| Kandidatnummer(e)/Navn: | | | |
| --- | --- | --- | --- |
| 10124 | | | |
| Dato: | Fagkode: | Studium: | Ant sider/bilag: |
| 12.12.22 | IDATT1001 | Programmering 1 | 4 / |

|  |
| --- |
| Faglærer(e) : |
| Muhammad Ali Norozi |

|  |
| --- |
| Tittel : |
|  |

|  |
| --- |
| Sammendrag: |
| Skriv et kort sammendrag av hva du/dere har utviklet og hva denne rapporten handler om |

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

INNHOLD

[1 SAMMENDRAG 1](#_Toc121779980)

[2 TERMINOLOGI 1](#_Toc121779981)

[3 INNLEDNING – PROBLEMSTILLING 1](#_Toc121779982)

[3.1 Bakgrunn/Formål og problemstilling 1](#_Toc121779983)

[3.1.1 Del 1 1](#_Toc121779984)

[3.1.2 Del 2 1](#_Toc121779985)

[3.1.3 Del 3 2](#_Toc121779986)

[3.2 Avgrensninger 2](#_Toc121779987)

[3.3 Begreper/Ordliste 2](#_Toc121779988)

[3.4 Rapportens oppbygning 4](#_Toc121779989)

[4 BAKGRUNN - TEORETISK GRUNNLAG 4](#_Toc121779990)

[4.1 Coupling 4](#_Toc121779991)

[4.2 Cohesion 4](#_Toc121779992)

[4.3 Design Pattern 4](#_Toc121779993)

[4.3.1 Builder Pattern 5](#_Toc121779994)

[4.4 Modularisering 5](#_Toc121779995)

[4.5 Abstrahering 5](#_Toc121779996)

[5 METODE – DESIGN 6](#_Toc121779997)

[6 RESULTATER 7](#_Toc121779998)

[7 DRØFTING 8](#_Toc121779999)

[8 KONKLUSJON - ERFARING 9](#_Toc121780000)

[9 References 9](#_Toc121780001)

[10 VEDLEGG 10](#_Toc121780002)

[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

WMS Warehouse Management System

IDE Integrated Development Environment

# INNLEDNING – PROBLEMSTILLING

## Bakgrunn/Formål og problemstilling

Oppgaven var å utvikle en programvare som skulle brukes av et varehus. Programvaren skal kunne brukes til å håndtere varelageret til varehuset som benytter programmet. Programmet skulle til slutt bestå av et register som lagrer informasjon om de forskjellige produktene på lager, og skulle kunne brukes gjennom et tekstbasert brukergrensesnitt. Det skal være mulig å gjøre enkle operasjoner på registeret som gjenspeiler varehus funksjoner i den virkelige verden.

Programmet skulle utvikles igjennom 3 iterasjoner, en del 1, 2 og 3, hvor hver del har sine egne definerte kravspesifikasjoner.

### Del 1

I denne delen skulle man implementere en entitetsklasse som skulle representere en vare. Kravspesifikasjonen for denne delen handlet om informasjonen som skulle registreres, med spesifisert datatype.

### Del 2

I denne delen skulle man implementere et vareregister for å holde på en eller mange varer. Registeret skulle inneholde enkle funksjoner for å modifisere varene i registeret. I tillegg skulle et brukergrensesnitt implementeres for å kunne bruke registeret og funksjonene i registeret. Kravspesifikasjonen for denne delen handlet om de enkle funksjonene som registeret minst skulle ha.

### Del 3

I denne delen skulle man ferdigstille applikasjonen og endre på tidligere kode for å optimalisere den. I tillegg skulle man sette sitt eget preg på løsningen ved å frigjøre seg litt fra de tidligere nevnt kravspesifikasjonene.

[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.)]

## Avgrensninger

Oppgaven spesifiserer datatyper som felt variablene i vare klassen skal ha. Dette er for eksempel at pris skal være heltall, vare nummer skal være en streng med både tall og bokstaver, høyde skal være desimaltall osv. I tillegg skulle koden verifiseres av en Checkstyle-plugin, som sammenligner koden med en kode-standard.

