

WESTERDALS OSLO ACT

PG3300 Software Design EKSAMEN

Frist 26.11.17, C#, Leveres som PDF

Alexander T Gardehall, Alex Ahnon Hansen og Margrethe Thorsen

Relevante linker

[Github](#)

[Arbeidskrav dokumentasjon](#)

[LucidChart \(Hovedside\)](#)

[LucidChart \(Extra pga. mangel på plass i hovedsiden\)](#)

Dokumentasjon

Eksamensoppgaven ble gitt ut i forelesning 11, den 01.11.17.

Prosjektet startet med at dokumentasjon, UML-verktøy og GitHub ble satt opp. Vi tenkte at siden vi fikk god tilbakemelding på arbeidskravet, burde vi bruke samme mal i eksamenen. Vi begynte først med UML-delen av innleveringen, ettersom dette kunne gi oss en bedre forståelse og et større bilde over hvordan oppgaven kunne bli seende ut og senere utført. Det ble først laget et use case diagram som viste hvordan sluttresultatet ville fungere i praksis, og videre ble det dannet klasser som vi tilslutt implementerte i et klassediagram. På denne måten fikk vi bedre oversikt over hvilke klasser/metoder vi ville implementere videre i oppgaven. Det var litt diskusjon over bruk av "include" eller "extends" i use case, men vi kom frem til at "include" gir mer mening når man tar med definisjonen av uttrykket.

I innleveringen fikk vi mulighet til å velge hvilket verktøy (IDE) vi skulle bruke for å skrive oppgaven. Løsningen vår ble skrevet i Visual Studio som har vært det verktøyet vi har brukt og blitt kjent med dette semesteret i software design. Vi implementerte (så langt det var mulig) kode-designet vi kom fram med til i UML, videre i visual.

Idéen vi kom frem til, som ble utgangspunktet for UML klassediagrammet er at bruker starter Client som starter en Controller. Controller delegerer oppgaver til Factory, som oppretter objektene, og videre til en tråd-manager som starter "markedsprosessen". Factory bruker markedsroller som enum for å vite hva slags objekter som skal lages, og får navn fra en navnegenerator som lager navn til varer, kunder og butikker basert på markedsrollen, og varetypen for varer. Varene sendes til butikkene via en "StockItems"-metode og legges ut til salg hvor kundene kan se og kjøpe dem. Resultatet av et kjøp blir printet ut slik vi kan validere at en vare ikke selges til flere personer.

Etter vi nå hadde fått en forståelse av hva vi ville frem til i oppgaven, begynte kodingen. Det ble lagt mange tanker bak hovedtemaene vi skulle ha med og ta stilling til, og dette ble mye diskutert under prosjektet - noe som vi presiserer nærmere senere i oppgaven. Ettersom

innleveringen dreide seg om en bazaar, endte vi opp med å ha klasser som; Item, Customer, Shop og ShopInventory(sistnevnte ble fjernet pga. unødvendig informasjon jf. oppgavebeskrivelse), samt klasser som kunne gjøre arbeid "bak kulissene"; Fabricator, NameGenerator, ThreadManager, MarkedManager, Controller og Client. Varene blir delt opp i flere kategorier, ItemType(enum) som blant annet; mat drikke, leker, klær og møbler, i tillegg til at markedet blir delt opp i roller, MarkedRole(enum), hvor vi har kategoriene; butikk, vare, kunde.

Når vi levere koden bruker vi 2 butikker og 3 kunder. Vi har testet med flere av begge, opptil 4 butikker og 8 kunder, og med færre kunder enn butikker. Alt virket, men når man har flere butikker enn kunder har varene en tendens til å ligge ute på markedet en stund. Bruk av stack fremfor en queue gjør dette litt vanskeligere å spore, men kunder har en tendens til å gå for de ferskeste varene først i virkeligheten og.

UML diagrammer

Benytt UML og sett opp use case diagram for deres bazaar. Lag deretter et class diagram. Disse skal implementeres i pdf dokumentet som en del av besvarelsen.

