| Kandidatnummer(e)/Navn: | | | |
| --- | --- | --- | --- |
| 10124 | | | |
| Dato: | Fagkode: | Studium: | Ant sider/bilag: |
| 12.12.22 | IDATT1001 | Programmering 1 | 23 / |

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

|  |
| --- |
| Tittel : |
| En rapport om en Applikasjon for et Varehus Styringssystem |

|  |
| --- |
| Sammendrag: |
| Denne rapporten er besvarelsen til den obligatorisk innlevering i faget IDATT1001 Programmering 1. Den forteller om applikasjonen som skal løse problemstillingen, spesifisert i oppgave teksten.  Det skulle lages en applikasjon som skulle utvikles for et smart varehus, hvor det skulle være mulig å registrere nye varer, søke etter varer eller fjerne varer.  Resultatet ble en applikasjon med et tekstbasert brukergrensesnitt som har muligheten til å navigere seg gjennom registeret ved hjelp av forhåndsdefinerte funksjoner for å kunne utføre minimums krave til oppgaven.  Programmet er kodet fullstendig i Java ved bruk av IDE’en InteliJ. I tillegg har det blitt brukt Github for versjonskontroll og historikk av koden. |

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

INNHOLD

[1 SAMMENDRAG 1](#_Toc121818657)

[2 TERMINOLOGI 1](#_Toc121818658)

[3 INNLEDNING – PROBLEMSTILLING 1](#_Toc121818659)

[3.1 Bakgrunn/Formål og problemstilling 1](#_Toc121818660)

[3.1.1 Del 1 1](#_Toc121818661)

[3.1.2 Del 2 1](#_Toc121818662)

[3.1.3 Del 3 2](#_Toc121818663)

[3.2 Avgrensninger 2](#_Toc121818664)

[3.3 Begreper/Ordliste 2](#_Toc121818665)

[3.4 Rapportens oppbygning 4](#_Toc121818666)

[4 BAKGRUNN - TEORETISK GRUNNLAG 4](#_Toc121818667)

[4.1 Coupling 4](#_Toc121818668)

[4.2 Cohesion 4](#_Toc121818669)

[4.3 Design Pattern 5](#_Toc121818670)

[4.3.1 Builder Pattern 5](#_Toc121818671)

[4.4 Modularisering 5](#_Toc121818672)

[4.5 Abstrahering 6](#_Toc121818673)

[4.6 Funksjonell Programmering 6](#_Toc121818674)

[4.7 Datatyper og Minne 6](#_Toc121818675)

[4.8 Tidskompleksitet 7](#_Toc121818676)

[5 METODE – DESIGN 7](#_Toc121818677)

[6 RESULTATER 7](#_Toc121818678)

[6.1 Item 8](#_Toc121818679)

[6.1.1 Felt variabler 8](#_Toc121818680)

[6.1.2 Konstruktør 9](#_Toc121818681)

[6.1.3 Metoder 10](#_Toc121818682)

[6.1.4 Refaktorering 10](#_Toc121818683)

[6.2 ItemBuilder 11](#_Toc121818684)

[6.2.1 Felt variabler 11](#_Toc121818685)

[6.2.2 Metoder 11](#_Toc121818686)

[6.3 ItemRegister 12](#_Toc121818687)

[6.3.1 Felt variabler 12](#_Toc121818688)

[6.3.2 Konstruktør 12](#_Toc121818689)

[6.3.3 Metoder 12](#_Toc121818690)

[6.3.4 Refaktorisering 15](#_Toc121818691)

[6.3.5 Color 15](#_Toc121818692)

[6.3.6 Category 15](#_Toc121818693)

[6.3.7 IllegalNumberException 16](#_Toc121818694)

[6.3.8 Brukergrensesnittet 16](#_Toc121818695)

[6.3.9 Ferdig Struktur 17](#_Toc121818696)

[6.3.10 Brukerveiledning 17](#_Toc121818697)

[7 DRØFTING 18](#_Toc121818698)

[7.1 Item 18](#_Toc121818699)

[7.2 ItemBuilder 18](#_Toc121818700)

[7.3 ItemRegister 18](#_Toc121818701)

[8 KONKLUSJON – ERFARING 19](#_Toc121818702)

[9 References 19](#_Toc121818703)

# SAMMENDRAG

Denne rapporten er besvarelsen til den obligatorisk innlevering i faget IDATT1001 Programmering 1. Den forteller om applikasjonen som skal løse problemstillingen, spesifisert i oppgave teksten.

