STUDENTRAPPORT

NTNU

KANDIDATNUMMER(E)/NAVN:			
Ola Henrik Otterlei Navelsaker			
DATO:	FAGKODE:	STUDIUM:	ANT SIDER:
23.11.2022	IDATG1001	IDATG	16

FAGLÆRER(E) :
Kiran Bylappa Raja

TITTEL :
Applikasjon for varelageret til Smarthus AS

SAMMENDRAG:
Gjennom denne oppgaven har det blitt utviklet en applikasjon for Smarthus AS. Hensikten med denne applikasjonen er å fungere som et varelager hvor brukeren kan legge til varer og manipulere informasjon i lageret. Denne rapporten beskriver hvordan dette ble gjennomført og forklarer ulike design valg som har blitt gjort underveis i utviklingen av applikasjonen, og vurderer disse valgene.

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

INNHOLD

1 SAMMENDRAG 1
2 INNLEDNING – PROBLEMSTILLING 1
2.1 Bakgrunn/Formål og problemstilling 1
2.2 Avgrensninger 2
2.3 Begreper/Ordliste 2
3 BAKGRUNN - TEORETISK GRUNNLAG 3
3.1 Cohesion 3
3.2 Coupling 3
3.3 Code duplication 4
3.4 Responsibility-driven design 4
3.5 Abstraction and Modularization 4
3.6 Encapsulation 4
3.7 Refactoring 4
4 METODE – DESIGN 5
5 RESULTATER 6
5.1 Beskrivelse av programmet 6
5.1.1 Oversikt 6
5.1.2 Product klassen 7
5.1.3 Register klassen 7
5.1.4 ProductRegisterUI 7
5.2 Designvalg 7
5.3 Robusthet 9
5.4 Brukervennlighet 9
5.5 Dokumentasjon 9
5.6 Refaktorering 10
5.6.1 Pakkestruktur 10
5.6.2 Test data 10
5.6.3 Kode duplisering 11
5.7 DRØFTING 12
5.7.1 Kode opp mot teori 12
5.7.2 Robusthet 13
5.7.3 Brukervennlighet 13
6 KONKLUSJON - ERFARING 13
7 REFERANSER 14

1 SAMMENDRAG

Rapporten omhandler en mappeoppgave utgitt i faget IDATG1001 Programmering 1 hvor hensikten med mappeoppgaven er å lage en applikasjon til Smarthus AS som skal fungere som et varelager der brukeren kan legge til/endre/slette produkter i lageret. For å utvikle kode og sørge for at koden ikke strider mot gode design prinsipper har utviklingsmiljøet IntelliJ IDEA sammen med plugins CheckStyle, SonarLint og Git vært benyttet. Den ferdigutviklede applikasjonen inneholder den nødvendige funksjonaliteten beskrevet i kravspesifikasjonen, med noe forbedringspotensiale når det gjelder brukervennlighet. Rapporten beskriver tankeganger og vurderingene som er gjort gjennom utviklingsprosessen, samt endringer som ble gjort i koden for å følge gode design prinsipper.

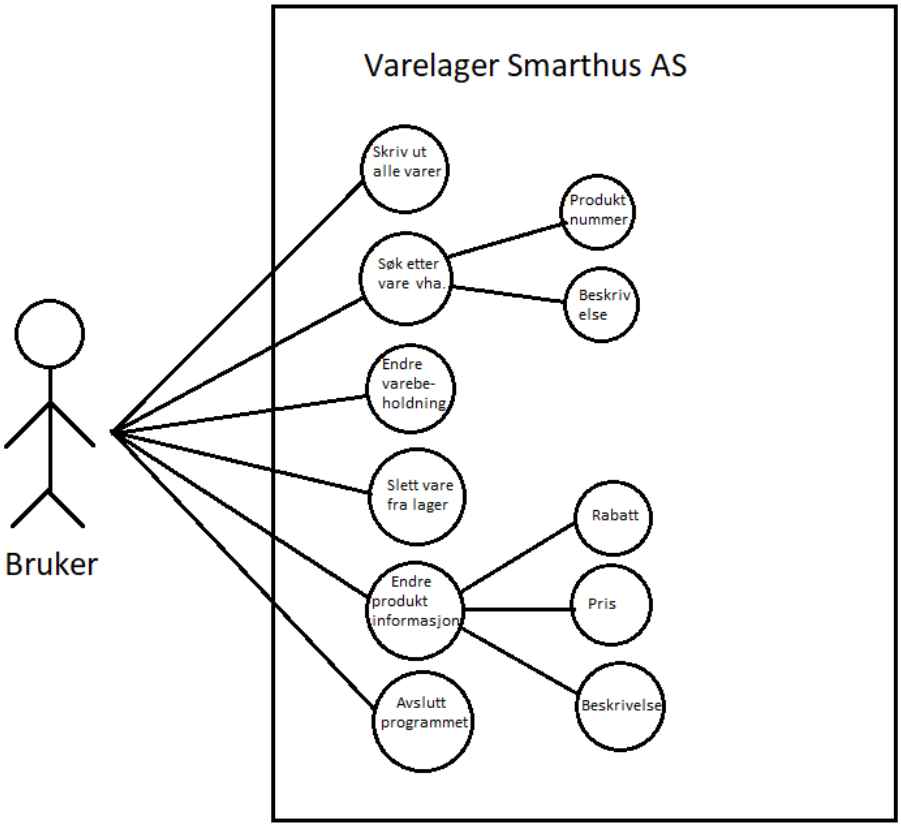
2 INNLEDNING – PROBLEMSTILLING

2.1 Bakgrunn/Formål og problemstilling

Denne oppgaven er gitt som en mappeoppgave i faget IDATG1001 med tre deler hvor det skal utvikles en programvare som vil benyttes av et varehus – Smarthus AS. Smarthus AS Leverer hovedsakelig varer til bygg-industrien, så typiske varer er laminatgulv, dører og vinduer, lister og annet trevirke. Løsningen skal bestå av et tekstbasert brukergrensesnitt og et register som lagrer informasjon om varene. Koden skal verifiseres med CheckStyle-plugin for IntelliJ og bruke kodestilen til Google. Koden skal være på engelsk og alle klasser, metoder og variabler skal ha gode, beskrivende navn som gjenspeiler hvilken tjeneste en metode tilbyr, eller hvilken verdi variablene representerer/holder på.

