STUDENTRAPPORT

NTNU

KANDIDATNUMMER(E)/NAVN:			
10122			
DATO:	FAGKODE:	STUDIUM:	ANT SIDER/BILAG:
11.12.22	IDAT1001	Dataingeniør, B.sc. BIDATA	14 /

FAGLÆRER(E) :
Muhammad Ali Norozi

TITTEL :
Mappevurdering – Smarthus A/S Warehouse Management System (WMS)

SAMMENDRAG:
Det skal utvikles en programvare som skal brukes av et varehus, hos Smarthus AS.
Programvaren skal brukes til å håndtere varelageret til Smarthus AS. Slike systemer kalles ofte "Warehouse Management System», eller bare «WMS» på engelsk.
Smarthus AS leverer hovedsakelig varer til bygg-industrien, så typiske varer er laminatgulv, dører og vinduer, lister og annet trevirke.
Løsningen skal til slutt bestå av et tekstbasert brukergrensesnitt og et register som lagrer informasjon.

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

INNHOLD

1 SAMMENDRAG

Denne rapporten inneholde besvarelse på mappevurdering i emnet IDATT1001 (Programmering 1) ved studiet for Dataingeniør ved NTNU.

Rapportens hensik er å beskrive prosjektets problemstilling, planleggingen for å løse oppgaven og sluttresultatet.

Prosjektet gikk ut på å skrive et program for å håndtere lagerføring av varer. Dette skulle gjøres i henhold til krav satt frem i oppgaveteksten, hvor det ble definert hvordan en vare skulle representeres, og hvordan det hele skulle utformes. Prosjektet var delt i tre deloppgaver, som hver bygget videre på den foregående.

Det ble først skissert, i grove trekk, hvordan et slikt system kunne utformes. Deretter ble det systematisk implementert funksjonalitet i henhold til kravene beskrevet i oppgaven. For å ha et godt utgangspunkt, ble god kodestil brukt fra start, og mye tid brukt på å forsikre seg om at koden ble skrevet i henhold til gode kodeprinsipper.

Gjennomføringen av del 2 gikk mye ut på å legge grunnstrukturen for applikasjonen, mens det i del 3 skulle settes sammen til en velfungerende løsning. I del 3 ble det en del opprydding og omskriving for å øke lesbarhet og sørge for at de logiske operasjonene fungerte om ønsket.

Resultatet ble et program som fungerer i stor grad slik jeg så for meg, selv om det har vært en del arbeid for å komme dit. Mye nytt måtte testes, prøves og læres for å knytte sammen alle klassene og funksjonalitetene. God kodestil har vært avgjørende for å lese koden; både for meg selv og andre. Om noe skulle blitt gjort annerledes, kunne det nok blitt lagt mer arbeid i å se på alternativene til ArrayList for å holde på vareregisteret. Men i denne omgang vurderte jeg ArrayList for å være et godt valg, siden det fungerer greit til formålet, og at det var en type jeg allerede hadde kjennskap til.

2 TERMINOLOGI

UML	Unified Modeling Language
UP	Unified Process
WMS	Warehouse Management System
AL	ArrayList
IDE	Integrated Development Environment
GUI	Graphical User Interface
TUI	Terminal User Interface
UI	User Interface (generellt)
UX	User Experience

3 INNLEDNING – PROBLEMSTILLING

3.1 Bakgrunn/Formål og problemstilling

Denne rapporten omhandler prosjektoppgaven for mappevurdering i første semester på bachelor i Dataingeniør ved NTNU. Mappevurderingen tar plassen til konvensjonell eksamen, og har som formål å gi emneansvarlig et grunnlag for å vudere studentenes kunnskap, samt å gi studentene mulighet til å vise hvilke evner og egenskaper innen programmering de har tilegnet seg i løpet av studiets første semester.

Oppdragsgiver er således Norges teknisk- naturvitenskapelige universitet (NTNU), ved Institutt for datateknologi og informatikk (IDI), ved emneansvarlig Mohammed Ali Norzi.

Oppgavens problemstilling går ut på å designe et program for administrasjon av et varehus for byggevarer («Warehouse Management System», WMS).

Kravspesifikasjonen kan oppsummeres ved at det skal lages det overnevnte WMS-programmet, som skal brukes av lageransatte for å holde kontroll på lagerbeholdningen. Den skal inneholde metoder for å manipulere lagerbeholdning. Det vil si at det skal lages et brukergrensesnitt, hvor ansatte skal kunne legge til og fjerne varer; oppdatere lagerbeholdningen av en bestemt vare, endre pris og varebeskrivelse og sette rabatter på bestemte varer. I tillegg må lagerbeholdningen kunne søkes i, enten ved bruk av serienummer eller enkeltord som matcher hele eller deler av varebeskrivelsen. Videre må man selvfølgelig også ha mulighet til å vise hele lagerbeholdningen, altså «skrive den ut» på skjermen.