[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) | Betydning/beskrivelse |
| --- | --- | --- |
| Produkt | Product | Varehuset skal håndtere produkter inn og ut av et lager |
| Lager | Storage | Oppbevaringssted for varer |
| Register | Register | Liste over oppgitte varer |
| Brukergrensesnitt | User Interface | Et grensesnitt for å kommunisere mellom bruker og program |
| Iterasjon | Iteration | Utføre en handling flere ganger |
| Klasse | Class | Representerer en gjenstand, et objekt eller en funksjon i den virkelige verden |
| Datatype | Datatype | Beskrivelse av hva slags type en verdi har |
| Variabel | Variable | En verdi som kan variere |
| Kobling | Coupling | Et prinsipp som handler om koblingen mellom klasser |
| Sammenheng | Cohesion | Et prinsipp som handler om hvordan forskjellige funksjoner henger sammen i en klasse |
| Utvidelse | Plugin | En ekstern utvidelse som har en bestemt funksjon |
| Modul | Module | En pakke som inneholder klasser som henger sammen |
| Parameter | Parameter | En verdi som en metode bruker for å utføre funksjonen sin. X-en i F(x). |
| Objekt | Object | En forekomst av en klasse |
| Statisk | Static | Noe som ikke endrer seg |
| Statisk variabel | Static variable | En variabel som er delt mellom alle objekter av en klasse |
| Metode | Method | En funksjon som tilhører et objekt |
| Felt variabler | Field variables | Variablene som defineres av en klasse og som tilhører objekter av klassen |
| Design Mønster | Design Pattern | En generell mal som løser vanlige problemer |
| Abstrakt | Abstract | En klasse som har generelle felt variabler og metoder |
| Mutator metode | Mutator method | En metode av en klasse som kan endre på et felt variabel |
| Aksessor metode | Acessor method | En metode av en klasse som henter informasjon fra klassen sine felt variabler |
| Enum klasse | Enum klasse | En klasse som inneholder statiske definisjoner som tilhører klassen |
| Kompromittert | Compromised | Har blitt gjort sårbar |
| Refaktorisering | Refactoring | En endring på tidligere kode |
| Streng | String | En rekke bokstaver eller tall |
| Heltall | Integer | Et helt tall |
| Unntak | Exception | En uforventet feil som har skjedd |
| Null | Null | Regnes som «ingenting» av datamaskinen, forskjellig fra tallet 0 |
| Strøm | Stream | En strøm med informasjon |
| Grensesnitt | Interface | Noe som kobler sammen to systemer |
| Tilførsel | Input | Det som blir gitt til programmet fra eksterne kilder |
| Utgang | Output | Det som blir returnert av funksjoner |
| Indeks | Index | Representerer posisjonen av et objekt i en liste dersom objektene er i rekkefølge |
| Boolsk | Boolean | En verdi som enten er sann eller falsk |

## Rapportens oppbygning

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

# BAKGRUNN - TEORETISK GRUNNLAG

## Coupling

Coupling er et abstrakt konsept som er graden av uavhengighet mellom moduler. Kan ses på som sannsynligheten for hvor mye en endring eller modifisering av en modul, påvirker hvor mye en programmerer må ta for seg en endring i en annen modul. Hvis denne sannsynligheten er høy, så er disse to modulene «highly coupled», som vil si at en endring i den ene modulen påvirker sterkt den andre modulen. Man vil ofte oppnå «loose coupling», som betyr at moduler ikke er sterkt avhengig av hverandre, og at man kan endre på koden uten å påvirke andre moduler. Dette fører til mer uavhengighet og gjør det lettere for en programmerer å implementere nye funksjoner eller fjerne eksisterende funksjoner.

Noe som påvirker coupling kan være en referanse i en klasse til en annen klasse. En intermodulær kobling oppstår når referansen skjer mellom to separate moduler. I tillegg kan kompleksiteten av koblingen være korrelert med høyere coupling. Dette vil si at en kobling fra en klasse som spesifiserer mange parametere har høyere coupling enn en kobling med få parametere. (Yourdon & Constantine, 1975)

## Cohesion

Måten man deler opp et system, i sammenheng med problem strukturen, kan påvirke struktur kompleksiteten og antall intermodulære referanser. Derfor er det viktig og ta hensyn til at man deler opp i moduler som henger godt sammen med funksjonen den ønsker å oppnå. Dette kalles cohesion. Dette gjelder også for funksjonaliteten av individuelle metoder i en klasse. I den virkelige verden kan et isolert element være funksjonelt relatert til en variert mengde av andre elementer. Dette leder ofte til en subjektiv tolkning av problem strukturen, og det oppstår derfor forskjellige løsninger på hvor mye forskjellige elementer henger sammen. Likevel kan man si at hvis alle metoder gjør den spesifikke oppgaven den er ment til å gjøre, så gir dette «høy cohesion». En ulogisk gruppering av funksjoner i en metode kan man mene er «lav cohesion».