Det skulle lages en applikasjon som skulle utvikles for et smart varehus, hvor det skulle være mulig å registrere nye varer, søke etter varer eller fjerne varer.

Resultatet ble en applikasjon med et tekstbasert brukergrensesnitt som har muligheten til å navigere seg gjennom registeret ved hjelp av forhåndsdefinerte funksjoner for å kunne utføre minimums krave til oppgaven.

Programmet er kodet fullstendig i Java ved bruk av IDE’en InteliJ. I tillegg har det blitt brukt Github for versjonskontroll og historikk av koden.

# TERMINOLOGI

UML Unified Modeling Language

UP Unified Process

WMS Warehouse Management System

IDE Integrated Development Environment

GUI Graphical User Interface

# 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.

## 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.

## Begreper/Ordliste

| 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 |
| Try-catch (Prøv-fang) | Try-catch | En kodeblokk for å forsøke å fange unntak som kan kastes |
| Allokering | Allocating | Dedikere en mengde av den totale mengden til noe som trenger det |

## Rapportens oppbygning

Rapporten er bygd opp med hoved overskrifter, som identifiseres med store bokstaver og et eneste tall som viser til kapittel indeksen. Disse markerer en stor endring i tema i rappporten.

Videre har hoved overskriftene underoverskrifter, som består av tykke bokstaver og et tall med kapittel indeksen og et punktum med del kapittel indeks bak. Gjentatte punktumer indikerer del kapiteler av del kapiteler.

Rapporten avslutter med en konklusjon og en referanseliste.

# 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)

## Datatyper og Minne

De fleste programmeringsspråk deler de samme navnene for primitive datatyper. Dette inkluderer datatyper som int, float, byte, char osv. Selv om hver datatype har sine egne definerte navn utefra hva man ønsker å gjøre med variabelen så har betyr de forskjellige datatypene bare forskjellige størrelser på antall bytes som blir brukt i minne ved allokering. Maskiner forstår bare 1’ere og 0’ere, så man har bestemt at en bit, som enten er 1 eller 0, er en byte når 8 bits er gruppert sammen. Alle andre datatyper er bare forskjellige antall bytes. I Java så er alle primitive datatyper signert. Dette vil si at den helt fremste bit’en bestemmer om taller er negativt eller ikke. For eksempel så er int 4 bytes i størrelse. 4 bytes \* 8 = 32 bits, men siden den fremste bit’en er dedikert for å si noe om tallet er negativt eller ikke, så har man effektivt 31 bits for å representere tallet. Under ser man de forskjellige byte størrelsene for de forskjellige datatypene i Java.

|  |  |
| --- | --- |
| Datatype | Antall Bytes |
| char | 1 |
| short | 2 |
| int | 4 |
| long | 4 |
| Long long | 8 |
| float | 4 |
| double | 8 |
| Long double | 16 |

(Oracle, 2010)

## Tidskompleksitet

Innenfor informatikk er tidskompleksiteten det som beskriver hvor mye datatid det tar å kjøre en algoritme. Dette er ofte en estimering som viser hvor mye datatiden øker som en funksjon av størrelsen på datasettet. Dette estimeres ved å anta at hver elementær operasjon tar en fast mengde tid å utføre, og deretter summere antall elementære operasjoner utført av algoritmen.

Stor O-notasjon er ofte benyttet for å fortelle noe om tidskompleksiteten til en algoritme. For eksempel, så vil O(n) bety en lineær tidsalgoritme, hvor datatiden er proporsjonal med n, hvor n er størrelsen på datasettet. O(1) vil si at en elementær operasjon har konstant tid, og regnes som den raskeste tidskompleksiteten, siden den er uavhengig av datastørrelsen.

Ved skalering av et system vil man ofte tenke på tidskompleksiteten og hvordan det kan påvirke behandlingstid. Dette bestemmer også om systemet burde skaleres vertikalt, eller horisontalt. Man vil ofte unngå store tidskompleksiterer som ofte kan skje ved å ha for-løkker inni for-løkker. (Sipser, 2006)

# 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.

# 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. Builder design mønsteret promoterer en kode som er mer lesbar ved at man kan tydelig se variablene som blir gitt en verdi.