Vi valgte å bruke Lucidchart i eksamensløsningen vår og, ettersom vi følte at vi fikk godt utbytte av å bruke dette verktøyet i arbeidskravet vårt. Det er et verktøy vi har blitt godt kjent med, og som har gode implementasjoner til UML.

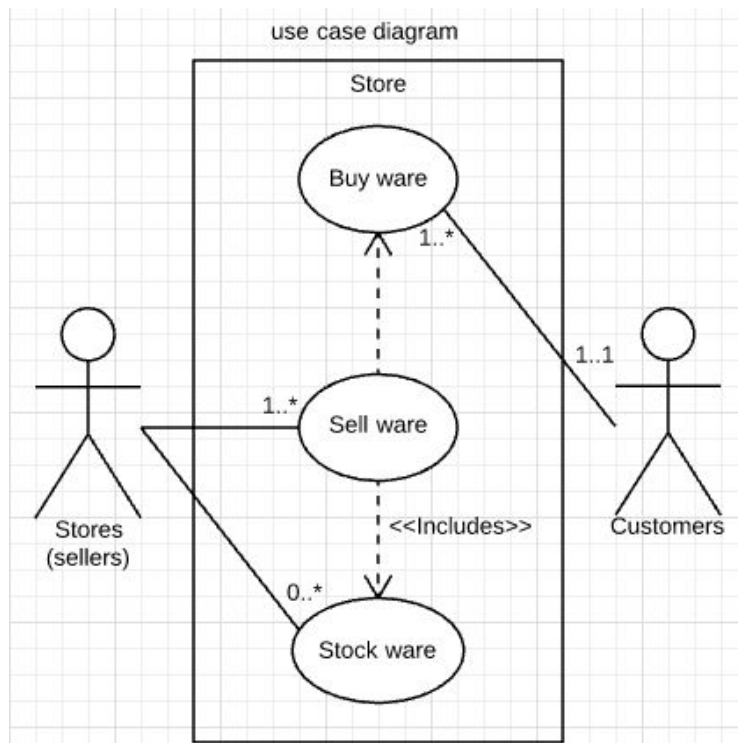
UML Use Case Diagram

En use case beskriver interaksjon mellom en aktør og systemet

I use case diagrammet vårt er butikk og kunder eksterne aktører.

'Sell ware' inkluderer, <<include>>, 'Stock ware', ettersom den er avhengig av denne oppgaven (det er nødt å være stocks/varer på lageret, for at det kan være mulig å kunne selge selve varen).

Foreslått løsning til use case:



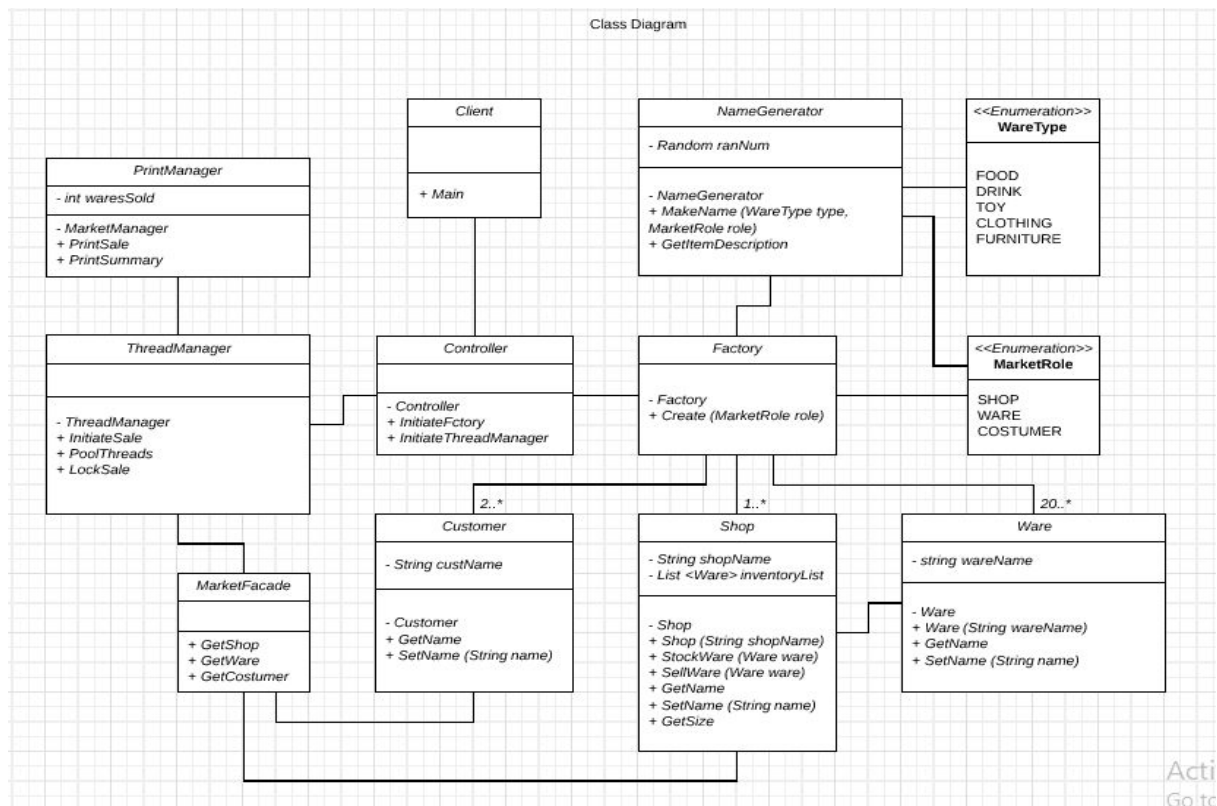
UML Class Diagram

Likt EAR modellering fra databaser, men her opererer vi med assosiasjoner for klasser i klassediagrammer

Klassediagrammet var nok det vi brukte mest tid på av UML-delen av innleveringen, grunnet til at diagrammet var en kravspesifikasjon over hvilke objekter vi ville trenge for å fullføre oppgaven. Underveis ble det flere endringer i diagrammet som følger av diskusjon over hvordan diagrammet og løsningen skulle bli seende ut, men dette følte vi var en fordel, ettersom vi da fikk diskutert oss frem til et svar vi alle ble fornøyd med.

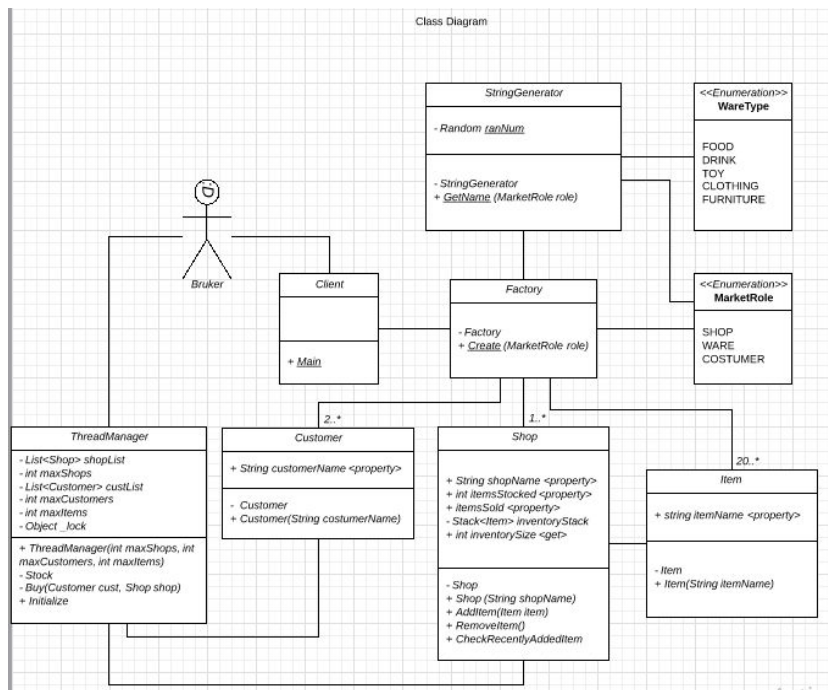
Vi ble også enige om at vi skulle lage to klassediagram, ett før koding og ett etter. Dette gjorde vi av den grunn at vi da kunne ha et diagram over hvordan vi tenkte løsningen skulle bli, vs. hvordan løsningen faktisk ble. Ved å gjøre dette, kunne vi se hvilke klasser/metoder vi trengte eller ikke trengte, og dermed få større oppfatning av kravspesifikasjon.