Man ønsker ofte å oppnå høy cohesion. Dette er fordi det leder ofte til en bedre organisert struktur på koden, og betyr at det er lettere å finne frem til metodene som gjør en spesifikk oppgave. I tillegg oppnår man høy cohesion når en klasse inneholder metoder som gir logisk mening for den klassen å ha. (Yourdon & Constantine, 1975)

## Design Pattern

Et design mønster i henhold til programmering er en generell løsning til problemer som ofte oppstår under strukturering og programmering av en applikasjon. Disse er enten en forklaring eller en mal på hvordan man kan løse et problem, og er effektivt når man designer et program eller et system. Subkategorier innenfor design mønstre er for eksempel skapende-mønster, struktur-mønster, oppførsels-mønster osv. (Gamma, Helm, Ralph, & John, 1994)

### Builder Pattern

Builder pattern er et design mønster av subkategori skapende-mønster. Dette vil si at design mønsteret er en løsning på hvordan et objekt er skapt. Intensjonen med å bruke builder pattern er å separere konstruksjonen av et objekt fra representasjonen av objektet, og konstruksjon prosessen kan også derfor brukes til å lage flere representasjoner av objektet. Dette øker modulariteten ved å isolere koden for konstruksjon av objektet fra representasjonen.

Dette design mønsteret kan brukes når algoritmen for å lage et komplekst objekt burde være uavhengig av delene som utgjør objektet og hvordan den konstrueres. Det kan også brukes hvis konstruksjon prosessen tillater at fins flere forskjellige representasjoner av et objekt. (Gamma, Helm, Ralph, & John, 1994)

## Modularisering

Modularisering er en teknikk for å dele et program eller et system opp i mindre, uavhengige deler som kalles moduler. Dette innebærer å identifisere ulike funksjoner eller komponenter, og deretter skille dem ut i egne moduler, hvor hver modul har et definert formål og ansvar. Disse kan kommunisere med andre moduler gjennom intermodulære referanser. På denne måten kan man lett bytte ut eller endre enkelte moduler uten å påvirke resten av systemet.

Modularisering kan også bidra til å øke samarbeidet mellom programmerere ved å gjøre det enklere å dele og koordinere arbeidet på ulike deler av et program eller system. Dette gjør det også mulig og teste og kvalitet sikre individuelle moduler, og kan føre til økt kvalitet og stabilitet.

Dersom en del av et system blir kompromittert kan dette ha store konsekvenser for resten av systemet. Derfor kan modularisering bidra til å øke sikkerheten ved å begrense skadeomfanget dersom det skulle skje noe med en modul. Om bare en modul skulle bli påvirket, blir det lettere for en programmerer å rette opp i problemet.

Ved modularisering av et system oppnår man mer oversiktlighet, en enkel, forståelig og vedlikeholdbar kode, og mer fleksibilitet for å gjøre endringer eller legge til nye funksjonaliteter. Moduler kan også gjenbrukes i andre programmer eller systemer, som kan spare tid og arbeid. (Gosling, Bill, Guy, & Gilad, 2005)

## Abstrahering

Abstrahering i programmering referer til evnen til å samle likheter mellom ulike objekter eller konsepter og definere en generell modell eller struktur som kan følges av objektene. Dette gjøres ved å identifisere de viktigste egenskapene og metodene et objekt kan ha, og deretter lage en abstrakt klasse som inneholder disse felles egenskapene og metodene. Dette gjør det mulig for klasser som kan grupperes som et objekt av den abstrakte klassen til å arve fra denne klassen. Abstrahering gjør det mulig for programmerere å være mer generelle og fleksible, som gjør det enklere å lage og vedlikeholde koden. For eksempel så kan man ha en abstrakt klasse «Dyr», hvor klassene «Løve», «Hund» og «Elefant» arver fra «Dyr» klassen. I dette tilfellet så ville den abstrakte klassen inneholde metoder og felt variabler som er felles for alle dyr, og klasser som arver av denne klasse kan spesifisere mer konkret noen abstrakte metoder som for eksempel en løpe-metode, eller en spise-metode. (Abstraksjon (datavitenskap), 2022)

## Funksjonell Programmering

Generelt så er funksjonell programmering en paradigme, og handler om å programmere med funksjoner. I mange programmeringsspråk er dette oppnådd med lambda operatorer og streams. Å implementere funksjonell programmering i Java betyr å implementere en funksjonell grensesnitt klasse, som betyr at grensesnitt klassen kun har en abstrakt metode. Fordeler med funksjonell programmering er blant annet at programmer som bruker funksjonell programmering er deterministisk, som gjør det lettere å forstå. Tilførsel til programmet vil alltid gi det samme resultatet. Funksjonelle programmer er også modulære fordi de er oppbygde av funksjoner som kun har en input og output.