Programmet sjekkes i henhold til enten BlueJ eller Google sin kodestil, ved hjelp av CheckStyle-plugin. Videre skal også koden være dokumentert i henhold til JavaDoc-standarden. Dette for å sørge for at koden i stor grad følger de konvensjonene som er etablert for Java-programmering, slik at det blir lettere for andre å lese og skjønne koden. Det skal altså tas i bruk konvensjoner for sikker, god og robust kode, som studentene skal ha tilegnet seg i løpet av semesteret.

Mer utfyllende informasjon kan leses i oppgavetekstens kravspesifikasjoner.

UML-diagrammer:

3.2 Avgrensninger

Oppgavens avgrensninger går i hovedsak ut på hvilke attributter som skal eksistere på hver vare (*Item*), og hvilke metoder som er ønsket implementert. Hver deloppgave setter således begrensninger på hva som skal implementeres for hvert trinn.

Eksempelvis får vi, i del 1, gitt en liste med atributter som skal registreres på en vare (*Item*). Disse setter en begrensning på hvordan man skal implementere *Item*-klassen. For eksempel vil de som har enten jobbet på et trevarelager, i byggindustrien, har bygget ting hjemme, eller bare har litt praktisk innsikt, fort legge merke til at det mangler et vesentlig parameter i å beskrive dimensjonene på varene, nemlig «bredde» (eng., «*width*»). Uten denne er det vanskelig å planlegge bruken av byggevarer, siden man som regel bygger i tre dimensjoner.

I tillegg har vi «pris». Den er i kravspesifikasjon representert bare med én variabel, men når vi i deloppgave 2 får beskjed om å legge til metode for å sette rabatt på en vare, kan det å ha bare

den ene prisattributten være en begrensning, i at man enten må overskrive den originale prisen med rabattert pris, eller lage en ny liste hvor man lagrere rabatterte varer med ny pris. Heldigvis får man, i deloppgave 3, i oppgave å «*sette ditt eget preg på løsningen*». Dermed åpnes det opp for å tenke litt selv, og dermed kan man implementere slik «manglende» funksjonalitet, så lenge en fortsatt opererer innefor det som er fornuftig gitt oppgaven. Endringer og forbedringer kan da implementeres, som f.eks å representere tømmerlast i tre dimensjoner, eller å legge til en egen prisattributt for rabattert pris.

Utover disse er det ganske grunnleggende funksjoner som etterspørres i kravspesifikasjonen, og den er formulert slik at man står relativt fritt til å implementere dem slik man selv anser som hensiktsmessig.

3.3 Begreper/Ordliste

Begrep (Norsk)	**Begrep (Engelsk)**	**Betyding/beskrivelse**
Produkt	Product	Varehuset håndterer produkter inn og ut av lager.
Lager	Storage/ warehouse	
Vare	Item	En vare, et produkt.
Lagerbeholdning	Stock	Varer på lager.
Lagerbeholdning	Inventory	Varer på lager.
Antall varer på lager	Items on shelf	Antall varer som «ligger på hylla», dvs tilgjengelig på lager
Varelager	Inventory of items	Et varelager; varebeholdningen.
Produktregister	Product Register	Et register som holder på produktene/varene som håndteres av lageret og systemet.
	setter	Metode for å sette/endre/oppdatere verdier i et objekt.
	getter	Metode for å hente ut/lese verdier i et objekt.
	ArrayList	En liste inni en tabell.
Dyp kopi	Deep copy	En separat kopi av et datasett/objekt. Man har da to like, men separate, datasett.

Begrep (Norsk)	Begrep (Engelsk)	Betyding/beskrivelse
Grunn kopi	Shallow copy	En kopi av et objekt som peker tilbake på det opprinnelige objektet/datasettet. Man har i praksis bare ett datasett, men to forskjellige objekter som refererer til/peker på dette.
Samhold	Choesion	Sa
Kobling	Coupling	
Aggregering	Aggregation	Objekter eksisterer uavhengig av hverandre. Konstruerer andre objekter ved å referere til informasjon fra andre. Man aggregerer informasjon fra en eller flere separate kilder.
Komposisjon	Compsition	Et «hovedobjekt» består av andre delobjekter, hvis eksistens avhenger av hovedobjektet. Man bygger objekter ved å komponere dem av flere delobjekter.
Modularisering		
Metode	Method	Et «avsnitt» kode som utfører en handling. Også kjent som «funksjon» (function).

3.4 Rapportens oppbygning

Denne rapporten er bygget opp ved å først drøfte teorien bak programmets konstruksjon. Deretter går den inn på metode og design, hvor det gjøres rede for planlagt gjennomføring, og hvordan planen endret seg gjennom prosjektet.

Deretter vises det til prosjektets resultater, med påfølgende drøfting av disse, og en konklusjon hvor erfaringer og læring blir diskutert.

Til slutt presenteres referanser og vedlegg.