Det skal implementeres en klasse som representerer en vare og holde på følgende informasjon for en vare: varenummer, beskrivelse, pris, merkenavn, vekt, lengde, høyde, farge, antall på lager og kategori. I tillegg skal det implementeres et vareregister som holder på en eller flere varer. Vareregisteret skal ha de nødvendige metodene slik at brukergrensesnittet kan ha følgende funksjonalitet: Skrive ut alle varer på lageret, søke etter en gitt vare basert på varenummer/beskrivelse, legge til en ny vare i registeret, endre varebeholdningen for en vare, slette en vare fra varelageret, endre rabatt/pris/varebeskrivelse for en vare.

Under er et aktivitetsdiagram som illustrerer hvilke muligheter og funksjonaliteter brukeren skal ha når han benytter seg av programmet.



Figur 1: Aktivitetsdiagram

2.2 Avgrensninger

Programvaren må utvikles i programmeringsspråket Java, og koden må skrives på engelsk. Brukergrensesnittet for programvaren må være tekstbasert, vanligvis ville et mer brukervennlig brukergrensesnitt være å foretrekke.

2.3 Begreper/Ordliste

Begrep (Norsk)	**Begrep (Engelsk)**	**Betyding/beskrivelse**
Produkt	Product	Varehuset skal håndtere produkter inn og ut av et lager
Lager	Storage	....
	Accessor	En type metode som har til hensikt å hente et felt, gjerne privat, tilhørende objektet og returnere det. Gjør at en kan beskytte feltene ved å bruke private.
	Mutator	En type metode som har til hensikt å endre et privat felt tilhørende objektet.
Felt	Field	Et felt er en karakteristikk av en klasse. Eksempel på felter i Product klassen er: productNumber, price, description, category. Felter holder på verdier om objektet.
Konstruktør	Constructor	Oppretter en instanse av et objekt av en klasse.
Pris	Price	Prisen til produktet. Produktet har et felt som heter Price.
Beskrivelse	Description	Beskrivelse av produktet. Produktet har et felt som heter Description

Begrep (Norsk)	Begrep (Engelsk)	Betyding/beskrivelse
Produkt nummer	Product number	Produkt nummeret til produktet
Produkt spesifikasjoner	Product Specifications	De fysiske dimensjonene til et produkt + fargen. ProductSpecifications klassen holder på dimensjonene til et produkt.
Nøkkel	Key	Key blir brukt ved lagring av data i et HashMap. Nøkkelen må være unik og kan brukes til å hente data som er assosiert med den spesifikke nøkkelen.
Bibliotek	Library	Libraries inneholder klasser og funksjoner som kan brukes ved å importere de.
Programvareutvidelse	Plugin	Plugins gir ekstra funksjonalitet til utviklermiljøet, som for eksempel en automatisk sjekk av kode.
Sekvensdiagram	Sequence diagram	Et diagram som viser steg for steg hvordan en metode fungerer og samhandler med andre klasser/metoder.
Abstrahering	Abstraction	Handler om å ignorere små detaljer for å fokusere på det store bildet.
Modularisering	Modularization	Handler om å bryte opp et problem i flere mindre problem som er enklere å løse hver for seg.
Samhold	Cohesion	Samhold går ut på hvor spesialiserte metoder/klasser er, metoder gjør kun en ting og klasser har kun et ansvar.
Kobling	Coupling	Kobling går ut på hvor sammenkoblet klassene er. Klassene skal kunne fungere alene slik at endringer i en klasse ikke skal gjøre at andre klasser slutter å fungere.

3 BAKGRUNN - TEORETISK GRUNNLAG

3.1 Cohesion

"The term *cohesion* relates to the number and diversity of tasks for which a single unit of an application is responsible. Cohesion is relevant for units of a single class and an individual method. Ideally, one unit of code should be responsible for one cohesive task. A method should implement one logical operation, and a class should represent one type of entity. The main reason behind the principle of cohesion is reuse: if a method or a class is responsible for only one well-defined thing, then it is much more likely it can be used again in a different context." (Barnes og Kölling, 2017, s.260)

3.2 Coupling

"The term *coupling* refers to the interconnectedness of classes. We aim to design our applications as a set of cooperating classes that communicate via well-defined interfaces. The degree of coupling indicates how tightly these classes are connected. We strive for a low degree of coupling, or *loose coupling.* The degree of coupling determines how hard it is to make changes in an application. In a tightly coupled class structure, a change in one class can make it necessary to change several other classes as well. This is what we try to avoid, because the effect of making one small change can quickly ripple through a complete application." (Barnes og Kölling, 2017, s.259)

3.3 Code duplication

"Code duplication is an indicator of bad design. The problem with code duplication is that any change to one version must also be made to another if we are to avoid inconsistency. This increases the amount of work a maintenance programmer has to do, and it introduces the danger of bugs. It very easily happens that a maintenance programmer finds one copy of the code and, having changed it, assumes that the job is done. There is nothing indicating that a second copy of the code exists, and it might incorrectly remain unchanged." (Barnes og Kölling, 2017, s.261)