Iterering gjennom en liste foregår ofte gjennom en for-løkke i de fleste programmeringsspråk, som er en ekstern iterasjon fordi man itererer over en liste. Streams handler om å kunne la listen i seg selv gi ut hvert element i listen, en indre iterering. (Saumont, 2017)

[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
* Osv.

# METODE – DESIGN

Hele koden har blitt kodet i IDE’en «InteliJ». Dette er en IDE som ble utviklet for å programmere i Java. Github har også blitt benyttet for å dokumentere koden underveis, og for å gjøre sikre endringer på koden uten risikoen for at alt blir kompromittert. Dette er også viktig ved flere refaktoriseringer for å opprettholde versjonskontroll. Applikasjonen har blitt utviklet ved å følge fossefall utviklings modellen gjennom hver iterasjon av oppgaven. Framgangsmåten startet med å analysere kravene som problemstillingen spesifiserte, og deretter opprette et design i forhold til kravene. Designet ble opprettet som et klasse diagram, som gjør det veldig enkelt å implementere. Vedlegget under illustrerer det første utkastet av klasse diagrammet som ble designet etter analyse av oppgaven.

Etter implementasjonen så ble koden verifisert ved å bruke JUnit test klasser. JUnit er en plugin som kan brukes i InteliJ for å danne test klasser som kan verifisere funksjonaliteten til metoder og konstruktører av klasser. I tillegg ble det brukt en Checkstyle-plugin for å sammenligne koden med en kode-standard. Feil funnet i enten JUnit testingen eller Checkstyle ble rettet opp på før videre utvikling av programmet. Disse stegene ble utført for hver iterasjon av oppgaven helt til det ferdige produktet ble ferdig.

[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.]

# RESULTATER

## Item

Item klassen er klassen som skal representere en virkelig-verden vare. Den inneholder felt variabler som ble definert av oppgaven, i tillegg til egen definerte variabler. I tillegg har den mutator og aksessor metoder for disse variablene, og to konstruktører.

### Felt variabler

Alle definerte felt variabler er private for at man ikke skal kunne endre på variablene etter konstruksjon av objektet, med mindre man bruker de eksisterende mutator metodene. Dette fører til loose-coupling. Variabler med nøkkelordet «final» gjør at variabelen ikke kan endres på etter konstruksjon av objektet.

itemNumber En streng variabel som representerer vare nummberet til varen. Denne variabelen har datatypen String fordi et varenummer skal kunne inneholde en kombinasjon av bokstaver og tall. Den har også nøkkelordet final fordi varenummeret skal være ferdig definert.

description En streng som representerer vare beskrivelsen. Denne variabelen har datatypen String fordi en beskrivelse er en tekst.

brandName En streng som representerer merkenavnet til varen. Denne variabelen har datatypen String fordi et navn er en tekst. Den har også nøkkelordet final fordi merkenavnet til en vare endrer seg ikke over tid.

price Et tall som representerer prisen til varen. Denne variabelen har datatypen int fordi prisen skal være et heltall, og siden int har en maks verdi på (2^31) -1, så passer den bra fordi det er usannsynlig at prisen blir så høy

warehouseStock Et tall som representerer antall varer det er på lager. Denne variabelen har datatypen int fordi et antall burde være heltall når det skal representere virkelige objekter, når disse ikke kan være desimaltall. Int sin maks verdi gjør det også passende dersom det er usannsynlig at denne verdien oversteges.

discount Et desimal tall som representerer en rabatt oppgitt i prosent. Denne variabelen har den primitive datatype double, fordi det skal kunne lagres desimaltall i denne variabelen.

weight Et desimal tall som representerer vekten til varen i kilogram. Denne variabelen har den primitive datatypen double fordi den skal kunne ha presise målinger av vekt. I tillegg er variabelen final fordi det er ulogisk at vekten endrer seg.

length Et desimal tall som representerer lengden til varen i meter. Denne variabelen har den primitive datatypen double fordi den skal kunne ha presise målinger av lengden. I tillegg er variabelen final fordi det er ulogisk at lengden endrer seg.

height Et desimal tall som representerer høyden til varen i meter. Denne variabelen har den primitive datatypen double fordi den skal kunne ha presise målinger av høyden. I tillegg er variabelen final fordi det er ulogisk at høyden endrer seg.

width Et desimal tall som representerer bredden til varen i meter. Denne variabelen har den primitive datatypen double fordi den skal kunne ha presise målinger av bredden. I tillegg er variabelen final fordi det er ulogisk at bredden endrer seg.

color En variabel som representerer fargen til varen. Denne variabelen har datatypen Color, som er en egendefinert enum klasse for spesifiserte farger. Den har også nøkkelordet final fordi fargen ikke endrer seg over tid.

category En variabel som representerer kategorien til varen. Denne variabelen har datatypen Category, som er en egendefinert enum klasse for spesifiserte kategorier. Den har også nøkkelordet final fordi kategorien ikke endrer seg over tid.