4 BAKGRUNN - TEORETISK GRUNNLAG

Når en skal designe en løsning for bruk i varehus er det viktig at den gjør jobben til lager- og logistikkarbeiderene så enkel som mulig. Ganske enkelt må det kunne sies at å jobbe med programvaren ikke skal føles som om programvaren er i veien for å gjøre jobben effektivt. Dessverre er nettopp det at programvare oppleves som et hinder tilfellet med veldig mange systemer som er i bruk i dag, og en fellesnevner her er at disse programmene ofte er basert på gamle systemer som aldri var tiltenkt dette bruksområdet, men som har blitt bygget på og utvidet etterhvert som behover ble avdekket. Dette fører ofte til veldig begrensede løsninger, som sjeldent fungerer mer enn «godt nok». Et godt eksempel er Helseplattformen ved St. Olavs hospital, som tar i bruk det som må kunne sies å være et utdatert og dårlig system, som aldri var bygget med den hensikt å fungere som en løsning for å behandle pasientdata, og i alle fall ikke designet for norske forhold. Et annet eksempel som ligger nærmere oppgavens kjerne, er JD Edwards («JDE») WMS or ERP. Dette er populære systemer brukt i industri og

varehushåndtering, til tross for at de er kjent for å være tungvindte og lite optimaliserte løsninger for det meste.

Det er derfor viktig å identifisere sluttbrukerens behov, og bygge programvaren opp slik at disse blir ivaretatt. Samtidig må man strukturere koden slik at det ikke er rom for feil, som f.eks å omgå de innebygde funksjonene og utføre handlinger som ikke var tiltenkt mulig.

Dette kan oppnås ved å identifisere hvilke objekter som er naturlig å jobbe på, og hvordan de skal aksesseres, manipuleres og lagres, hvilket videre informerer oss om hvilke klasser om er hensiktsmessige å bygge opp løsningen rundt. Da er det viktig å skille ut ansvarsområde i forskjellige klasser, slik at klassene ikke blir for avhengige av hverandre. Java har allerede gode konvensjoner for hvordan klasser skal designes for å være både solide og forståelige [1][2]. I denne løsningen blir det nødvendigvis en grad av avhengighet, da vi har en Item-klasse som vil brukes i en Register-klasse som lager et register for å holde på Item-objekter. Dette vil gi oss en viss kobling[3] («coupling») mellom objekter og klasser. Vi ønsker i hovedsak av koblingen mellom enheter skal være «løs» («loose»), slik

For å oppnå løsere kobling, som jo et av de designprinsippene som etterstrebes, kan man skille de forskjellige klassene mest mulig, ved f.eks å gjøre at Register-klassen ikke manipulerer Item direkete, men via å lage objekter av den som lagres i en type liste i Register-klassen. Videre kan man oppnå lav kobling ved å ikke tillate Register-klassen å gi direkte tilgang til register-objektet, men kun returnere en dyp kopi[4] av den aktuelle listen. Dette vi i praksis bety at ingen klasser vil få tilgang til annet enn å lese en kopi av den «originale» listen med Items, med mindre det gjøres via metodene som er definert for å manipulere originallisten.

I samme åndedrag er vi inne på konseptet «cohesion», hvor det er ønskelig at hver klasse og metode kun har ansvaret for én enkelt logisk oppgave, og en klasse skal kun håndtere én enhet. I denne oppgaven kan man innledningsvis se for seg Item, ProductRegistry og Client som de naturlige klassene å dele programmet opp i, siden dette er de tre instansene som naturlig faller ut av kravspesifikasjonen. Da har hver klasse én oppgave, hvilket er i tråd med prinsippet om «høy cohesion»[5]. Om man i stedet lot f.eks ProductRegistry lage både et register *og* varen («item»), blir det hele brått mer uoversiktlig i tillegg til at man øker kompleksiteten til klassen. Likens med metodene, er det ønskelig at hver enkelt metode gjør én oppgave. F.eks å legge til en vare. Dette gjør det enklere å skrive god kode, samtidig som det blir mer oversiktlig hva som skjer, og at man minimerer sjansen for sikkerhetshull som følge av unødvendig høy kompleksistet. Et eksempel på dårligere kode og cohesion ville vært om man hadde en kombinert metode for å både legge til og slette en vare.

Et vanlig prinsipp er modularisering og innkapsling av data[6]. Vi setter variabler, metoder og klasser som ikke skal manipuleres direkte til *private*. Dette forhindrer at man får tilgang til dem, og deres variabler, uten å gå via de metodene som er tiltenkt å brukes for å arbeide på klassene. I denne oppgavens tilfelle er det spesielt klassen *Item* som er hensiktsmessig å sette til *private*, slik at den ikke kan endres eller brukes direkte fra klient-klassen.

Dokumentasjon av kode må også nevnes. Her er det JavaDoc-standarden[7][8] som dikterer hvordan god dokumentasjon skal føres, slik at kodens funksjoner blir forklart på en enkel og god måte.

5 METODE – DESIGN

Utviklerverktøy: JetBrains IntelliJ IDEA (IDE), CheckStyle (plug-in), Github (backup og revisjonskontroll).

