiver ikke krevde, men som allikevel kan være nyttig å ha.

«List to string» var en av disse metodene, og ble laget for å unngå unødvendige tegn i displayet som bruker ser. Dette vil gjøre det litt mer oversiktlig for bruker.

«Find» - metodene ble laget for å gjøre ting lettere for bruker når de bruker programmet. Dette gir bruker flere valg, som kan gjøre prosessen de vil gjennomføre lettere. For eksempel holder det med at man husker en av verdiene navn, nummer, beskrivelse, kategori eller ca. Pris for å kunne finne et item. Uten dette hadde bruker blitt betydelig begrenset og hadde måttet finne andre måter å søke frem objektet om de ikke kan nummeret, men skal endre og/eller slette et objekt fra registeret. I tillegg er disse

ekstra funksjonene lagt opp på en slik måte som gjør at det ikke forvirrer bruker [ref. pkt. 4.6], men allikevel skaper det lille ekstra funksjonalitet.

Brukerveiledningen gir en kort oversikt over alle funksjoner og valg programmet tilbyr, samt hvordan reagere på feilmeldinger som kan dukke opp. Brukerveiledninger ligger som vedlegg [Vedlegg 4] i dette dokumentet.

7 KONKLUSJON – ERFARING

Det endelige resultatet endte i et velfungerende program som følger det mest grunnleggende, men også de viktigste prinsippene i OOP.

Oppgaven som helhet var veldig lærerik. Det å kode og bearbeide kode gjorde at kunnskap som har vært sittende bak i hodet en stund, sakte kom frem igjen. Rapporten var derimot mer lærerik enn jeg kunne forvente meg. Det fikk meg til å stille spørsmålet «Hvorfor» ved hver eneste handling som ble gjort, istedenfor å kun kjøre på autopilot. Det å jobbe med teoretiske prinsipper i dybden på en slik måte, gjorde at jeg fikk en helt annen forståelse av programmet og hvordan ting fungerer.

7.1 Forslag til forbedringer:

Kreativiteten økte mer og mer som innleveringsfristen nærmet seg, og en del forbedringsområder kom til overflaten sent nok til at det ikke var mulig å gjennomføre, men som kunne ha utviklet programmet enda mer.

Til rapporten ser jeg at bruk av diagrammer for å forklare programmet kan være nyttig. Jeg hadde også laget en mye mer spesifikk brukerveiledning som er umulig å misforstå, ved ikke å bare forklare valgene menyen tilba vagt, men forklare hva som kreves av bruker og en forklaring av feilmeldinger samt hvordan bruker skal gå frem for å løse dem om de skulle oppstå.

I programmet kunne jeg for eksempel:

Det ble brukt to «scannere» for å løse oppgaven, noe jeg ikke ville gjort igjen. Til neste program skal jeg heller bruke en skanner og heller «pharse» denne, da dette generelt er bedre kode, men også fordi det gjør det lettere å fjerne andre små «bugs» som oppstår som whitespace feil i input fra bruker ved å kunne bruke trim() som kun er mulig for «strings». Dette hadde f.eks. også gjort det mulig å sette enda strengere krav på input om «itemnumber» som kan være så strengt som at det bare skal bestå av en bokstav, 3 tall og ett punktum.

- «Description» brukte «HashSet» som satte en stopper for hvor vidt jeg kunne sortere beskrivelsene. Jeg skulle gjerne ha hatt tid til å finne ut av hvordan jeg kan få sortert disse varene, ved at

varebeskrivelsen med flest treff i søket havner øverst, nest flest treff på plass nummer to osv.

- Jeg skulle gjerne fått mer tid til å lære hva som krever mer av maskinen og ikke. Men det å bruke break der jeg først gjør iterasjoner og å tenke over valg av datastrukturer, hjelper noe. Men jeg skulle gjerne ha gjort mer.
- Metoder som «delete item» kunne blitt litt bedre ved at spørsmålet som stiller bruker om de er sikre på at de vil slette en vare, sitt sikkerhetsspørsmål, er tilfeldig og auto generert.
- Laget en egen enum klasse for farger, da dette vil hindre småfeil som skrivefeil og vil gjøre det lettere å endre på verdiene.
- Har ubrukte settere i enum klassen, da jeg ikke tenkte på dette før sent. Men ville definitivt funnet ut hvordan man kan fjerne, endre og/eller legge til en ny kategori, til større formål enn det som det har blitt brukt til hittil. Dette gjør større endringer så mye lettere og fører til bedre funksjonalitet.
- Generelt å «fail proof» koden i all sin helhet, ved å tenke på flere ting enn det jeg har gjort hittil. For å bruke mer regex for å styre input, og ikke minst lære å lage egne god regex.
- Metoder som «change», «set password», og «total value in stock» kunne blitt forbedret, ved at det bliver laget som selvstendige metoder som ikke er hardkodet. Om det er mulighet for å unngå hardkoding av informasjon som kan være mutabelt, er det som regel bedre.

8 REFERANSER

[1] https://www.robust-reliability.com/en/robust-design/robust-designb/

[2] Hartman, J. (2022) What is Abstraction in OOPs? Java Abstract Class & Method. Tilgjengelig fra: https://www.guru99.com/java-data-abstraction.html (Hentet 09.12.2022)
[3]https://www.uio.no/studier/emner/matnat/ifi/INF1050/v14/timeplan/inf1050.smidig.22.1.2014.pdf

[4] ”Objects First With Java”, Sixth edition, av Barnes og Kölling. ISBN

[5] Forelesning 5: Kathayat, S. 2022. Samarbeid mellom objekter. IDAT1001 Programmering 1. Tilgjengelig fra: https://ntnu.blackboard.com/ultra/courses/_34267_1/cl/outline (Hentet 6.12.2022)

[6] Bilde tatt fra: https://mechomotive.com/coupling/

[7] http://www.ulven.biz/it2/kompendium/kap14/

[8] https://www.baeldung.com/java-interface-segregation

[9] https://www.drschore.com/fordelene-av-modularisering/

[10] Forelesning 10: Sandstrak, G, 2022. Samarbeid mellom objekter. IDAT1001 Programmering 1. Tilgjengelig fra: https://ntnu.blackboard.com/ultra/courses/_34267_1/cl/outline (Hentet 11.12.2022)

[11] Forelesning 8: Kathayat, S. 2022. Samarbeid mellom objekter. IDAT1001 Programmering 1. Tilgjengelig fra: https://ntnu.blackboard.com/ultra/courses/_34267_1/cl/outline (Hentet 4.12.2022)

9 VEDLEGG

Vedlegg 1, del 1 oppgave

Vedlegg 2, del 2 oppgave

Vedlegg 3, del 3 oppgave

Vedlegg 4, Brukerveiledning

Vedlegg 5, Fremdriftsplan

Vedlegg 6, Programvare «Zipped» -> blir sendt ved som fysisk vedlegg da zip filer ikke kan legges til i word.