### Konstruktør

Den første konstruktøren tar alle felt variablene i Item klassen som parameter for å konstruere et objekt med de gitte verdiene for hver variabel. Begge konstruktørene er pakke beskyttet, som vil si at kun klassene i samme pakke har tilgang til konstruktørene. Dette er videre diskutert i avsnittet om ItemBuilder klassen.

Denne konstruktøren kaster også en NullPointerException hvis enten vare nummeret, fargen eller kategorien oppgitt er veriden «null». Dette er et unntak som brukes når man prøver å bruke et objekt som er «null». Konstruktøren kaster derfor NullPointerException for å sikre at verdiene ikke kan være «null» for å unngå problemer som kan oppstå av det. De primitive datatypene kan ikke være null uansett, så disse sjekkes ikke. I stedet blir variablene for pris og antall varer testet for om verdien er under 0, altså negativ. Dette er fordi det ikke gir logisk mening å ha en negativ pris, eller negativ antall varer på lager. Legg merke til at vi tillater prisen til å være 0 her. Dette er fordi man kanskje har en gratis vare på lager. Variablene for vekt, lengde, høyde og bredde kan ikke være negativt eller 0. Dette er fordi det ikke gir logisk mening å tillate dette. Om dette skulle skje, blir en IllegalNumberException kastet. Dette er et egen definert unntak for når det er ulovlig tall som har blitt oppgitt. Testene for pris og antall varer kaster også IllegalNumberException hvis det blir oppgitt et negativ tall. Ellers hvis ingen unntak blir kastet, får alle felt variablene de spesifiserte verdiene som har blitt oppgitt, bortsett fra variabelen for rabatt (discount). Dette er fordi det ikke gir logisk mening å kunne sette rabatt med engang, så derfor får denne variabelen verdien 0 ved konstruksjon.

Den andre konstruktøren tar en annen vare av klassen Item som parameter for å konstruere en dyp-kopi av den oppgitte varen. En dyp-kopi er en kopi av et objekt med de samme verdiene til de samme variablene, men ulik minneplassering. Dette betyr også at en «.equals()» metode vil returnere falsk. Med dyp-kopiering kan man ofte oppnå loose-coupling, fordi man ikke returnerer det originale objektet, men en kopi, sånn at ingen endringer på det kopierte objektet påvirker det originale.

Denne konstruktøren sjekker om den gitte varen i parameteret er verdien «null» for å sikre at ingen feil kan komme av at den er null. Ellers får alle variablene verdiene til den oppgitte varen sine verdier ved å bruke aksessor metodene til Item klassen. Dette gjøres for alle variabler bortsett fra for variabelen for pris. Dette er fordi aksessor metoden for pris gjør en kalkulasjon med variabelen for rabatt, som hadde ført til dobbel rabatt til det dyp-kopierte varen.

### Metoder

Item klassen har aksessor metoder for alle felt variabler. Dette er fordi man skal kunne hente all informasjon om en vare for videre bruk. Alle aksessor metodene er enkle metoder som returnerer verdien til den spesifiserte variabelen, bortsett fra aksessor metoden for pris. Denne metoden tar til seg verdien for rabatt for å kalkulere den totale prisen til varen. Den enkle formelen er (prisen \* (1 – (rabatt / 100))). Resultatet vil bli den total prisen så lenge rabatt er et tall mellom 0 og 100, og pris er positiv. Siden rabatt er av datatypen double, og pris er av datatypen int, så blir resultatet rundet ned.

Klassen har definert noen mutator metoder. Alle felt variablene som ikke har nøkkelordet «final» har en mutator metode, fordi disse har blitt bestemt at de skal kunne endres. Det eksisterer mutator metoder for beskrivelse, pris, rabatt og antall varer på lager. Mutator metodene for pris, antall varer på lager og rabatt sjekker om det oppgitte antallet er negativ. Hvis den er negativ blir IllegalNumberException kastet. I tillegg så sjekker mutator metoden for rabatt om den oppgitte verdien er over 100. Mutator metoden for beskrivelse sjekker om den oppgitte strenger er verdien null. Hvis den er null blir ikke NullPointerException kastet. I stedet får beskrivelsen verdien til en tom streng.

En toString() metode er også definert for å kunne lett skrive ut informasjon om hele vare objektet. Denne overskriver den vanlige toString() metoden. Alle metodene har en oppgave og relevante navn for å oppnå høy cohesion.