Kilder: Java Volume 1 Fundamentals 12th Edition (lærebok), StackExchange, StackOverflow, Internett forøvrig, forelesningsnotater IDATT1001.

Fremdriftsplanen kan enkelt sies å gå ut på å ta hver deloppgave for seg, og utvikle applikasjonen i disse tre stegene. Selv om det var fristende å utvikle den ferdige applikasjonen med en gang, direkte ut fra alle tre deloppgavene samtidig, falt valget på at det ville bli mer oversiktlig å følge prosessen stegvis, selv om dette høyst sannsynlig ville bety at det ble omskrivinger av koden utover i deloppgavene.

Det var fra start ganske tydelig hvilke klasser og objekter som ville gi mening å benytte i denne oppgaven. Da jeg gikk i gang med å implementere Item-klassen ble den laget med det formål å bli brukt i en klasse som skulle håndtere et register av slike Item-objekter. De tre klassene *Item*, *ProductRegister* og *Main* (client) gir også mening fra det designteoretiske perspektivet, siden hver oppgave som er naturlig å utføre da får sin egen klasse. Dermed er det også enklere å eventuelt utvide funksjonaliteten ved å innføre ytterligere klasser, dersom behovet for dette skulle melde seg i de senere deloppgavene.

Den første delen av oppgaven (del 01) la grunnlaget for hvordan en vare skulle registreres, og var således ikke en stor oppgave å gjøre. Det viktigste jeg så at måtte gjøres her var å sette tilgangsprivilegier på klassefeltene, i tillegg til å lage metoder for å hente verdier (*«getters»*) og for å endre verdier («*setters»*). Spesielt *setters* er viktige å implementere korrekt, og da bare lages for de attributtene som faktisk gir mening å kunne endre på.

Videre bet jeg meg merke i at det ikke var bedt om en egen variabel for bredde på varene. Siden de to andre dimensjonene, høyde og lengde, var spesifisert, var det for meg merkelig at den tredje skulle mangle. Dette ble notert ned i kommentar i koden som en «*to do*».

I oppgavens andre del (del 02) ble det skissert hvilke metoder som var ønskelig å implementere for å manipulere data, i tillegg til at selve registreret som skal holde på Item-objekter skulle settes opp. Her gav oppgaven frihet til å selv velge hvordan registeret skulle implementeres, dvs hvilke klasser fra Java-biblioteket man selv syntes egnet seg best til å løse oppgaven. Min første tanke gikk da til å benytte *ArrayList*, siden dette er en dataklasse som har blitt benyttet i tidligere øvinger dette semesteret. Det vil si at jeg har kjennskap til hvordan den fungerer og hvordan man kan jobbe med det. Andre alternativer ville vært å bruke f.eks *HashMap*. Etter å ha gjort litt grunnelggende research på hash map, og forskjellene mellom HashMap og ArrayList vurderte jeg det slik at det var hensiktsmessig å jobbe med en kjent dataklasse, i stedet for å bruke en ny og ukjent klasse. Dette gjør jo selvfølgelig at jeg ikke lærte meg å bruke HashMap, men tanken er at nye dataklasser og metoder er best å lære seg utenfor faktisk applikasjonsbygging, slik at man ikke må bruke tid på å lære seg dette samtidig som det skal implementeres. Selvfølgelig, om dette hadde vært en ekte applikasjon som skulle lages, hadde det nok blitt investert med tid i å undersøke om f.eks HashMap har noen fordeler, om enn små, i akkurat denne situasjonen som den mest optimale dataklassen, og eventuelt lære seg å bruke den, fremfor å velge fortrinnsvis basert på hva en allerede kan. Avveiningen her er ble noe pragmatisk

Planen ble altså å komponere en ArrayList som holdt på datatypen Item, og inneholdt Item-objekter. Hvis man ser på dataregisteres som om det eksisterte i den virkelige verden, blir det riktig å si at varene (Item) ikke kan eksistere separat fra lageret som holder på dem (ProductRegister). Dermed er det naturlig at man bruker komposisjon for samarbeid mellom objekter. En vare eksisterer, for alle praktiske formål, kun så lenge den er på lageret;

programmet skal ikke håndtere varer som ikke er fysisk til stedet. Da følger det at programmet som skal håndtere varene burde følge samme prinsipp.

Et annet viktig prinsipp jeg hadde i bakhodet, som jeg lærte fra Arbeidskrav 11[9] i Programmering 1, var at selve ArrayList-en ikke skal være aksesserbar utenfor noen av motodene som er definert for å manipulere den. Det vil si at metoden for å hente ut og lese registeret (*getInventoryOfItems*) må returnere en kopi, og da spesifikt en *dyp kopi* («deep copy») av listen. Dette vil si at det lages en helhetlig, separat kopi av listen, i stedet for en såkalt *grunn kopi* («shallow copy») hvor man bare lager en kopi hvis referanse er til den originale listen. Hele listens innhold på altså kopieres og legges inn i en helt ny liste, som da kan returneres via *getter-metoden* til listen.

