| Kandidatnummer(e)/Navn: | | | |
| --- | --- | --- | --- |
| 10120 | | | |
| Dato: | Fagkode: | Studium: | Ant sider/bilag: |
|  | IDATT1001 | Bachelor i ingeniørfag, data - Trondheim | 7 / |

|  |
| --- |
| Faglærer(e) : |
| Muhammad Ali Norozi  Arne Gerhard Styve  Kiran Bylappa Raja  Surya Bahadur Kathayat |

|  |
| --- |
| Tittel : |
| Warehouse management system |

|  |
| --- |
| 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. Dette programmet skulle ha et brukergrensesnitt som ansatte kan bruke for å utføre endringer på systemet. Denne rapporten dokumenterer det teoretiske grunnlaget for hvordan applikasjonen ble utviklet, samt utviklingsprosessen og diskusjon om det endelige resultatet. |

*Denne oppgaven er en besvarelse utført av student(er) ved NTNU.*

INNHOLD

[1 SAMMENDRAG 1](#_Toc116650246)

[2 TERMINOLOGI 1](#_Toc116650247)

[3 INNLEDNING – PROBLEMSTILLING 1](#_Toc116650248)

[3.1 Bakgrunn 1](#_Toc116650249)

[3.2 Formål og problemstilling 1](#_Toc116650250)

[3.3 Avgrensninger 1](#_Toc116650251)

[3.4 Begreper/Ordliste 1](#_Toc116650252)

[3.5 Rapportens oppbyggning 2](#_Toc116650253)

[4 BAKGRUNN - TEORETISK GRUNNLAG 2](#_Toc116650254)

[5 METODE – DESIGN 3](#_Toc116650255)

[6 RESULTATER 3](#_Toc116650256)

[7 DRØFTING 4](#_Toc116650257)

[8 KONKLUSJON - ERFARING 4](#_Toc116650258)

[9 REFERANSER 4](#_Toc116650259)

[10 VEDLEGG 5](#_Toc116650260)

[Denne rapporten inneholder ferdigdefinerte **stiler** som du/dere kan benytte for de mest vanlige avsnittene. Følgende stiler er definert:

Heading 1 Overskrift på nivå 1

Heading 2 Overskrift på nivå 2

Heading 3 Overskrift på nivå 3

Brødtekst Standard tekst i et avsnitt. Benytt denne for all ”vanlig” tekst

Definition Benyttes hovedsakelig i avsnittet ”TERMINOLOGI”

References Benyttes i REFERANSER-avsnittet.

Comment Denne grønne teksten. Fjern all tekst av denne typen i rapporten.]

# SAMMENDRAG

[Et kortfattet sammendrag (abstrakt) av rapporten; om hensikt, oppgave, omfang, framgangsmåte, resultater og konklusjoner.

**Kommentar**: Et sammendrag er som regel lurest å skrive til slutt. Gi her en kort oppsummering av hva rapporten inneholder. For eksempel : Denne rapporten er besvarelsen til obligatorisk innlevering i faget IDAT... Programmering…..]

# TERMINOLOGI

[Definisjoner, begreper og symboler som kan være ukjente for leseren. (Bruk stilen ”Definition”, som vist nedenfor). Utelat dette kapittelet dersom du ikke anvender begreper og symboler som det er behov for å definere.]

UML Unified Modeling Language

//UP Unified Process

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

Refactoring (?) Omskriving av programmet for å forbedre robustheten til koden eller endre funksjonaliteten. (Er dette kanskje et prinsipp mer enn et uttrykk)?

# INNLEDNING – PROBLEMSTILLING

## Bakgrunn/Formål og problemstilling

[Dette er første kapitlet i den faglige rapporten. Det bør behandle bakgrunnen for oppgaven, eventuell oppdragsgiver, problemstillingen og/eller oppgaven som skal løses – og omfanget eller avgrensningen av oppgaven.

**Kommentar**: Det er her du/dere skal presentere selve produktet/problemstillingen som skal løses og eventuelle avgrensninger som gjøres. Merk at det er på den problemstillingen du/dere definere her som resultatdelen og konklusjonen skal vise en løsning for. Dersom oppgaven har fått utdelt en kravspesifikasjon, trenger ikke hele kravspesifikasjonen gjengis her. Henvis i så fall til kravspesifikasjonen, og skriv et sammendrag av kravspesifikasjonen her. Er kravspesifikasjonen relativt kort (1-2 sider) kan den gjerne gjengis i sin helhet her.