Foreslått løsning til løsning før ferdig prosjekt:



I begynnelsen av innleveringen, da vi jobbet med klassesdiagram, måtte vi ha en idé om hvordan programmet skulle fungere. En av de første beskrivelsene/utgangspunktene til programmet vårt ble som følger:

start -> butikk får vare -> butikk legger vare ut for salg -> kunder prøver å kjøpe vare -> thread manager hindrer alle unntatt en kunde å kjøpe varen -> kunden som var først kjøper varen -> gjenta 20 - 50 ganger for hver butikk

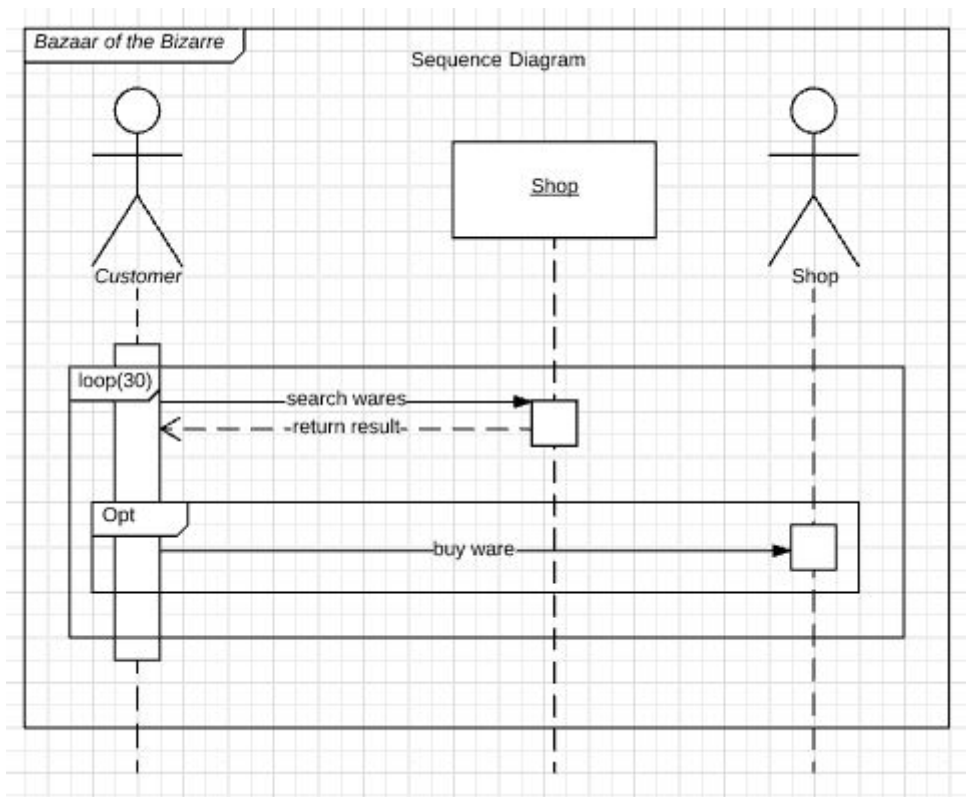


Dette diagrammet er nærmere den faktiske løsningen. Merk at Controller, MarketFacade, og PrintManager ikke er tilstede.