3.4 Responsibility-driven design

"Responsibility-driven design expresses the idea that each class should be responsible for handling its own data. Often, when we need to add some new functionality to an application, we need to ask ourselves in which class we should add a method to implement this new function. Which class should be responsible for the task? The answer is that the class that is responsible for storing some data should also be responsible for manipulating it." (Barnes og Kölling, 2017, s.271)

3.5 Abstraction and Modularization

"As a problem grows larger, it becomes increasingly difficult to keep track of all details at the same time. The solution we use to deal with the complexity problem is *abstraction.* We divide the problem into sub-problems, then again into sub-sub-problems, and so on, until the individual problems are small enough to be easy to deal with. Once we solve one of the sub-problems, we do not think about the details of that part any more, but treat the solution as a single building block for our next problem. This technique is sometimes referred to as *divide and conquer."* (Barnes og Kölling, 2017, s.68)

"*Modularization* is the process of dividing large things into smaller parts, while abstraction is the process of ignoring details to focus on the bigger picture." (Barnes og Kölling, 2017, s.69)

3.6 Encapsulation

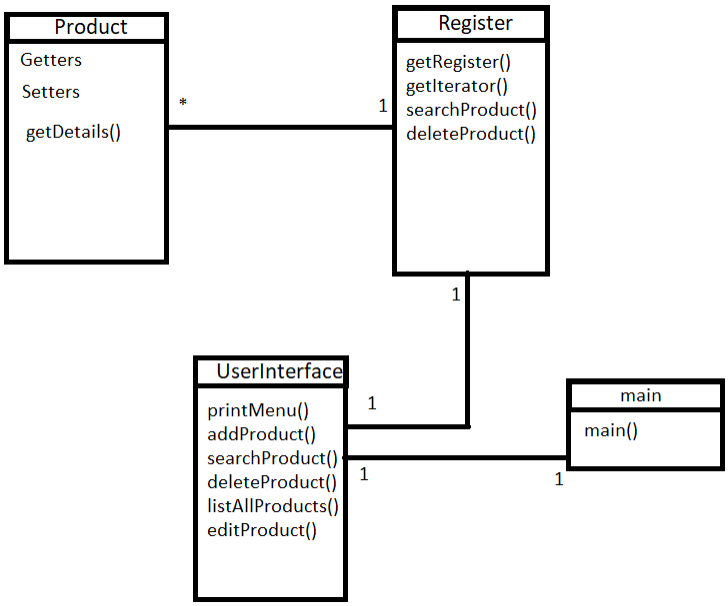
"The encapsulation guideline suggests that only information about *what* a class can do should be visible to the outside, not about *how* it does it. This has a great advantage: if no other class knows how our information is stored, then we can easily change how it is stored without breaking other classes. We can enforce this separation of *what* and *how* by making the fields private, and using an accessor method to access them." (Barnes og Kölling, 2017, s.266)

3.7 Refactoring

"Refactoring is the rethinking and redesigning of class and method structures. Most commonly, the effect is that classes are split in two or that methods are divided into two or more methods. Refactoring can also include the joining of multiple classes or methods into one, but that is less common than splitting." (Barnes og Kölling, 2017, s.282)

4 METODE – DESIGN

Oppgaven ble utgitt i tre deler, hvor første del bestod av å implementere klassen som skulle representere en vare og inneholdt en kort beskrivelse av hvordan applikasjonen vil se ut, nemlig at løsningen skal bestå av et tekstbasert brukergrensesnitt og et register som lagrer informasjon. Etter å ha lest denne delen, lagde jeg et kjapt klassediagram slik jeg såg for meg applikasjonen ville se ut.