Bruk her gjerne **UML-diagrammer** som **Use-Case**, **Aktivitetsdiagram** osv for å beskrive krav til funksjonalitet (NB! Uten å dra inn hvordan du/dere har løst det.)]

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 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.

## Avgrensninger

[Er det gitt noen avgrensninger/begrensninger i oppgaven? Beskriv i så fall disse her.]

## Begreper/Ordliste

[Når man utvikler programvare for en kunde, er det viktig å etablere en felles forståelse for begreper/terminologi/ord som benyttes av/hos kunden. Det er derfor svært vanlig å lage en "ordliste" og/eller en "Domene modell". Denne ordlisten er også et svært godt utgangspunkt for å finne frem til hvilke mulige **klasser** det kan være aktuelt å implementere i løsningen. Bruk tid på denne slik at du har en god forståelse for begrepene.]

| 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 tar innputt fra brukeren for å utføre operasjonene som inventaret tilbyr |

## Rapportens oppbygning

[I vitenskapelige rapporter er det svært vanlig å gi et sammendrag her om hvordan rapporten er bygget opp. Typisk "]

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.

# BAKGRUNN - TEORETISK GRUNNLAG

[Oppgaver og problemstillinger står i en sammenheng. Denne delen skal vise at en har oversikt over denne sammenhengen, at en er eller har gjort seg kjent med tidligere resultater og andres forslag til eller forsøk på løsninger. Det er altså tale om å gi et faglig underlag for ens eget arbeid, evt. en beskrivelse av teoretiske forutsetninger, med referanse til litteratur og andre kilder en støtter seg til.

**Kommentar:** Presenter den teorien som er relevant for de vurderinger som skal gi en god løsning på problemstillingen, som for eksempel teori rundt hvilke metoder som benyttes for å analysere kravspesifikasjon og identifisere gode kandidater til klasser og objekter. Det viktige her er å få fram det teoretiske grunnlaget du/dere senere skal bruke til å vurdere og argumentere for at din foreslåtte løsning er utviklet etter gode designprinsipper og kvalitetskriterier.

Alle vurderinger du/dere gjør senere i besvarelsen skal ha referanse til dette kapittelet. Det er særdeles viktig å ha tydelige referanser til de kildene du/dere bruker når du/dere skriver dette kapittelet. All teori du/dere beskriver her skal altså ha en referanse, og denne skal skrives inn i teksten.

**Eksempel:**

”…et viktig moment ved analysen av problemstillingen er å identifisere kandidater til objekter som senere danner grunnlag for klassene. En mye benyttet metode i følge læreboka [1]…..etc.”

Tilsvarende skal det under referanser være et tall som ramser opp forfatter, årstall, tittel på bok eller artikkel osv. (se punktet om referanser). Forelesninger kan også refereres til, da med tittel på fag og navn på foreleser.]

Typiske teorier i programmering: Hver klasse, **et** ansvar/en rolle Hver metode kun **en** oppgave Felt i klasser **skal** være private Modularisering og abstrahering

# METODE – DESIGN

[Denne delen skal redegjøre for hvordan man planla å gå fram / har gått fram for å løse oppgaven og sannsynliggjøre framgangsmåten. Framgangsmåten kan være en utviklingsoppgave, for eksempel utvikling av et datasystem, databasesystem, en grafikkrutine, et kontrollprogram osv. Ta da også med valg av utviklingsmetode, framdriftsplan, organisering og rapportering, hvordan arbeidet utprøves, kontrolleres og korrigeres, om systemdokumentasjon og brukerveiledning, om overlevering til oppdragsgiver og avtalte kriterier for fullført oppgave osv.]

[Beskriv også hvilke verktøy du/dere har benyttet for å løse oppgaven, hvilket utviklingsmiljø du/dere har jobbet i (BlueJ, Netbeans, Eclipse, IntelliJ, CheckStyle, SonarLint Bitbucket, Git, Wiki, Issue-tracking som JIRA el.l. osv.]