Metoder for å manipulere registeres skal også ha *minst mulig funksjonalitet*, det vil si at hver metode skal helst utføre kun én oppgave.

I deloppgave 3 ble det åpnet for å sette sitt eget preg på applikasjonen, samt bruke *enum* for å håndtere kategorier. Dermed brukte jeg tid på å lære meg å bruke enum, i tillegg til å notere ned de tingene jeg anså som hensiktsmessige utvidelser av den opprinnelige oppgaven.

Der hvor hovedtyngden av arbeidet med å bygge applikasjonen ble planlagt gjort i del 2, skulle del 3 brukes til refaktorering, finpuss og optimalisering av koden.

6 RESULTATER

Applikasjonen ble i stor grad funksjonellt ferdigstillt i deloppgave 2, hvor metodene for manipulering av registeret ble implementert. Fremgangsmåten min var å først få en oversikt over hvilke metoder som krevdes, ut fra den funksjonaliteten som be etterspurt til *brukergrensesnittet*. Jeg benyttet *switch* for å skissere rammen for en brukermeny, og satte opp de forskjellige funksjonene beskrevet i oppgaven. Dette informerte videre hvilke metoder som måtte skrives for å samsvare med menyen.

Etter at jeg hadde satt opp en kladd av menyen, skrev jeg metoder uten innhold for å få en liten oversikt over kodens struktur. Jeg gjorde deretter enkle diagramskisser og kladding på papir for visualisere hvordan metodene skulle interagere med klasser og hverandre, slik at jeg kunne planlegge strukturen og rekkefølgen det ville være hensiktsmessig å skrive dem i.

Det var åpenbart at det ville være gunstig med i alle fall to toString-metoder; en som gav den fulle og hele informasjonen om hver enkelt vare i registerert, og én med kompakt og konsis informasjon som egnet seg bedre til en kompakt oversiktsliste.

Det var åpenbart at det ville være gunstig med i alle fall to *toString*-metoder; én som gav den fulle og hele informasjon om hver enkelt vare, og én med kompakt og konsis informasjon som egnet seg bedre til en kompakt oversiktsliste.

Jeg innså tidlig at en egen metode for å søke etter og returnere en enkelt vare ville være nyttig. Siden de fleste av funksjonene er avhengig av å lokalisere en vare for å arbeide på den, ville dette være et effektivt og hensiktsmessig valg for å slippe å implementere søkefunksjonalitet i hver eneste metode. Oppgaven etterspurte en funksjon for å søke på «varenummer og/eller beskrivelse». Denne ble laget som en generell søkefunksjon for å vise varene med delvis match til søkeordet, enten som del av varenummer eller som del av beskrivelsen. I tillegg anså jeg det som et bra valg å implementere en metode som søkte spesifikt på korrekt serienummer. Denne skulle benyttes for å utføre endringer på en bestemt vare. Tanken bak dette var at hvis en

ønsker å endre på varedata, eller kanskje slette en vare, bør en kreve at den aktuelle varen kan spesifiseres nøyaktig av den som ønsker å utføre endringen. Funksjonene er da satt opp slik at man kan søke etter en delvis match, eventuelt skrive ut en liste over alle varene, for å finne varenummer (serienummer) på varen man ønsker å endre. Deretter går man til funksjonen for å f.eks slette en vare, og taster inn serienummeret når funksjonen etterspør det. Dermed er det, med overlegg, lagt inn et lite ekstra steg for å kunne utføre endringer på varer. Med større kunnskap om programmering og konstruksjon av metoder og brukergrensensitt, kunne nok dette blitt gjort via f.eks å søke opp et utvalg varer på delvis match, deretter velge en av disse, og så få opp en meny med handlinger som kan utføres. Dette ville dessverre kreve en del mer know how enn jeg innehar per dags dato, men hadde vært en mye mer brukervennlig måte å håndtere det på.

I deloppgave 3 ble det som nevnt tidligere, åpnet for å legge sitt eget preg på løsningen. I tillegg til å utvide funksjonaliteten, skulle jeg her finpusse på eksisterende funksjoner. Spesifikt planla jeg å legge inn funksjoner som sjekker om en vare allerede eksisterer i registeret når man legger inn en ny, for å unngå duplikater. Videre skal det implementeres metoder for å teste den dype kopien av registeret som returneres i «getInventoryOfItems», både for å sjekke at det blir en faktisk (dyp) kopi, og å sjekke om den er mutabel eller immutabel (Read/Write).

Etter noe research ble det tydelig at metoden for å lage en dyp kopi av en ArrayList faktisk krever litt mer jobb enn forventet. Dermed måtte det en del prøving og feiling til, samt testmetoder, for å faktisk lage en kopieringsfunksjon som fungerte som ønsket. Med veldig varierende kvalitet på eksemplene som er å oppdrive på Internett, tok det mange forsøk før jeg kom frem til noe som fungerte. Her ble det en override-metode for clone-funksjonen som ble løsningen.