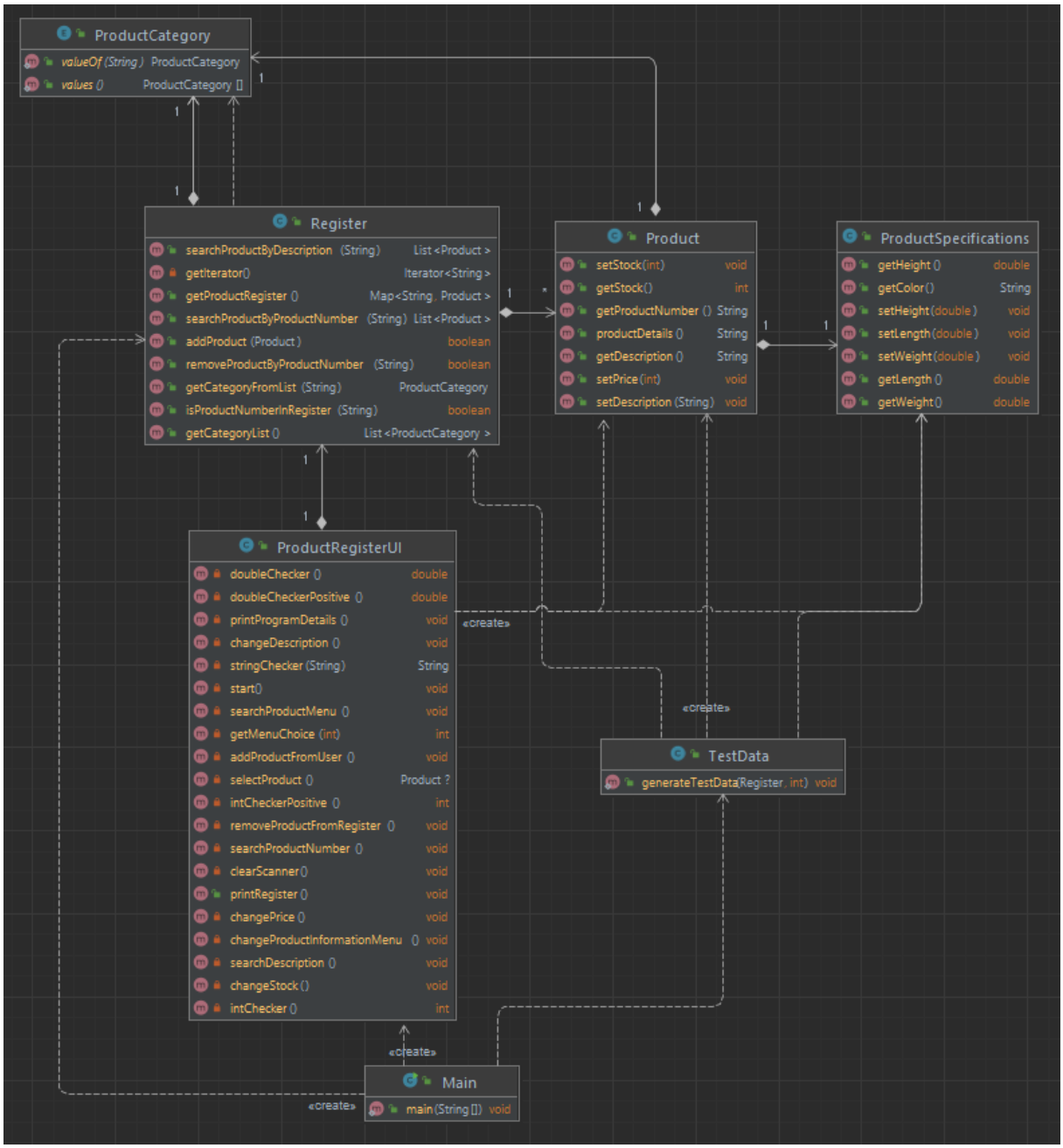
Figur 2: Tidlig klassediagram

Koden skulle inneholde en Product klasse som representerte en vare, Register klassen som representerte selve lageret og en UserInterface klasse som lagde og håndterte det tekstbaserte brukergrensesnittet. Jeg valgte å bruke utviklingsmiljøet IntelliJ IDEA med CheckStyle og SonarLint plugins for all utvikling og testing av kode som ble skrevet i Java versjon 8. Jeg benyttet meg også av versjonskontrollsystemet Git, som har den fordelen at en kan jobbe på tvers av datamaskiner og at hvis en endring gjør at koden slutter å fungere som den skulle, kan en enkelt gå tilbake til en eldre versjon som fungerer ved hjelp av Git.

5 RESULTATER

5.1 Beskrivelse av programmet

5.1.1 Oversikt



Figur 3: Klassediagram

Programmet har hovedsakelig tre klasser: Product, Register og ProductRegisterUI. Det er fullt mulig å løse problemstillingen i en klasse, men for å oppnå god kodekvalitet må en følge Product klassen representerer et produkt som bedriften har på lageret og har metoder for å innhente og endre nødvendig informasjon om produktet. Register klassen representerer selve lageret i bedriften ved å lagre og håndtere flere objekter av Product klassen. ProductRegisterUI klassen inneholder det tekstbaserte brukergrensesnittet som hjelper brukere med manipulering av data. For at Product klassen ikke skal holde på for mange felter blir det tatt i bruk en ekstra klasse, «ProductSpecifications», som holder på

vekt, lengde, høyde og farge. Denne klassen bruker setters, som Product klassen, slik at vekt, lengde og høyde ikke kan være negative tall. I tillegg holder enum klassen, ProductCategory, på en liste av godkjente kategorier.

5.1.2 Product klassen

Product klassen inneholder informasjon om: produkt nummer (productNumber), beskrivelse (description), pris (price), merke (brand), vekt (weight), lengde (length), høyde (height), farge (color), antall varer (stock), og kategori (category). Alle disse feltene er satt til «private» for å oppnå «encapsulation». Produkt nummer bruker datatypen String, på grunn av at de fleste produkt nummerene bruker både bokstaver og tall. Vekt, lengde og høyde bruker datatypen double, for å kunne ha med desimaler, da dette er normal for slike detaljer. Klassen har også de accessor og mutator metodene som er nødvendige for å oppnå kravspesifikasjonene, i tillegg til en metode som returnerer all informasjonen om et produkt slik at det er mulig å gi brukeren en tekstbasert representasjon av produktet. I konstruktøren til klassen blir enkelte setters benyttet slik at verdier som «price» og «stock» ikke kan bli satt til å være negative tall. Det samme blir gjort med lengde, høyde og vekt, men dette håndteres i en annen klasse kalt «ProductSpecifications». For å være sikker på at riktige kategorier blir opgitt benyttes en enum klasse kalt «ProductCategories». Når et objekt av Product klassen lages, vil kun kategorier som finnes i ProductCategories kunne benyttes.

5.1.3 Register klassen

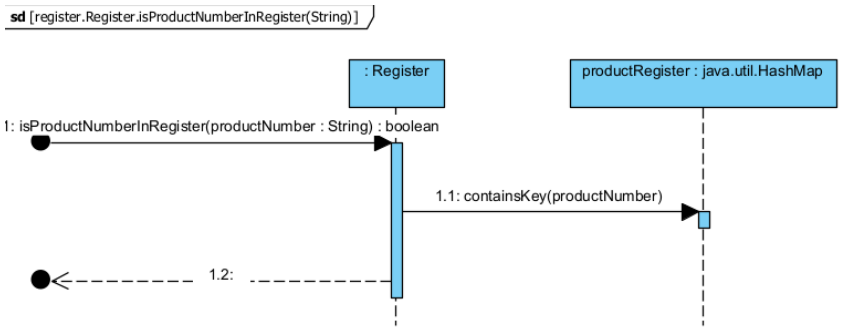
Register klassen lagrer flere produkt i et HashMap hvor productNumber feltet blir brukt som Key og Product objektet blir brukt som Value i HashMappen. Manipulering av data i registeret skjer kun i denne klassen for å følge prinsippet om ansvars-drevet design. Klassen har metoder for å legge til et produkt i registeret, finne et produkt gjennom beskrivelsen/produktnummer, skrive ut registeret, fjerne et produkt og sjekke om et produktnummer er i registeret. I tillegg har klassen accessor metoder for å returnere en liste med kategorier, finne kategori og en iterator for gjenbruk.