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 (INSERT CITATION). 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. [Her kan vi lyve litt og legge inn at vi brukte et form for diagram for å konkretisere denne samhandlingen.]

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 (dette fører ikke nødvendigvis til høyere coupling, som blir argumentert for i avsnitt 6.) 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.

# RESULTATER

[Dette er rapportens største del. Ved oppgave som omfatter teorigjennomgang, analyse eller teknisk/vitenskapelig undersøkelse: resultater av undersøkelsen - uten vurdering (disse kommer under drøfting). Ved oppgave som omfatter utviklingsoppgave: beskrivelse av løsning, bruksmåte, installasjon, drift og sikkerhet.

Eventuelle UML-diagrammer som klassediagrammer, sekvensdiagrammer osv med tilhørende forklaringer/begrunnelser for valg kan inngå her.

Det er også her viktig å få frem ulike løsninger man har vurdert i prosessen for å komme frem til endelig valgt løsning. Begrunnelse skal gis for hvorfor den ene løsningene ble valgt fremfor den andre.

Beskriv også det endelige resultatet; hva ble til slutt produsert/utviklet i prosjektet? En kort brukerveiledning kan også være på sin plass. Er løsningen stor, kan brukerveiledningen legges ved som et vedlegg og henvises til fra dette kapittelet.]

Hva ble endret i forhold til opprinnelige design contra det siste designet som ble innlevert (**refaktorering**)? Her kan dere benytte klassediagram, sekvensdiagram, osv for å illustrere endringer underveis.

Hva som ble gjort for å oppnå

* robust programvare (bruk av testing, debugging osv, SonarLint)
* godt dokumentert kode (CheckStyle)
* «idiot-sikker» programvare (fail-safe)
* bruker-vennlig design og graceful termination for å ikke krasje program tilfeldig

**Kommentar**: Det er her du/dere skal bearbeide arbeidet ut fra de teorier og metoder som er nevnt i de to foregående kapitlene, og som kan gi et forslag til løsning på den problemstillingen som er definert i innledningen. Merk at det da er nødvendig å gjøre en del henvisninger tilbake til disse to kapitlene for at den som leser rapporten skal kunne følge bakgrunnen for de vurderinger du/dere nå gjør. Husk at du/dere aldri må gjøre vurderinger og analyser uten at dette er dokumentert i teori kapittelet. Ubegrunnet synsing er fullstendig verdiløst. I en oppgave som denne der selve læreprosessen er vesentlig, bør du/dere være flinke til å formulere de tanker og vurderinger som gjøres i selve argumenteringen, altså beskrive både prosess og løsning. Som en huskeregel kan du/dere tenke at normalt har man en tendens til å ikke skrive ned nok rundt selve prosessen med argumentering.]

[**Konkret for Programmeringsemner**:

Her beskriver du/dere hvilke kandidater til klasser du/dere har funnet med bakgrunn i kravspesifikasjonen og begreps-kapittelet. Beskriv også hvilke funksjonalitet som det er stilt krav til i kravspesifikasjonen og hvordan denne er løst. For hver klasse du/dere har identifisert, skriv kort hva som er klassens ansvar/rolle i systemet (gjerne i form av en tabell).]

##### Oppgave 1

Hvorfor valgte du de datatypene du valgte (for feltene til klassen)?

Har du implementert mutator-metoder for noen av feltene? I så fall; for

hvilke felt og hvorfor?

Reflekter/diskuter hvilke tiltak du har innført for å sikre at klassen er

implementert som en robust klasse (en robust klasse er en klasse der det

ikke er mulig å angi ugyldige verdier til feltene i klassen).

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. 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. Det følger også service-oriented-design prinsippet, ettersom at Product sin oppgave kun er å holde på gyldige verdier og det er opp til brukeren å gi gyldige verdier (kanskje litt svakt).

## Del 2

##### Oppgave 2

Beskriv med egne ord hvilke valg du har gjort når du implementerte klassen

som skal representere et varelager/vareregister:

Hvilke klasse fra biblioteket (JDK) valgte du for å holde på varene, og

hvorfor? X

Hva har du gjort for å oppnå løs kobling (loose coupling) i designet

av register-klassen din? X

Reflekter/diskuter hvilke tiltak du har innført for å sikre at klassen er

implementert som en robust klasse (en robust klasse er en klasse der

det ikke er mulig å angi ugyldige verdier til feltene i klassen). X

Brukergrensesnitt/brukerinteraksjon:

Hvordan valgte du å løse brukergrensesnittet til applikasjonen din og

hvorfor?