Også funksjonen for å sjekke om en vare allerede eksisterer i registeret viste seg å være mer innviklet enn først antatt. Her også kom Internett til kort, med mange dårlige og alt for generelle eksempler. Avanserte funksjoner som «reflection» ble foreslått av en bekjent som til daglig jobber med C#, og har tidligere erfaring med Java, men dette var veldig ukjent territorium. I stedet ble det, også her, skrevet en egenimplementasjon av eksisterende funksjoner, nemlig HashCode.

Da jeg begynte å undersøke enum-klassen, ble det tydelig at man ikke kan bruke tall som «navn» når man tilordner kategorier via enum. I stedet må man legge dette inn som en verdi som holdes inni en annen verdi. Jeg valgte da å navngi kategoriene med navn. For at enum skulle fungere med forskjellige input, altså å kunne tilordne kategori ved å skrive inn både «1» og «Window» i input-menyen, måtte det lages egne funksjoner for dette i enum-klassen.

Begrensningen i spesifikasjone om at gulvkategorien skal hete «Floor Laminate» gikk jeg bort fra, og gave den heller navnet «Flooring», siden dette er det korrekte, generelle, begrepet for trevarer for gulv.

Testing av applikasjonen ble i hovedsak gjort av meg selv, under utviklingen. Etterhvert som applikasjonen ble funksjonell, ble et par familiemedlemmer og noen nære venner engasjert i å forsøke å bruke programmet, og etterpå rapportere om sin brukeropplevelse; hva som fungerte, hva som ikke fungerte. Om det var noe som fungerte annerledes enn de forventet, og hva som eventuelt kunne vært gjort enklere eller forbedret. I tillegg fikk jeg en «code review» utført av et familiemedlem som jobber med programmering. Her ble det oppdaget for eksempel at det var mulig å slette flere enn en vare om gangen, fordi jeg hadde brukt feil søkemetode for å lokalisere varen; den generell i stedet for den spesifikke, og det ble returnert en liste med varer i stedet for bare den ene varen.

En annen ting som ble tatt tak i var rabattfunksjonen, og hvordan denne best kunne implementeres. Slik oppgaven formulerte det, var det ikke mange muligheter utenom å direkte endre den eksisterende prisen. Ideer om å legge til en ekstra attributt for rabattert pris, lage et eget register som holdt på rabatterte varer, eller en kombinasjon, var det som tidlig slo meg som de enkleste løsningene. Løsningen jeg landet på var å sette en egen «Final Price» attributt i Item-klassen. Denne skulle reflektere prisen man betaler i kassa. Denne settes i utgangspunktet til å være lik Item Price. Rabatt-funksjonen kan deretter benyttes for å endre Final Price, enten med en prosentvis endring (f.eks 20% rabatt) eller en heltallsendring (f.eks 150 kr avslag). Dette viste seg å være en enkel og grei løsning, siden man da beholder den opprinnelige prisen i Item-objektet, men kan også vise en rabattert pris.

Mot slutten så jeg også at jeg hadde enkelte metoder til «undermenyer» som var skrevet inni switch-menyer. Selv om disse i utgangspunktet var små metoder, vurderte jeg at flere, om ikke alle, med fordel kunne flyttes ut til egne metoder, og at menyen heller videresendte til den aktuelle metoden, slik som hovedmenyen gjorde.

En ny review av koden viste at jeg brukte mange unødvendige og uklare «throw new Exception». Disse ble byttet ut med returverdier av typen *true/false* eller *null*, og metodene som fikk disse heller kunne prosessere disse logiske operatorene for å velge å sende en feilmelding eller gå videre i koden. Dette førte til en del refaktorering, men totalt sett ble koden enklere. I tillegg ble vanlige for-loops byttet ut med såkalt *enhanced for-loop*.

God kode-stil ble også verifisert ved å sjekke lesbarhet for andre (review), og å formattere i henhold til CheckStyle med Google-stilen.

Selve brukergrensesnittet var alltid tiltenkt å være av typen TUI (Terminal User Interface) i stedet for et GUI (Graphical User Interface). Dette ganske enkelt fordi det er enklere å programmere, siden vi ennå ikke har lært å bruke mer avanserte verktøy for grensesnitt. I tillegg er det jo et enkelt og forståelig design, uten mye rom for feil. Å skrive inn et tall som representerer menyvalg, og deretter skrive inn informasjonen som etterspørres er enkelt og brukervennlig, selv om det kanskje ikke gir følelsen av spesielt «moderne» programvare. Men samtidig er TUI som regel ganske enkle å forstå seg på, takket være at det ikke tilbyr så mange og kompliserte knapper og funksjoner.

En spesiell funksjon jeg anså som lurt å implementer var en dobbel bekreftelse for å slette en vare. Først må man velge menyen for å slette. Da får man spørsmål om man er sikker på at man vil inn i slette-menyen. Deretter søker man opp varen man ønsker å slette. Før denne slettes får man enda en dialog for å bekrefte, hvor man kan velge å ikke gjennomføre. Dette er en vanlig måte å designe slike løsninger, og selv om den ikke er helt «idiot-sikker», er den såpass tydelig at den bør forhindre de fleste tilfeller av sletting ved feiltagelse.