### Refaktorering

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidenceVedlegget over viser første utkast av applikasjonen etter første iterasjon av oppgaven. Her ser vi at «Item» klassen har blitt definert med felt variablene som ble spesifisert av oppgaven, med relevante mutator og aksessor metoder. Det ble bestemt at variablene for forklaring (desc), antall varer på lager (warehouseStock) og pris (price) skulle ha mutator metoder. Dette er fordi det skal være mulig å endre på disse variablene siden de kan endres over tid, og var de mest åpenbare å implementere med engang. Andre felt variabler skal ikke kunne endres dersom de defineres ved konstruksjon av objektet, og det gir ikke logisk mening at de endres over tid. For eksempel, så endrer ikke høyden eller lengden på et objekt over tid, og derfor er de fast bestemt med engang. I tillegg til «Item» klassen så ble det fort tydelig at det kunne implementeres enum klasser for både kategori (Category) og color (Color).

Vedlegget ved siden av viser siste utkast av Item klassen. Forskjeller mellom første og siste versjon er blant annet ekstra konstruktøren for dyp-kopiering. Det har også blitt lagt til en ny felt variabel for rabatt.

## ItemBuilder

ItemBuilder klassen skal representere noe som «bygger» objekter av Item klassen. Denne klassen oppnår dette ved å bruke builder design mønsteret. Dette er fordi Item klassen har mange parametere, i tillegg til at ikke alle parametere trenger å bli definert, som er gode grunner for å bruke builder desing mønsteret. I tillegg er ItemBuilder klassen og Item klassen i samme modul, og de enesete i den modulen, som betyr at ItemBuilder klassen er den eneste klassen som har tilgang til Item klassen sine konstruktører. Dette betyr at man må bruke ItemBuilder klassen hvis en klasse fra en annen modul ønsker å lage nye Item objekter. Dette fører til loose-coupling mellom Item klassen og alle andre klasser, men tight-coupling mellom ItemBuilder og Item klassen.

### Felt variabler

Alle definerte felt variabler er private for at man ikke skal kunne endre på variablene etter konstruksjon av objektet, med mindre man bruker de eksisterende mutator metodene. Dette gjøres for å oppnå loose-coupling. De må også være identiske til felt variablene i Item klassen. Her har vi ingen variabler med nøkkelordet final fordi builder design mønsteret krever at alle variablene skal kunne endres mens Item objektet lages.

### Metoder

Det må finnes en mutator metode for alle felt variabler som skal føres inn i parameteret til Item konstruktøren ifølge builder design mønsteret. Hver mutator metode har også identiske tester for å kaste unntak for de forskjellige variablene som Item klassen også utfører. Når alle variabler har blitt spesifisert med mutator metodene kan build() metoden kalles for å konstruere Item objektet. Denne metoden kaller alle mutator metodene i klassen for å verifisere at alle variablene har de verdiene de skal ha. Et unntak blir kastet om en obligatorisk variabel ikke har fått en gyldig verdi. Denne klassen tilbyr også bruk av Item sin dyp-kopi konstruktør med den statiske metoden deepCopy(). Alle metodene har en oppgave og relevante navn for å oppnå høy cohesion.

## ItemRegister

ItemRegister klassen skal representere et virkelig-verden register som inneholder en eller flere objekter av Item klassen. Registeret tilbyr enkle metoder for modifisering av Item objektene registrert i registeret. Begrensninger ved registeret er blant annet at det ikke skal kunne finnes dupliserte vare nummere, som også ignorer små og store bokstaver. I tillegg skal det ikke være mulig å endre på noen av varene sine verdier uten å bruke registeret sine metoder.

### Felt variabler

Det eneste felt variabelen i ItemRegister klassen er variabelen itemList. Denne variabelen er av datatypen List, som er en klasse fra java.util modulen. List er en grensesnitt klasse som en rekke «liste» klasser implementerer, og brukes for å ha en samling av et spesifisert objekt. Variabelen har også nøkkelordet «final» som betyr at den kan kun initialiseres en gang, og er også private for å oppnå loose-coupling.

### Konstruktør

Konstruktøren i ItemRegister klassen er relativt enkel, den inneholder kun en operasjon. Her blir itemList variabelen initialisert og bruker klassen ArrayList, som er en klasse som implementerer List klassen. ArrayList er en liste med enkle metoder for å legge til og hente ut objekter i listen. Dette skal brukes videre for å kunne iterere gjennom listen, og utføre spesifikke operasjoner.