Hvordan har du valgt å håndtere det at brukeren skriver inn feil

verdier/informasjon? Og hvorfor valgte du å gjøre det slik?

Hvilke øvrige vurderinger har du gjort for å ende på implementert løsning ?

———————————————————————————————————

Som beskrevet i seksjon 5, så ble det bestemt at Inventory-klassen er den eneste metoden som kan inneholde og endre på produkter. Klassen må altså ha en samling som holder på Product-objekter, og den må ha metoder som tar inn informasjon om hvilke produkter den skal endre på. 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, henholdsvis å hente og å legge til et objekt i samlingen. <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html> Det vil si at det tar kun en operasjon for å hente elementet du vil søke etter i HashMapet. I ArrayList er søketiden lineær [https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html). Det vil si at du må på det meste bruke like mange operasjoner som det finnes elementer i listen for finne elementet du søker etter.

Måten HashMap oppnår konstant søketid på, er ved at en tildeler hvert element en nøkkel. 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 med denne typen samling, og det gir derfor mening å bruke det i dette programmet.

Som nevnt, så er det kun Inventory-klassen som har funksjonalitet for å endre på Product-objektene. Dette følger prinsippet om cohesion ettersom at funksjonaliteten til å endre på inventaret ligger i klassen som representerer konseptet om inventaret. Det medfører også loose coupling mellom Inventory og brukere av klassen, siden brukeren kun har mulighet til å utføre handlinger som er definert av Inventory.

Det er også Inventory som lager Product-objektene som blir lagt til i samlingen. Dette medfører komposisjon for håndteringen av Product-objektene, altså at levetiden til disse objektene er avhengig av levetiden til Inventory-objektene. Når Inventory returnerer informasjon om produkter til andre klasser, så gir den dyp-kopier av objektene. Dette sikrer at andre objekter ikke kan endre på produktene som ligger i inventaret direkte, som øker robustheten til programmet.

Siden andre klasser ikke kan endre på produkter i Inventory direkte, så må de ha en måte å peke på hvilke produkter Inventory skal behandle. Det ble valgt at id’en til et produkt skal være unik, og en kan derfor bruke id’en til å referere til et spesifikt produkt. Noen som bruker klassen må derfor kunne hente ut en id. I dette programmet, så skal en bruker kunne søke etter et produkt basert på id og beskrivelse. Designet falt da på at metoden for å hente

Inventory inneholder metoder som lar brukere av klassen finne id’er til produktet basert på søk etter id og beskrivelse. Siden en må gi den id’en, så kan inventaret bruke den til å sjekke om det er et eksisterende produkt, og utløse et unntak hvis det ikke eksisterer. Dette garanterer at brukere av inventar klassen ikke kan utføre ugyldige handlinger på lageret. Siden inventar klassen kun utfører handlinger som er relatert til å endre eller hente informasjon om inventaret, så har klassen høy cohesion og deltar i modulariseringen av programmet for øvrig. Den oppnår også lav kobling ved at den ikke er avhengig av andre klasser for å utføre handlingene på inventaret.

Det ble valgt 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.

En utfordring med å hente informasjon fra terminalen er å validere at verdier er gyldige. Dataene en henter fra terminalen kommer i form av tekst. Når brukeren av programmet skal gi verdier til f.eks. prisen til et produkt må konvertere teksten til en tallverdi. Utfordringen med dette er at det er fullt mulig for brukeren å skrive inn noe annet enn et tall når en spør om tallverdier. 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 innputten til brukeren skjer i både Client og Inventory.

Client er anvsarlig for at brukeren sitt innputt kan gjøres om til verdier som kan bli brukt direkte i behandlingen av et Product-objekt. Dette innebærer at Client må ta seg av å konvertere brukeren sin innputt, som kommer i form av tekst, til tallverdier som kan brukes av Inventory. Inventory er så ansvarlig for å forsikre at innputten 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. Spesielt for Inventory, så følger dette prinsippet om cohesion.