En ting jeg strevde veldig med, i del 3, var å implementere en måte å sjekke om det eksisterte duplikater i ArrayListen før en ny vare ble lagt til. Siden Java sine metoder for dette er veldig enkle, tok det meg faktisk flere dager med research for å finne en måte som fungerte. Å bare bruke vanlig .equals, .contains, .matches og så videre, lot seg ikke bruke på Item-objektet på samme måte som jeg tidligere hadde brukt dem på å sjekke primitive datatyper. Selv om jeg gikk tilbake og sjekket at jeg hadde bygget for-looper og itererte korrekt gjennom ArrayList, ville det ikke fungere. Å kjøre filene i Debug-mode viste at for-loopen ble hoppet over, fordi sjekken som skulle avgjøre om newItem var likt et eksisterende objekt i registeret ikke fungerte som den skulle. Det lot seg tydeligvis ikke gjøre å kjøre en enkelt sjekk som jeg tidligere hadde benyttet i funksjonen som søker etter en vare basert på Item ID Number. Dette kan selvfølgelig være fordi jeg hadde satt opp if-sjekken eller for-loopen feil, eller andre feil i den logiske konstruksjonen.

Det som var en gjenganger i alle instruksjonene jeg fant på Internett, var at man var avhengig av å konstruere såkalte «override»-metoder for å benytte *equals* og *contains* for å effektivt sjekke etter duplikater av objekter i ArrayLister. Alternativt benytte en annen klasse for å holde på dataene enn en ArrayList. Sånn sett ville kanskje et HashSet vært en bedre løsning, siden et Set ikke tillater duplikater. Men til gjengjeld er den noe mer komplisert å jobbe med, siden den er basert på nøkler i stedet for en fast indeks, og lar seg ikke bli iterert over like enkelt som en ArrayList for å finne objekter.

Det jeg endte opp med var å konstruere to override-metoder: en for *equals* og en for *hashCode*. I metoden for å legge til en vare laget jeg en logisk sjekk, hvor metoden sendte hashCode av «itemIDNumber» (*itemIDNumber.hashCode()*) til en separat metode for å sjekke for duplikater. Der itererte metoden gjennom objektene i register-listen, via en for-loop. Inni for-loopen var det en ny if-sjekk, som sammenlignet hash-en til ItemIDNumber fra objektet på plass «i» i listen med hash-en som metoden fikk inn fra «addNewItem»-metoden. Deretter returnerer checkForDuplicate enten true eller false. True betyr at hash-ene er like, og false betyr at de er ulike. Når addNewItem får tilbake true eller false, går den videre i if-sjekken. Om den får tilbake false, vil den legge til den nye varen. Om den får true, vil den gå videre, og selv returnere en «false» til addNewItem i Main/Client klassen. Har skriver metoden ut en melding som sier hvorvidt varen ble lagt til eller ikke. Dermed ble resultatet at applikasjonen sjekker for duplikater, kun basert på ID-nummeret til varer. Dette er i praksis en ganske fornuftig løsning. Om man har jobbet på lager, eller bare sett rundt om kring på nettbutikker, oppdager man fort at det faktisk er mange varer og produkter som er svært like, både i beskrivelse og dimensjoner, men som skilles på forskjellig ID- eller serienummer og kanskje en liten variasjon i «navn». I praksis oppnår man det samme ved å sjekke den parameteren som MÅ være unik (ID/serienummer) som ved å sjekke hele objektet som helhet. Så lenge ID-nummeret er ulikt, vil det helhetlige objektet også være ulikt.

Enum var også en utfordring, siden forklaringene man finner sjeldent passer akkurat til det du forsøker å gjøre. Og når man ikke har spesielt mye erfaring med Java, blir det vanskeligere å generalisere. Men etter mye prøving, feiling, lesing og spørring, fikk jeg bygget klassen, og skrevet om resten av koden slik at enum-implementasjonen av Item Category ble benyttet.

Klasser:

Item.class	En «item» er en «vare». Denne holder på vare-data, som pris, størrelse, beskrivelse, osv. Disse lagres i et register.
ProductRegister.class	Klassen som implementerer en ArrayList som holder på data av typen Item. Altså et vareregister. Her finner vi metodene som er skrevet for å manipulere registeret og Item-objektene.
Main.class	Klassen hvor brukerinteraksjon foregår. Brukermeny, med metoder for å aksessere, sende og motta informasjon fra ProductRegister-klassen.
Category.class	Enum-klasse som representerer varekategorier. Muliggjør å tildele kategori både via tall-input og string.