5.1.4 ProductRegisterUI

Alt som har med det tekstbaserte brukergrensesnittet og brukere å gjøre, skjer i denne klassen. Den har metoder som start(), searchProductMenu() og changeProductInformationMenu() som skriver ut brukergrensesnittet og bruker metoden getMenuChoice() for å hente hvilket meny element brukeren velger. Klassen inneholder alle metodene brukeren kan benytte i programmet i henhold til programspesifikasjonene. Det betyr at brukeren kan endre Price, Stock og Description feltene til produktet, han kan søke etter et produkt gjennom beskrivelsen eller produkt nummer, fjerne et produkt fra registeret, legge til et produkt til registeret og skrive ut alle produkt i registeret.

5.2 Designvalg

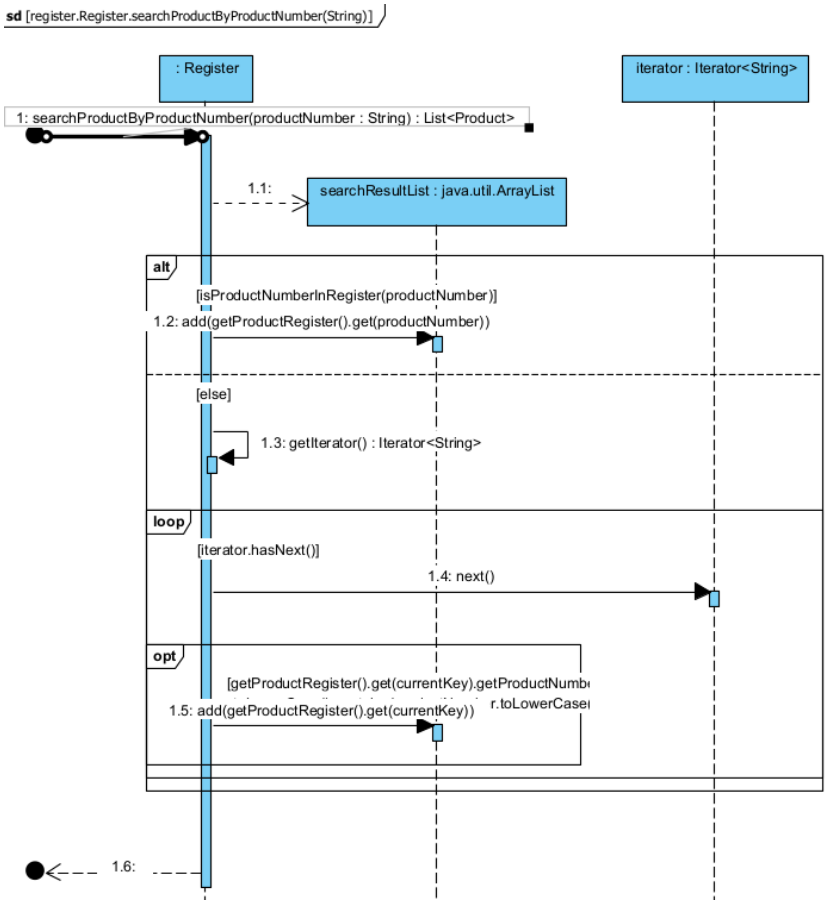
For å representere selve lageret valgte jeg å bruke HashMap i stedet for å bruke ArrayList. HashMap lagrer data ved å assosiere en verdi (produktet i dette tilfellet) med en unik «Key» (produkt nummeret), sammenlignet med ArrayList som lagrer data ved å legge data i en liste og holde på indeksen til objektet som blir lagt til listen. Det at «Key»en må være unik gjør at en ikke kan legge til duplikate «Key»er, men man kan legge til duplikate verdier (Produkt) med forskjellig «Key». For mitt program er dette en fordel, da det ikke er nødvendig å lagre duplikate verdier og gjør at det blir enkelt å sjekke om et produkt allerede finnes i registeret ved å bruke containsKey() funksjonen innebygd i HashMap biblioteket.



Figur 4: isProductNumberInRegister()

Sekvensdiagrammet illustrerer hvor enkelt det er å bruke HashMap til dette formålet. containsKey() metoden returnerer true eller false basert på om produkt nummeret som blir sendt med er i registeret eller ikke. Hvis dette skulle blitt gjort ved hjelp av ArrayList, måtte man ha laget en metode som sjekker om produkt nummeret er likt det produkt nummeret som blir sendt med, for hvert objekt i listen til den finner et likt produkt nummer.

Når brukeren søker etter et produkt ved hjelp av både produkt nummer og beskrivelse tenkte jeg at det var nyttig om brukeren fikk en liste av produkt som inneholdt deler av beskrivelsen/produkt nummeret. Spesielt for beskrivelser ville det være urealistisk å annta at brukeren husker hele beskrivelsen ordrett når han ønsker å finne et spesifikt produkt. Både søk gjennom produkt nummer og beskrivelser vil resultere i en liste med produkt, utenom når det finnes en eksakt match med produkt nummer. Under er et sekvensdiagram som viser hvordan searchProductByProductNumber() metoden fungerer.



Figur 5: searchProductByProductNumber

Metoden searchProductByProductNumber sjekker først om produkt nummeret som ble sendt med finnes i registeret, hvis det stemmer returnerer funksjonen kun produktet med det produkt nummeret. Hvis det ikke stemmer benytter metoden en iterator for å gå gjennom registeret og sjekker om produkt nummeret som ble sendt med finnes i deler av andre produkt nummer. For eksempel om en søker etter «AB32» og det er et produkt i registeret med produkt nummer «AB328543», så vil dette bli lagt til i listen og returnert, men kun hvis det ikke finnes et produkt med «AB32» som produkt nummer.