### Metoder

Alle metodene har en oppgave og relevante navn for å oppnå høy cohesion. I tillegg blir unntak kastet for å sikre at gyldig verdier er det som innføres i registeret for å unngå mulige problemer som kan oppstå av det. Metodene som er implementert er først og fremst oppgaven sine minimums krav for hva registeret skal kunne utføre. I tillegg har det blitt implementert ekstra metoder som gir logisk mening for et slikt register. For metoder som tar inn varer som parameter må det gjøres en iterasjon gjennom vare listen med stream. Denne iterasjonen sammenligner varen som er gitt sitt vare nummer med vare nummerere som er registrert for å finne varen i registeret. Dette må gjøres fordi alle metoder som returnerer en vare eller en liste med varer er dyp-kopiert for å oppnå loose-coupling, men betyr også at en enkel «.equals()» metode ikke kan brukes. Det er brukt funksjonell programmering i nesten alle metodene. toString() metoden som overskrives i denne klassen kan brukes til å få en streng som inneholder alle varene.

#### addItem()

Denne metoden tar inn et Item objekt som parameter, og oppgaven er å legge objektet til i registeret. Dette oppnår den ved å bruke «.add()» metoden av List klassen. Før dette kan skje må metoden sjekke at varen som skal legges til ikke har et vare nummer som allerede eksisterer i registeret. Hvis dette er tilfellet, blir IllegalArgumentException unntaket kastet. Dette unntaket brukes for når argumenter som tilføres ikke er gyldig.

NullPointerException blir også kastet dersom varen som spesifiseres er verdien null. Metoden for å sjekke at vare nummeret ikke eksisterer fra før av bruker stream for å iterere gjennom listen.

#### searchByItemNumber()

Denne metoden brukes for å søke etter en vare ved å ta inn et vare nummer som parameter. Metoden tar det spesifiserte vare nummeret og sammenligner det med alle vare nummeret som er i listen ved å bruke stream. Om en vare med lik vare nummer blir funnet, blir det returnert en dyp-kopi av varen. Dette fører til loose-coupling. Ellers blir det returnert verdien null. Metoden kaster også NullPointerException hvis den spesifiserte strengen er null.

#### searchByDesc()

Denne metoden brukes for å søke etter varer som har den spesifiserte beskrivelsen som blir gitt i parameteret. Metoden tar den spesifiserte beskrivelsen og sammenligner det med alle beskrivelsene som er i listen ved å bruke stream. Om en vare med lik beskrivelse blir funnet, blir en dyp-kopi av varen lagt til i en liste som skal returneres. Ved å bruke dyp-kopiering oppnår man loose-coupling. Ellers blir det returnert verdien null. Metoden kaster også NullPointerException hvis den spesifiserte strengen er null. Her blir en liste med varer returnert fordi flere varer kan ha samme beskrivelse.

#### searchByCategory()

Denne metoden brukes for å søke etter varer som har den spesifiserte kategorien som blir gitt i parameteret. Metoden tar den spesifiserte kategorien og sammenligner det med alle kategoriene som er i listen ved å bruke stream. Om en vare med lik kategori blir funnet, blir en dyp-kopi av varen lagt til i en liste som skal returneres. Ved å bruke dyp-kopiering oppnår man loose-coupling. Ellers blir det returnert verdien null. Metoden kaster også NullPointerException hvis den spesifiserte kategorien er null. Her blir en liste med varer returnert fordi flere varer kan ha samme kategori.

#### increaseItemStock()

Denne metoden sin ene oppgave er å øke antall varer som det finnes av en vare i registeret ved å ta inn varen og antallet som parameter. Varen som blir spesifisert sitt vare nummer brukes i for å finne varen i registeret gjennom en stream. Om den finner varen med identisk vare nummer, blir det brukt mutator metoden fra Item klassen for å endre på antall varer. Antallet spesifisert blir lagt til det eksisterende antallet. Metoden kaster unntak om varen spesifisert er null, eller antallet er negativt. Det blir også kastet et NoSuchElementException unntak hvis varen ikke blir funnet i registeret.

#### decreaseItemStock()

Denne metoden sin ene oppgave er å redusere antall varer som det finnes av en vare i registeret ved å ta inn varen og antallet som parameter. Varen som blir spesifisert sitt vare nummer brukes i for å finne varen i registeret gjennom en stream. Om den finner varen med identisk vare nummer, blir det brukt mutator metoden fra Item klassen for å endre på antall varer. Antallet spesifisert blir subtrahert fra det eksisterende antallet. Metoden kaster unntak om varen spesifisert er null, eller antallet er negativt. Det blir også kastet et NoSuchElementException unntak hvis varen ikke blir funnet i registeret. I tillegg kan mutator metoden kaste et unntak hvis antallet som er spesifisert er større enn eksisterende antall varer, siden dette leder til et negativt tall.