Metodene Java tilbyr for å konvertere tekst til tallverdier utløser exceptions (unntak) hvis argumentet ikke kan konverteres til en tallverdi. Dette vil kræsje programmet. En av målene som ble satt for programmet er at det ikke skal kræsje når brukeren gjør noe feil, altså graceful termination. En må derfor håndtere disse unntakene når en tar disse metodene i bruk for å implementere dette. Metodene

Løsningen for å gjøre implementasjonen av funksjonaliteten i klient robust i henhold til feilsikkerhet, var at søkefunksjonen, som garanterer at id’en til et produkt den returnerer eksisterer i Inventory, blir brukt for alle metodene i klient som omhandler et enkeltprodukt.

For å gjøre søkefunksjonen brukervennlig, så ble det gjort to valg om hvordan den skal: den første er at brukeren trenger ikke å skrive inn hele id’en, og å søke etter beskrivelse består av å matche enkeltord i brukeren sitt innputt med enkeltord i beskrivelsen til et produkt. Den andre er at brukeren ikke må velge å enten bruke en metode for å søke etter id eller en for å søke etter beskrivelse, men at søkene på begge skjer i en metode.

Implementasjonen av søkemetoden ble

Fullføre feilhåndtering.

Hvordan Client bruker Inventory.

Øvrig design.

Implementering av enum.

Vurderinger rundt refactoring.

Enkeltpunktene på del 3.

Skrive om feilhåndtering av innputt når en skal endre på eller legge til et produkt.

For brukergrensesnittet valgte jeg å bruke terminalen til å hente innputt fra brukeren og for å vise informasjon. Siden vi ikke har lært å lage grafiske grensesnitt, så anså jeg terminalen som den beste måten å lage grensesnittet på.

Feilhåndtering av innputt skjer i klienten og i produkt klassen. Klienten sjekker at det brukeren skriver inn passer til typen som den verdien skal bli brukt til (e.g. hvis brukeren har skrevet en streng der det skal være en int.) Å sjekke at verdiene til produktet ikke blir satt til noe ugyldig skjer i produkt klassen. Dette fører til lavere kobling og enklere gjenbruk av koden, siden noen som bruker klassen ikke trenger å innføre sjekker for ugyldige verdier selv, men trenger bare å håndtere unntaket som blir utløst. Hvis hva som er gyldige verdier endrer seg, så trenger en bare å oppdatere produkt klassen. (Ikke helt sant. Hvis det virkelig skulle vært fullstendig avkopling, så måtte informasjonen om hva som er gyldige verdier som skal printes ut hentes fra produkt klassen, likt som den formaterte strengen.)

##### Oppgave 3

Med utgangspunkt i opprinnelig kravspesifikasjon, hvilke endringer/forbedringer

ville du ha gjort for at applikasjonen skal bli enda mer nyttig for brukeren? Her

har du lov til å gå bort fra opprinnelig kravspesifikasjon og tilføre dine tanker og

ideer. Løsningen din må selvsagt fortsatt oppfylle de grunnleggende

funksjonelle brukerkrav, men du står fritt til å endre på designet av klasser, og

legge til ny funksjonalitet.

Brukergrensesnitt: du må gjerne tenke alternativt her. Feks, er en meny beste

løsning (tatt i betraktning at dere ikke har lært å programmere grafisk

brukergrensesnitt enda)?

Dersom SmartHus ønsker å legge til en ny kategori av varer (feks,

Hagemøbler) hvor mye av koden din må du endre? Hva tenker du dette

forteller om ditt valgte design?

Hvordan møter din løsning prinsippet om lagdelt arkitektur?

Beskriv i rapporten hva du har foreslått av endringer og hvordan du har valgt å

implementere disse.

# DRØFTING

[Vurdering av metode og oppnådde resultater. Begrensninger, endringer eller avvik i prosjekt i forhold til plan / opprinnelig problemstilling - mulige feilkilder. Resultatenes betydning.

**Kommentar**: Her kan man for eksempel gjøre seg tanker rundt kvaliteten av det arbeidet som er nedlagt. Er de kildene du/dere bruker pålitelige, er det sprik mellom forskjellige kilder (og i så fall hvorfor), er det andre forhold som kan være med å gjøre noen av de vurderinger og valg du/dere har gjort usikre?]