5.3 Robusthet

Målet, når det gjelder robusthet, er å sørge for at programmet ikke kan krasje. Kanskje den største trusselen når det gjelder dette er personen som bruker programmet. Derfor blir det benyttet input validation på alle verdier brukeren skriver inn i programmet. Input validation er prosessen hvor en sjekker at det som blir sendt inn til programmet er riktig skrevet inn, og at det ikke vil forårsake en feil. Programmet inneholder funksjoner som intChecker() og floatChecker() for å verifisere og sørge for at brukeren skriver inn gyldig tall der det er nødvendig. Hvis brukeren fikk lov å skrive inn hva som helst i et felt som krever datatypen int, hadde programmet krasjet og data kunne gått tapt.

I tillegg til å sørge for at programmet ikke krasjer, brukes input validation for å sørge for at brukeren skriver inn verdier som gir mening. Når brukeren forsøker å legge inn et nytt produkt, blir det sjekket at produkt nummeret ikke er tomt, at produkt nummeret ikke allerede finnes i lageret, og at tallverdier som pris og antall på lageret ikke er negative siden dette ikke hadde gitt mening.

En viktig del av robusthet er at brukeren ikke klarer å sette seg fast i programmet slik at han ikke kommer seg videre uten å avslutte og gjenåpne programmet. For å sørge for at dette ikke skjer er det gjort slik at brukeren har et bestemt antall forsøk på å skrive inn det som kreves av funksjonen for å komme seg videre. Hvis brukeren overstiger dette antallet forsøk vil programmet avbryte funksjonen, skrive ut en feilmelding til brukeren, og så gå tilbake til hovedmenyen for at brukeren kan fortsette å benytte seg av programmet.

Har benyttet SonarLint pluginen som en verifisering på at koden ikke inneholder unødvendige ting som kan redusere effektiviteten til programmet. SonarLint ble også brukt til å oppdage potensielle farer slik at det ikke oppstår problemer senere i prosessen hvor det er mer tidkrevende å fikse slike problemer.

5.4 Brukervennlighet

For å sørge for at brukeren enkelt skjønner hva som skjer og hva som kreves av han når han bruker programmet, er det viktig å gi brukeren gode og deskriptive tilbakemeldinger mens han bruker programmet. Hvis brukeren gjør noe feil vil det komme en ny tilbakemelding som forklarer hva som kreves for det steget i funksjonen brukeren har valgt. Det gis også tilbakemelding når det ikke går som planlagt ved å bruke feilmeldinger med rød skrift, for eksempel når brukeren prøver å velge et menyvalg som ikke eksisterer, eller prøver å fjerne et produkt som ikke finnes i lageret.

Der det er enkelt å sette seg fast vil brukeren bli sendt tilbake til hovedmenyen hvis han ikke greier å forsyne programmet med den nødvendige informasjonen innen et gitt antall forsøk. Brukeren kan da fortsette å bruke programmet etter at han setter seg fast og trenger ikke å avslutte og gjenåpne programmet.

5.5 Dokumentasjon

God dokumentasjon er viktig for at programmet skal kunne brukes effektivt og opprettholdes. Det er ikke nødvendigvis personen som utvikler programmet som skal vedlikeholde programmet, og da er det en stor fordel for personen som vedlikeholder programmet at det er godt dokumentert og at metoder og felt har gode og beskrivende navn.

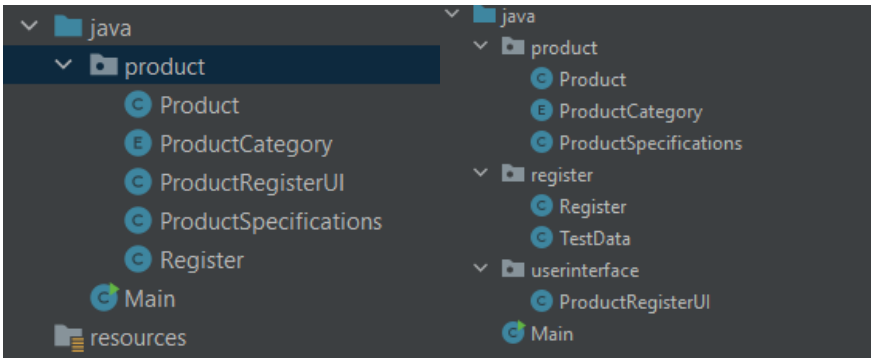
Alle metoder, klasser og konstruktører har derfor Javadoc som kort forklarer hva hensikten med metoden/klassen/konstruktøren er, og hva som blir returnert hvis dette er aktuelt. Metoder som er lange og vanskelig å lese har kommentarer inne i metoden for å forklare hva som skjer og gjør det enklere for personen som leser koden å skjønne hva som faktisk skjer i metoden.

Har brukt Google style gjennom CheckStyle pluginen for å sikre at koden er skrevet på riktig vis.

5.6 Refaktorering

5.6.1 Pakkestruktur

Når jeg jobbet med prosjektet gjennom semesteret jobbet jeg i små perioder og jobbet på enkelte metoder/funksjoner. Dette førte til at alle klassene mine endte opp i samme pakke, noe som ikke er særlig god praksis så jeg brukte refactor funksjonen innebygd i IntelliJ til å flytte Register og ProductRegisterUI klassene til egne pakker. Refactor funksjonen sørger for at koden fortsatt fungerer ved å legge til nødvendig kode på steder hvor metoder blir brukt på tvers av klasser og pakker.