#### changePriceOfItem()

Den ene oppgaven til denne metoden er å endre på prisen til en vare, ved å ta varen som skal endres og prisen som parameter. Varen som blir spesifisert sitt vare nummer brukes i for å finne varen i registeret gjennom en stream. Om den finner varen med identisk vare nummer, blir det brukt mutator metoden fra Item klassen for å endre på prisen. Antallet spesifisert er prisen som varen skal ha. Metoden kaster unntak om varen spesifisert er null, eller antallet er negativt. Det blir også kastet et NoSuchElementException unntak hvis varen ikke blir funnet i registeret.

#### changeDiscountOfItem()

Den ene oppgaven til denne metoden er å endre på rabatten til en vare, ved å ta varen som skal endres og rabbaten som parameter. Varen som blir spesifisert sitt vare nummer brukes i for å finne varen i registeret gjennom en stream. Om den finner varen med identisk vare nummer, blir det brukt mutator metoden fra Item klassen for å endre på rabatten. Antallet spesifisert er rabatten som varen skal ha. Metoden kaster unntak om varen spesifisert er null, eller antallet er negativt. Det blir også kastet et NoSuchElementException unntak hvis varen ikke blir funnet i registeret.

#### changeDescriptionOfItem()

Den ene oppgaven til denne metoden er å endre på beskrivelsen til en vare, ved å ta varen som skal endres og beskrivelsen som parameter. Varen som blir spesifisert sitt vare nummer brukes i for å finne varen i registeret gjennom en stream. Om den finner varen med identisk vare nummer, blir det brukt mutator metoden fra Item klassen for å endre på beskrivelsen. Beskrivelsen spesifisert er beskrivelsen som varen skal ha. Metoden kaster unntak om varen spesifisert er null. Det blir også kastet et NoSuchElementException unntak hvis varen ikke blir funnet i registeret.

#### getIndexOfItem()

Den ene oppgaven til denne metoden er å hente indeks nummeret til den spesifiserte varen i parameteret i registeret. Her brukes en IntStream for å telle antall varer som har blitt iterert over for å få indeksen til varen. Indeksen blir returnert dersom varen blir funnet, ellers returneres verdien -1. Dette ligner veldig på indexOf() metoden i List klassen. Om den spesifiserte varen er null blir NullPointerException kastet.

#### removeItem()

Det skal også være mulig å fjerne en vare som er lagt til i registeret. Denne metoden tar varen som skal fjernes som parameter. En stream brukes for å finne varen i registeret ved å bruke vare nummeret. Dersom en vare blir funnet brukes «.remove()» metoden fra List klassen for å fjerne varen og returnerer sant, ellers hvis varen ikke blir funnet returnerer metoden falsk. Dette kan brukes for å verifisere at fjerningen av varen var vellykket eller ikke.

#### sortListBy Metodene()

En rekke sorterings metoder kan brukes for å sortere varene i registeret avhengig av hvilken sorterings metode man bruker. Man har muligheten til å sortere med vare nummeret, fargen, kategori og merkenavnet, som er alfabetiske sorteringer. I tillegg er det mulig å sortere med pris og antall varer, som er sortert etter størrelse på antallet. Alle sorterings metodene tar en boolsk verdi som parameter som bestemmer om det skal sorteres fra lavest til størst, eller motsatt. De bruker .sort() metoden fra List klassen for å sortere.

#### getCopyOfList()

Denne metoden returnerer en liste av dyp-kopierte varer av varene som er i registeret. Dette fører til loose-coupling. Her brukes også funksjonell programmering ved å bruke streams.

#### getItem()

Denne metoden returnerer en dyp-kopi av varen som tilsvarer indeksen som er oppgitt i parameteret. Indeksen brukes for å hente ut varen ved å bruke «.get()» metoden i List klassen, og lage en dyp-kopi av varen som skal returneres. Dette fører til loose-coupling. I tillegg kan metoden kaste en IndexOfOutBoundsException om indeksen spesifisert er utenfor grensen for liste størrelsen.

### Refaktorisering

[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).]

# 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]

# 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.]

# References

*Abstraksjon (datavitenskap)*. (2022, November 22). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Abstraction\_(computer\_science)

Gamma, E., Helm, R., Ralph, J., & John, V. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley.

Gosling, J., Bill, J., Guy, S., & Gilad, B. (2005). *The Java Language Specification, Thrid Edition.* Boston: Addison-Wesley.

Yourdon, E., & Constantine, L. L. (1975). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* New York: YOURDON inc.

[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.]