### 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.

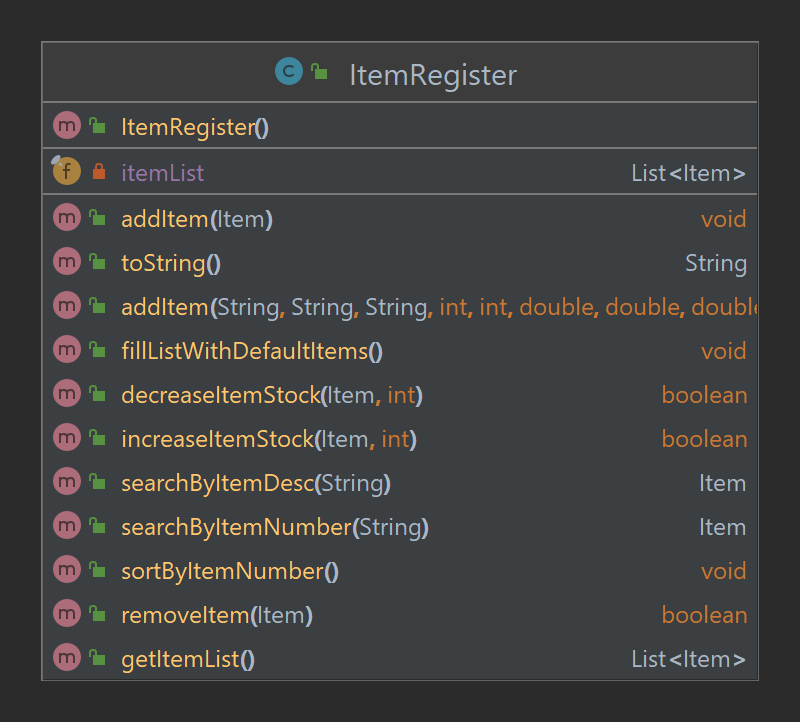
#### getCopyOfList()

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





Vedlegget over viser til det første utkaste av ItemRegister klassen ved andre iterasjon av oppgaven. De implementerte metodene er fulgt av kravspesifikasjonene som oppgaven hadde spesifisert. I denne versjonen av ItemRegister var det ingen bruk av funksjonell programmering og kun ytterlig iterasjon ble brukt, for eksempel en vanlig for-løkke. I tillegg var det ingen dyp-kopiering, som betyr at varene sin sikkerhet var lav fordi de kunne blitt endret utenfor registeret sin bevissthet, som fører til tight-coupling.

Vedlegget ved siden av viser til den siste versjonen av ItemRegister klassen. Her har det blitt implementert flere funksjonaliteter som er utenfor kravspesifikasjonene til oppgaven. I tillegg har det blitt brukt funksjonell programmering og dyp-kopiering i motsetning til det første utkastet. En annen forskjell er endringen i parametere for de forksjellige metodene, spesielt addItem() metoden. Her ser man i første utkast alle parameterne som må innføres for å bruke metoden. I siste utkast er det kun en parameter, som fører til loose-coupling.

### Color

Color er en enum klasse som representerer farger. Dette brukes av Item klassen for å spesifisere fargen til en vare. Dette gjør det lett å legge til nye farger, i tillegg til å verifisere at fargen som er oppgitt finnes.

### Category

Category er en enum klasse som representerer kategorien som varen tilhører. Implementasjonen av en enum klasse gjør det veldig enkelt for en programmerer å legge til nye kategorier. Ingen andre klasser blir påvirket av at man legget til flere kategorier i Category klassen, som betyr at det er høy cohesion og loose-coupling. I tillegg til de fire kategoriene som er spesifisert i kravet til oppgaven, gulv, laminater, vinduer, dører og trelast, så har det blitt lagt til en kategori for metall.

### IllegalNumberException

IllegalNumberException er en unntaks klasse for unntak som gjelder ugyldige tall. Den arver fra IllegalArgumentException unntaks klassen, og brukes når et ugyldig tall blir brukt i en metode, for eksempel at en variabel ikke kan være negativ.

### Brukergrensesnittet

Brukergrensesnittet tilbyr enkel funksjonalitet og lett forståelse for brukeren for å kunne bruke registeret på en effektiv måte. Hele menyen er laget i Menu klassen for høy cohesion. I tillegg er de forskjellige meny mulighetene definert i en pakke beskyttet enum klasse MenuOption for å gjøre det lett å legge til flere funksjoner.