7 DRØFTING

Applikasjonen ble gjennomført i henhold til både plan og kravspesifikasjon, og jeg er totalt sett fornøyd med arbeidet som har blitt lagt ned i den. Med tanke på det nivået av Java-kunnskap jeg hadde før jeg begynte på den, har det egentlig gått over all forventning. Det største problemet med å skulle skrive dette programmet har vært når jeg har måttet søke informasjon og kunnskap om hvordan jeg skal implementere funksjonalitet jeg ikke får til å fungere. Eksemplene som gis på f.eks internett er ofte veldig begrenset, og omhandler som regel bare de enkleste implementasjonene, og med alle mulige varianter er det ofte vanskelig å vite hvilket som er korrekt, og ikke minst relevant for akkurat denne løsningen.

Den største kilden til feil i programmet er nok likevel meg selv, og min manglende erfaring med Java og objektorientert programmering. Selv om jeg gjør mitt beste for å følge de prinsippene om god og sikker kode som jeg har lært gjennom arbeidskrav, forelesningsnotater og lærebok, er man alltid litt i tvil på om man har forstått det riktig, hvilket jo vil påvirke implementasjonen av det.

Det sagt, så har jeg gjort mitt beste for å følge opp de prinsippene vi har lært, både med minimere tilgang til objekter, klasser og metode, bygge koden etter anerkjente konvensjoner for kode-stil og å skrive god dokumentasjon iht. JavaDoc-standarden.

8 KONKLUSJON – ERFARING

Programmet ble, totalt sett, bra. Selv om det nok er mye som kunne vært gjort bedre, er jeg veldig fornøyd med læringen jeg har hatt gjennom å jobbe med det.

Om jeg skulle laget løsningen på nytt, ville jeg nok lagt mer tid ned i å undersøke alternativer til ArrayList for å holde på registeret. Som nevnt, hadde kanskje et HashSet vært mer hensiktsmessig, med tanke på håndtering av duplikater. I tillegg ville jeg ha implementert de ekstra Item-attributtene tidlig. Spesifikt, at jeg ikke ble nødt til å legge til attributter rundt om kring hvor Item-klassen ble brukt, og å slippe å skrive om metodene for pris og rabatt. Men å gjøre denne omskrivingen var, sånn sett, en del av oppgaven.

Et annet godt prinsipp å ha i bakhodet er å ikke begynne å skrive metoder inni en *switch*. Dette baller fort på seg, og blir veldig fort uoversiktlig. Da er det bedre å bare skrive dem utenfor med en gang, spesielt om disse metodene gjør forskjellige ting. Da er det lettere å sørge for at det ikke skjer noe utilsiktet.

Begrensninger i applikasjonen har jeg i stor grad forsøkt å holde til det jeg anser som «fornuftig». Altså at man ikke kan ha negative tall på lager, at man ikke kan slette mer enn en vare om gangen, og at alle varer må ha et unikt ID-nummer. En annen bevisst begrensning er at det ikke er mulig å endre ID-nummeret til en vare (Item). Dette var noe jeg diskuterte litt med meg selv. På den ene siden er det jo gunstig å kunne gjøre denne endringen, i tilfelle man skriver feil. Men samtidig burde et ID-nummer være noe statisk, hvor det ikke skal være enkelt å endre verdien. Dette er jo for så vidt enkelt å implementere, om nødvendig, og kunne kanskje vært tilgjenglig i en «Admin»-versjon av programmet.

Slik varekategorier er implementert, kan man legge til nye kategorier uten å måtte gjøre andre endringer enn å føre den opp i Category-klassen. Dette kan ikke gjøres via programmets meny, men må skrives i selve klasse-filen.

Totalt sett fungerer programmet ganske bra, og de brukertestene jeg har utført har både avslørt feil, som har blitt rettet opp i, og har vist at det i grunnen er et system som er enkelt å

bruke. Sistnevnt er nok mye takket være at programmet i seg selv er nokså begrenset, med et brukergrensensitt som ikke gir rom for spesielt mye forvirring rundt hva funksjonene gjør.

Dersom noe skulle forbedres i fremtiden, kunne det nok vært å bytte ut ArrayList med en mer egnet klasse for å holde på data.

9 REFERANSER

[1] Core Java Volume 1: Findamentals, 12th Edition, Cay S. Horstmann, kapittel 4.11 - Class Design (ISBN-13: 978-0-13-767362-9).

[2] Core Java Volume 1: Findamentals, 12th Edition, Cay S. Horstmann, kapittel 4.1.4 - Relationships Between Classes (ISBN-13: 978-0-13-767362-9).

[3] Forelesning 10 (side 5), IDATT1001, M. A. Norzi – Coupling.

[4] Forelesning 06 (side 16), IDATT1001, M. A. Norzi - Deep Copy.

[5] Forelesning 10 (side 5), IDATT1001, M. A. Norzi – Cohesion.

[6] Forelesning 05 (side 22), IDATT1001, M. A. Norzi – Innkapsling.

[7] Core Java Volume 1: Findamentals, 12th Edition, Cay S. Horstmann, kapittel 4.10 - Documentation (ISBN-13: 978-0-13-767362-9).

[8] Forelesning 10 (side 36), IDATT1001, M. A. Norzi – JavaDoc.

[9] Arbeidskrav 11, IDATT1001.