Figur 6: Dårlig vs god pakkestruktur

5.6.2 Test data

Mens jeg utviklet programmet var det nødvendig å teste om metodene og programmet generelt funket som det skulle. Da var det tungvint å lage produkt, legge de til registeret for så å teste det jeg jobbet på. For å unngå å måtte manuelt lage test data selv, lagde jeg en funksjon som gjor det for meg hver gang jeg kjørte koden.

```
/**
 * Adds products into the register to use for testing.
 * TODO: Lag mer test data.
 */
public void addTestData() {
  addProduct(new Product( productNumber: "LF432893", description: "Brown floor laminate", price: 499, brand: "Dunno",
      new ProductSpecifications( weight: 3, length: 200, height: 4.5, color: "Brown"), stock: 5, ProductCategory.LAMINATEFLOOR));
  addProduct(new Product( productNumber: "BMDR382", description: "Yellow door", price: 7999, brand: "BM",
      new ProductSpecifications( weight: 25, length: 80, height: 240, color: "Yellow"), stock: 5, ProductCategory.DOOR));
  addProduct(new Product( productNumber: "BMPL921", description: "Plank", price: 123, brand: "BM",
      new ProductSpecifications( weight: 8.5, length: 300, height: 25, color: "Light Blue"), stock: 149, ProductCategory.LUMBER));
  addProduct(new Product( productNumber: "OBSWND328", description: "Window with black frame", price: 4000, brand: "OBS",
      new ProductSpecifications( weight: 14.5, length: 100, height: 100, color: "Black"), stock: 5, ProductCategory.WINDOW));
}
```

Figur 7: addTestData()

Jeg hadde manuelt lagt til fire produkt til registeret gjennom en metode for å ha noen produkt jeg kunne bruke mens jeg utviklet programmet, men siden det var ønskelig med mer test data enn det som var rimelig å manuelt lage i en metode endte jeg opp med å lage en metode som kunne automatisk generere test data. Hvis dette skulle gjøres i

Register klassen slik som addTestData() metoden, ville Register klassen ha for mye forskjellig funksjonalitet. For å oppnå høy cohesion, endte jeg opp med å lage en helt ny klasse «TestData» som kun skulle ha rollen av å generere test data til registeret.

I stedet for å gjøre det manuelt som over lagde jeg TestData klassen med tre lister som inneholdt descriptions, brands og colors. Listene blir brukt til å automatisk generere produkt med tilfeldige verdier valgt fra disse listene.

```
public class TestData {
  private static final String[] descriptions = {"A very nice product",
      "Cannot be used with any flooring", "Works great with other products of the same brand",
      "A poorly optimized product", "Has three possibilities for extensions",
      "Compatible with other brands"};
  private static final String[] brands = {"Byggmakker", "COOP OBS",
      "Byggmax", "Maxbo", "Jernia", "Jula"};
  private static final  String[] colors = {"Black", "Red", "Grey", "Blue",
      "Light grey", "Green", "Oak"};
  private static final Random random = new Random();
```

Figur 8: TestData klasse

Funksjonen generateTestData() tar som argumenter: registeret hvor den genererte dataen blir lagt til i, og mengden produkt som skal genereres. På denne måten kan en lage så mye test data en bare ønsker og vil være nyttig for videre utvikling og eventuell testing.

```
/**
 * Generates random Product objects and adds them to the specified register.
 *
 * @param register add random Product objects to this register
 * @param amountToGenerate amount of random Product objects to create
 */
public static void generateTestData(Register register, int amountToGenerate) {
  for (int i = 0; i < amountToGenerate; i++) {
    //Creates a new product with random values chosen from the lists:
    //descriptions, brands and colors.
    register.addProduct(new Product( productNumber: "" + (i + 1),
          descriptions[random.nextInt(descriptions.length)],
          random.nextInt( bound: 1000),
          brands[random.nextInt(brands.length)],
          new ProductSpecifications( weight: Math.random() * 100,
                length: Math.random() * 100,  height: Math.random() * 100,
               colors[random.nextInt(colors.length)]),
          random.nextInt( bound: 50), register.getCategoryList()
          .get(random.nextInt(register.getCategoryList().size()))));
  }
}
```

Figur 9: generateTestData()

5.6.3 Kode duplisering

Når jeg lagde metoden som tok bruker input for å legge til et produkt i registeret brukte jeg en do-while loop for å sikre at brukeren skrev inn noe for produkt nummer, merke og farge. Denne do-while loopen funket bra, men gjorde at jeg hadde veldig lik kode flere steder i samme metoden. Bildet under viser et utklipp av metoden addProductFromUser() i ProductRegisterUI klassen hvor de røde linjene illustrerer hvordan koden var før, og de grønne linjene, hvordan koden ble etter refaktoreringen.

```
-   String tempDescription;
-   do {
-     System.out.println("Enter a description for the product: ");
-     tempDescription = sc.nextLine();
-   } while (tempDescription.isEmpty());
+   String tempDescription = stringChecker("Enter a description for the product: ");

    int tempPrice;
    System.out.println("Enter a price for the product: ");
    tempPrice = intCheckerPositive();
    clearScanner();

-   String tempBrand;
-   do {
-     System.out.println("Enter the brand of the product: ");
-     tempBrand = sc.nextLine();
-   } while (tempBrand.isEmpty());
+   String tempBrand = stringChecker("Enter the brand of the product: ");
```

Figur 10: Refaktorering fra Git

For å unngå den dårlige praksisen å ha duplisert kode, lagde jeg en funksjon som gjorde akkurat det samme som koden i den originale funksjonen ved å ta inn meldingen som skulle bli skrevet ut til brukeren som parameter for metoden og benytter en do-while loop for å sikre at brukeren skriver inn noe før han kan gå videre.

```java
/**
 * Checks the users' input to make sure it is not empty.
 *
 * @param message to print to the user
 * @return string input by the user
 */
private String stringChecker(String message) {
  String stringToReturn;
  do {
    System.out.println(message);
    stringToReturn = sc.nextLine();
  } while (stringToReturn.isEmpty());
  return stringToReturn;
}
```

Figur 11: stringChecker()

5.7 DRØFTING

5.7.1 Kode opp mot teori

For å oppnå ansvars-drevet design har hver klasse i programmet mitt en spesifikk rolle: Product klassen gjør alt som har med et spesifikt produkt å gjøre, Register klassen gjør alt som har med selve lageret å gjøre, TestData generer kun test data for et register, ProductRegisterUI klassen gjør alt som har med brukergrensesnittet og brukeren å gjøre. Den fornevnte klassen innser jeg er litt lang og burde kanskje blitt splittet i to deler: selve brukergrensesnittet og bruker input.