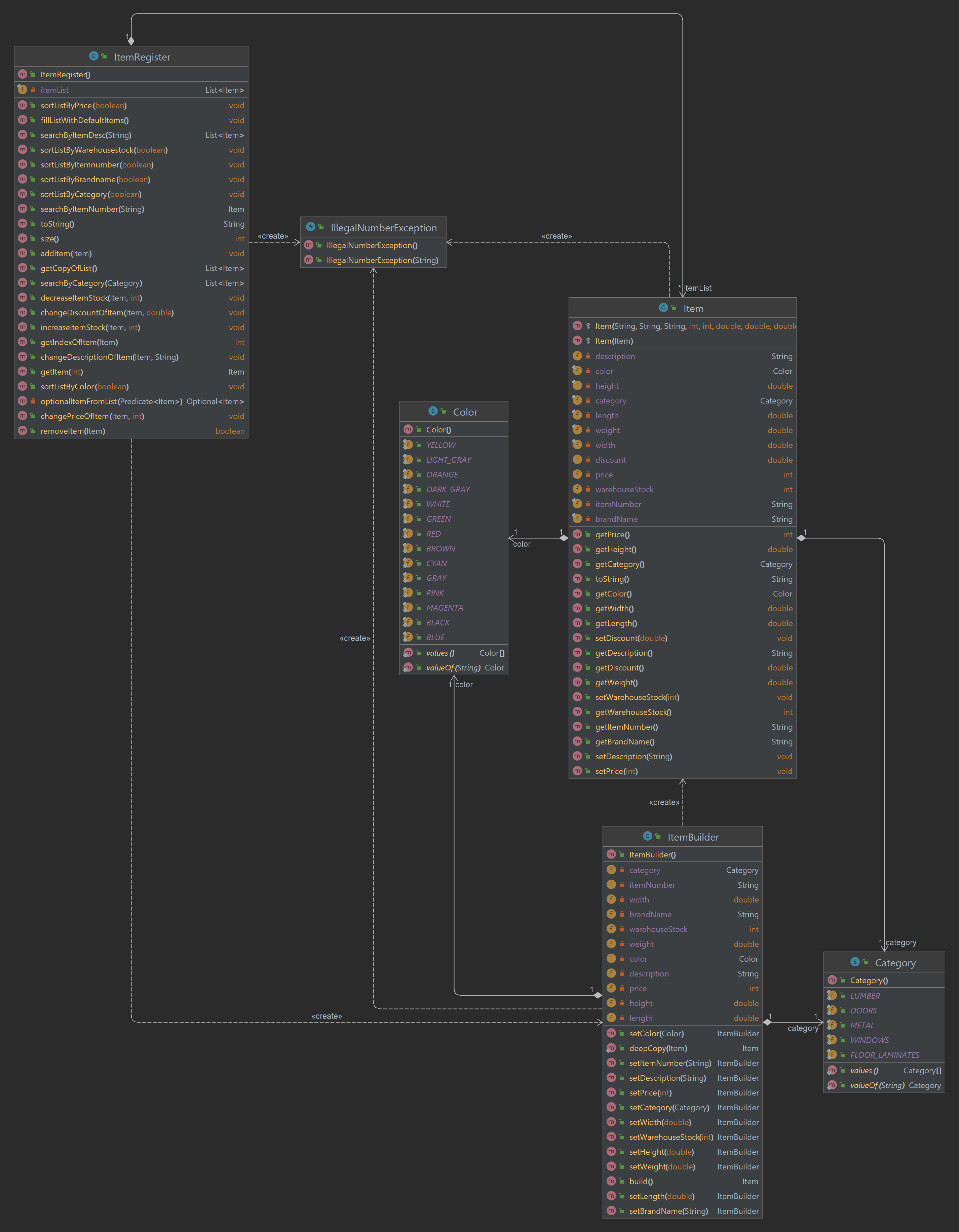
Det er også blitt brukt funksjonell programmering, ved at en egen definert funksjonell grensesnitt klasse UserInput brukes for å få inn input fra brukeren for å bearbeide det. Dette gjør det mulig å bruke lambda uttrykk.

Metodene som brukes fra ItemRegister klassen er ofte brukt i en try-catch block for å kunne unntaks håndtere mulige feil som kan oppstå. Dette fører til et feil sikkert program så lenge metodene kaster de forventede unntakene. Det gir også en bedre bruker opplevelse ved at programmet ikke krasjer om en feil skulle skje.