[Konkret for programmeringsemner: Her oppsummerer du/dere oppgaven. Hvor langt kom du/dere (resultat)? Hva fikk du/dere ikke gjort i forhold til oppgaveteksten ? Hva var de store utfordringene/problemene du/dere møtte, etc..

Spesielt viktig er det å drøfte din egen løsning i forhold til det du har lært om gode prinsipper for programmering (robust kode, kodestil, designprinsipper osv) som beskrevet i teori-kapittelet]

Brukergrensesnitt: du må gjerne tenke alternativt her. Feks, er en meny beste

løsning (tatt i betraktning at dere ikke har lært å programmere grafisk

brukergrensesnitt enda)?

Dersom SmartHus ønsker å legge til en ny kategori av varer (feks,

Hagemøbler) hvor mye av koden din må du endre? Hva tenker du dette

forteller om ditt valgte design?

Hvordan møter din løsning prinsippet om lagdelt arkitektur?

Drøfte at det er mulig å gi et argument for søketiden er lav, men at det er vanskelig å finne kilder.

Den høyere hastigheten for å hente et element i et HashMaper dessuten ikke nødvendig i dette programmet. Antallet elementer som skal ligge i samlingen er så liten at søketiden gjennom listen ikke kommer til å være merkbart for brukeren. Moderne datamaskiner kan søke gjennom lister på flere millioner elementer på millisekunder (INSERT CITATION). En rimelig antagelse er at et varehus på det aller meste kommer til å ha ulike varetyper målt i titusener (INSERT CITATION). The Home Depot, verdens største Det en sparer i beregningstid veier ikke opp for den mer kompliserte implementasjonen bruken av et HashMap vil føre til.

Var det egentlig lurt å gjøre hver del for seg?

Gjorde en grei jobb med å følge prinsippene, men slet litt med å forstå hva de var. Burde ha fått en grundig forståelse for de før jeg begynte.

Fullførte hele oppgaven. Kanskje litt mere på refactoring. Men så egentlig ikke så mye tegn til forbedring.

Kunne vært lurt å bruke flere diagrammer.

# KONKLUSJON - ERFARING

[Overbevisninger /erfaring som en er kommet fram til på grunnlag av det presenterte materialet.

* Hva ville du ha gjort annerledes dersom du kunne begynn på nytt?
* Hva slags begrensninger kan en forvente når en bruker løsningen?
* Hva skal tas opp i fremtidige arbeid dersom du eller noen andre ville ha tatt utvikling videre?

**Kommentar**: Her skal du/dere presentere de viktigste resultatene fra arbeidet sammen med de erfaringer du/dere har opparbeidet i prosessen.]

Jeg ville ha laget skjelettløsninger for eventuelle endringer, spesielt mtp. Det å kunne endre på alle feltene i forhold til kun de som ble gitt i oppgaven, for å vurdere om det ekstra arbeidet er verdt det.

Designprinsippene har vært nyttige til å forme koden på en konsekvent måte.

Brukeren kan forvente å måtte å begynne på nytt med enkelte oppgaver hvis de gir feil verdier

Videre utvikling ville gått på å forbedre brukeropplevelsen ytterligere ved å diskretisere enkelte funksjoner. F.eks. Å kunne velge spesifikke felter de vil endre på.

Mere konkret testkode underveis.

# REFERANSER

[Forfatter, årstall, tittel på bok eller artikkel, navn på tidsskrift eller forlag/utgiver, nr. eller dato for tidsskrift, sted som det vises til eller refereres fra i oppgaven.

**Kommentar**: se eksempel under]

[Konkret for programmeringsemner: Regner med at du/dere kommer til å måtte slå opp litt i læreboka, så den er en innlysende referanse. Dersom du/dere i tillegg benytter internett, så list URL’er til sidene du/dere har benyttet.]

1. ”Objects First With Java”, Sixth edition, av Barnes og Kölling. ISBN ….
2. http://.....
3. Osv.

# VEDLEGG

[Materiell som er utarbeidet eller innsamlet i tilknytning til rapporten, men som det ikke er naturlig eller hensiktsmessig å ta inn i hoveddelen, skal tas inn som vedlegg.

Vedleggene skal være nummererte og ha en overskrift.

Har du/dere ingen vedlegg, så droppes dette kapittelet.]