For å oppnå høy «cohesion», har alle metodene et spesifikt mål den skal oppnå. Når jeg oppdager at metoden blir lang vurderer jeg om metoden kan splittes opp i flere mindre, mer spesialiserte metoder som heller kan jobbe sammen for å oppnå samme funksjonalitet. De mindre metodene vil mest sannsynlig kunne bli gjenbrukt når det eventuelt vil bli lagt til mer funksjonalitet i applikasjonen gjennom dens levetid. Samtidig

har jeg prøvd å oppnå å ha «loose coupling» ved at alle klassene skal kunne fungere hver for seg. Når en endring i en klasse gjør at man må endre flere ting i andre forskjellige klasser er det et tegn på at klassene henger for tett sammen og strider mot målet om å ha «loose coupling».

Alle felter er satt til private sammen med alle metodene som ikke blir brukt på tvers av klasser. Dette er for å følge design prinsippet om «encapsulation». For å kunne nå og endre disse private feltene må en gå gjennom public metoder for å hente/endre feltene.

5.7.2 Robusthet

Når jeg skulle sjekke om programmet var sikkert og at det fungerte som det skulle, gikk jeg manuelt gjennom hver funksjon brukeren kunne nå og forsøkte å krasje programmet ved å gi dårlig inputs samtidig som jeg sjekket at programmet gjorde det som var forventet når riktig input ble gitt. Dette er tidkrevende og ikke veldig effektivt. Allerede for et lite program som dette blir det fort tungvint å manuelt sørge for at programmet funker som det skal når noe endres, men for et større program med flere funksjoner som må sjekkes er dette helt urealistisk å gjennomføre. Det jeg burde gjort, og ville gjort for å forbedre programmet er å implementere automatiske tester som kan kjøres for å verifisere at programmet fungerer som forventet når både riktig og feil input blir gitt, dette er et konsept kalt «positive and negative testing».

For at brukeren ikke skal kunne klare å sette seg fast hadde jeg noen plasser implementert en fail-safe hvor programmet ville gå tilbake til hovedmenyen hvis brukeren ikke klarte å gi programmet den nødvendige informasjonen innen et gitt antall forsøk. Denne typen fail-safe kunne vært implementert flere steder i koden, for eksempel når brukeren legger til et produkt i lageret.

5.7.3 Brukervennlighet

For å øke brukervennligheten av programmet skulle jeg gjerne implementert en måte for brukeren å gå tilbake et steg i menyen. Hvis brukeren velger å legge til et produkt til lageret, må han gå gjennom hele prosessen og kan ikke velge å avbryte den utenom å avslutte programmet og starte det opp igjen. Dette er ekstremt tungvint for brukeren hvis han med uhell velger feil menyvalg, noe som ofte kan skje.

Funksjonene for å fjerne et produkt og endre på pris/lagerbeholdning/beskrivelse tar et produkt nummer som argument. Dette produkt nummeret indikerer hvilken vare som skal slettes/endres og brukeren må skrive inn dette produkt nummeret selv fra hukommelse. En mulig løsning for å øke brukervennligheten til programmet er å benytte seg av søke funksjonene i programmet, la brukeren velge et produkt og deretter spør brukeren om han vil endre eller slette produktet han valgte. For å gjennomføre denne endringen vil det kreve en god del arbeid, da man må lage en metode som genererer menyvalg ut i fra listen med produkter som blir returnert fra søke metodene.

6 KONKLUSJON - ERFARING

Applikasjonen som har blitt utviklet fungerer i henhold med kravspesifikasjonene, med noen mulige forbedringer når det gjelder brukervennlighet og automatisk testing. Den har noe ekstra funksjonalitet utenfor kravspesifikasjonene, ved at brukeren kan søke etter deler av produkt nummer/beskrivelse og få en liste av treff. Koden følger «best practice» teknikker som høy «cohesion» og løs «coupling», ansvars-drevet design og unngått kode duplisering, som beskrevet i bakgrunn delen av rapporten. For å sikre at applikasjonen er robust, blir all input fra brukeren sjekket og verifisert. I tillegg går det ikke ann å lagre negative verdier i felter som price, stock, height, weight og width i Product og ProductSpecifications klassene. Dette sikres ved å sjekke for slike verdier i setter metodene til de respektive klassene.

For videre utvikling vil det være nødvendig å lage automatiske tester som sørger for at metoder og funksjoner fungerer slik de skal når endringer blir gjort. For å øke brukervennligheten til programmet vil det også være nyttig å benytte seg av et brukergrensesnitt som ikke er tekstbasert. Vil potensielt være lurt å splitte ProductRegisterUI klassen i to ulike klasser som håndterer inputs fra brukeren og brukergrensesnittet hver for seg, da ProductRegisterUI klassen begynner å bli litt stor.

7 REFERANSER

[1] Barnes, B.J. og Kölling, M. (2017) Objects First with Java – A Practical Introduction using BlueJ. 6. Utg. Edinburg: Pearson Education.