I tillegg er det brukt farger på teksten for å gi en klar skille mellom feil meldinger og positive meldinger, i dette tilfellet blir rød tekst brukt til feil meldinger og grønn tekst brukt til positive meldinger.

En bruker-vennlig funksjon som er implementert er mulig heten for å kunne returnere til hoved menyen når som helst i programmet. Dette kan gjøres ved å bruke den definerte kommandoen «/back».

### Ferdig Struktur

Vedlegget ved siden av viser klasse diagrammet for det siste utkastet av applikasjonen. En merkverdig forskjell mellom det ferdige utkastet og en tidligere, er bruken av ItemBuilder klassen i ItemRegister istedenfor at ItemRegister klassen bruker Item konstruktøren.

### Brukerveiledning

* Last ned en Java JDK versjon, minst versjon 16.0 ([Java nedlastning](https://www.oracle.com/java/technologies/downloads/))
* Last ned en IDE som har muligheten til å kjøre Java programmer (InteliJ, BlueJ)

[InteliJ](https://www.jetbrains.com/idea/)

[BlueJ](https://www.bluej.org/)

* Kjøre programmet i IDE:
* Åpne prosjektet i IDE’en
* Kompiler koden
* Kjør main metoden
* Kjøre programmet i konsoll:
* Naviger til mappen i konsollen
* Kompiler koden med kommandoen -javac Main.java
* Kjør programmet med kommandoen -java Main.java

# DRØFTING

## Item

Løsningen som er kommet frem i Item klassen kan man mene er en god løsning på oppgaven. Dette er fordi Item klassen er sterk, ved at metodene og konstruktøren kaster unntak ved feil verdier, og at det er høy cohesion fordi alle metodene har en oppgave og variablene har logiske data typer som passer. Til tross for dette så kan man mene at noen datatyper kunne vært mer optimalisert. For eksempel så kan man si at variabler for vekt, lengde, høyde og lignende kunne brukt datatypen float. Dette er fordi float kun har 4 bytes i størrelse i motsetning til datatypen double som har 8, som betyr at man kunne ha spart mer plass i minne. Likevel så fungerer double like bra, selv om de ekstra desimalene er effektivt ubrukelig i denne sammenhengen.

Man kan også mene at det skulle vært mutator metoder for variabler som farge eller kategori. Dette er fordi i den virkelige-verden så kan varer sine farger endre seg plutselig hvis produsenten ønsker det. I tillegg kan varer havne i andre kategorier enn det som var definert fra starten av. Dette kunne løses ved å ikke gjøre variablene final, og lagd mutator metoder for disse.

I denne løsningen så har variabelen for pris fått lov til å være 0. Dette indikerer at varen er gratis. Man kan argumentere for at det ikke burde være lov å ha pris lik 0 fordi det ikke skal være mulig å ha gratis varer i registeret. Til tross for dette så har det blitt bestemt at det skal kunne være lov, fordi om et varehus ønsker å gi bort mange varer, så skal disse kunne bli oppgitt som gratis.

Et avvik fra oppgaven sine spesifikasjoner her er implementasjonen av variablene for bredde og rabatt. Disse ble implementert på logisk grunnlag, fordi i en tre dimensjonal verden må man ha alle tre dimensjoner oppgitt, og det skal kunne være mulig å sette en rabatt på varer som trenger det.

## ItemBuilder

Man kan mene at bruken av ItemBuilder klassen promoterer oversiktlig kode, og skiller konseptet av et vare objekt fra skapelsen av varen. Dette er fordi ved å følge builder design mønsteret så er mutator metodene klar og tydelig på hvilke variabler som blir satt, som gjør det enkelt å lese koden. Uansett så kunne ItemBuilder klassen implementeres mer sterkt hvis flere parametere i konstruktøren til Item klassen var valgfrie. Dette er fordi builder design mønsteret funker bra når man kan konstruere et objekt på flere måter, uten å kunne trenge alle parametere hver gang. Man kunne ha løst dette ved å ikke kaste NullPointerExcetions i mutator metodene i ItemBuilder klassen, og heller hatt en standard verdi for «ikke oppgitt». Likvell så er ItemBuilder klassen robust i å sikre at vare objektene som skapes ikke har noen feil med verdiene.

## ItemRegister

Man kan mene at register klassen løser oppgaven sin problemstilling. Dette er fordi ItemRegister klassen tilbyr alle metodene som er spesifisert, i tillegg til enda mer funksjonalitet utenfor kravene.

Metodene har også høy cohesion. Grunnen til dette er fordi metodene har en oppgave som den utfører og metodene er bedre egnet til å være i denne klassen framfor å ha dem i for eksempel Item klassen. I tillegg har det blitt tenkt på loose-coupling. Dette er fordi det ikke er mulig å endre på varene i registeret uten at registeret vet at de har blitt endret på. Dette er oppnådd ved å bruke dyp-kopiering.

Gjennom hele klassen blir funksjonell programmering brukt. Dette bidrar til lettere å lese kode, i tillegg gjør stream at itereriringen skjer inni listen, i motsetning til å skje utenfor.

Klassen List brukes for å holde på varene i en liste. Man kan argumentere for at denne ikke er det mest optimale valget når det gjelder hvilke «liste» klasser som eksisterer i Java. Dette er fordi tidskompleksiteten er O(n) for å finne frem varer fordi metodene må sammenligne vare numrene og iterere gjennom listen for hver gang. Dette betyr at for veldig store datasett, vil dette være en ulempe for programmet. Bedre alternativer hadde vært for eksempel bruken av HashMap klassen. Denne klassen har en nøkkel og verdi hvor nøkkelen er det som hasher til objektet i minnet. Denne klassen hadde derfor vært et logisk valg med tanke på at registeret skal kun ha unike vare numre. I tillegg så er tidskompleksiteten for å slå opp en vare i en HashMap O(1), som er veldig praktisk når vare antallet stiger.

# KONKLUSJON – ERFARING

Applikasjonen er en god løsning på oppgaven. Programmet utfører det oppgaven spør etter, og tilbyr en bruker-vennlig brukergrensesnitt som gjør det lett å navigere registeret. Hvis man skulle begynt på nytt, hadde de små problemene som nevnt i drøfting blitt fikset, ved å for eksempel benytte HashMap klassen.

I tillegg så hadde det vært gunstig å implementere fil håndtering, for å kunne lagre register informasjonen. Dette hadde vært et krav for en virkelig-verden register siden all informasjon blir mistet når programmet skrus av, som er en stor sikkerhetsfeil. Om programmet lukkes eller maskinen skrus av blir alle nye registrerte varer mistet. Dette kan lett løses med et fil håndteringssystem, hvor registeret skriver hver vare inn i en fil, for å kunne lese fra filen igjen når programmet starter opp. Dette hadde skapt nye problemer, for eksempel at man kan endre varene i fila, så da kunne mann kanskje benyttet noe form for kryptering.

Andre begrensninger av løsningen er det tekstbaserte brukergrensesnittet. Det hadde vært blant annet mer bruker-vennlig å implementere en GUI for å bruke registeret. I tillegg må brukeren ha Java installert for å kjøre programmet, noe som mange, men ikke alle har. Til tross for dette tar programmet hensyn til farge blinde brukere, ved at bakgrunnsfargen ikke er en komplimenter farge med tekst fargen.

Om applikasjonen skulle videre utvikles kunne det kanskje vært smart å benytte abstrahering, og lagd klasser av Item klassen. Dette kunne for eksempel vært en matvare klasse, som hadde arvet av Item klassen. I tillegg kunne det implementeres flere grensesnitt klasser for bedre oversikt og cohesion. Å pakke programmet i en jar fil med alle de nødvendige avhengighetene hjelper også med å distribuere programmet på en enkel måte til mange potensielle brukere.

Totalt sett er applikasjonen et fungerende program for hva den prøver å være, og kunne blitt brukt av et enkelt smart varehus.

# References

1. *Abstraksjon (datavitenskap)*. (2022, November 22). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Abstraction\_(computer\_science)
2. Gamma, E., Helm, R., Ralph, J., & John, V. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley.
3. Gosling, J., Bill, J., Guy, S., & Gilad, B. (2005). *The Java Language Specification, Thrid Edition.* Boston: Addison-Wesley.
4. Oracle. (2010). *Data Types and Sizes*. Hentet fra Oracle: https://docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-2/index.html
5. Saumont, P.-Y. (2017). *Functional Programming in Java: How functional techniques improve your Java programs.* Shelter Island: Manning Publications.
6. Sipser, M. (2006). *Introduciton to the Theory of Computation: Third edition.* Boston: Thomson Course Technology.
7. Yourdon, E., & Constantine, L. L. (1975). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* New York: YOURDON inc.