UML Sekvensdiagram

Sekvensdiagrammer er noe vi benytter når vi skal vise hvordan elementene i en løsning skal kommunisere med hverandre. Altså, kommunikasjon over tid; per definisjon ovenfra og ned.

Forslag til sekvensdiagram:



Vi la ikke like mye vekt på sekvensdiagrammet, som på de andre diagrammene, ettersom det ikke var en kriterie for innleveringen. Men, vi tenkte at det kunne være kjekt å lage et enkelt diagram, for å enkelt vise hvordan objektene i programmet kommuniserer med hverandre. Og ikke minst for å kunne bruke litt av kunnskapen vi har lært i emne, videre i innleveringen.

GRASP

Creator

Sier noe om hvem som bør opprette objekter

Factory blir brukt til å danne objekter av klassene som kreves av programmet.

Controller

"Laget under GUI", har ansvar for mottak og håndtering av system events(feks. kall fra GUI)

Controller startes av Client, og forteller Factory hvor mange Customers, Shops og Items

(Shops * 20) den skal lage, og ThreadManager kontrollerer tråder som selger og kjøper varer.

High Cohesion

Objekter delegerer heller objekter videre enn prøver å gjøre alt selv

Vi har high cohesion ved å delegere oppgavene til flere klasser, som f.eks. Factory, NameGenerator og ThreadManager klassene.

Information Expert

Sier noe om hvor ansvaret for å kjenne til et gitt objekt bør ligge

I forhold til information expert så er det Shop som holder styr på Item, og ThreadManager som holder styr på Customer/Shop/Item forholdet.

Low Coupling

Gi ansvar for objekter på en slik måte at mengden (unødvendig) kobling holdes lavest mulig

Vi prøver å skape en low coupling ved å passe på at det finnes minst mulige koblinger mellom klassene. Kun de objektene som krever å snakke med hverandre har en kobling, som f.eks. Factory til alle objektene som skal dannes.

Design Patterns

Factory

Bruker ("klient") kaller på en factory som lager objekter for seg

Factory blir brukt til å skape objekter av Item, Shop og Customer.

Decorator

Kan simulere arv runtime.

Decorator kan bli brukt til å legge til nye egenskaper til Item, Shop og Customer i løpet av runtime, men siden programmet ikke er så avansert, så mente vi at det ikke var nødvendig.

Singleton

Sikre én instans av et objekt, der det bare er ønskelig med én instans gjennom hele applikasjonen

I programmet vårt, er Factory og Controller singletons

Model View X

Ev oversikt over hvordan programmet er delt inn i lag som kommuniserer med, eller er gjemt for bruker.

Vi benytter oss av Model View Presenter i dette prosjektet ved å bruke en designert utskrifts-klasse, PrintManager, som tar imot all informasjon rundt kjøp og salg av varer som blir gjort i klassen ThreadManager.

PrintManager blir derfor Presenter, ThreadManager blir Model og GUI blir View.

Composite

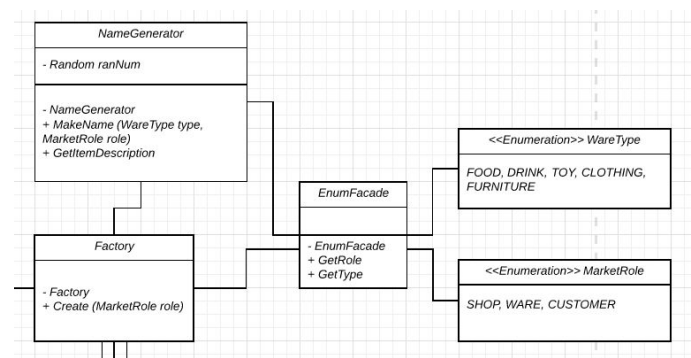
Lar programmet behandle flere instanser av et objekt som et enkelt objekt.

Composite er ikke relevant for programmet ettersom ingen objekter grupperes eller samles på måter som lar oss behandle gruppen av objekter som enkeltobjekter.

Façade

Et interface som samler kode og gjør den lettere å jobbe med.

Mulig bruk av Facade kunne vært å lage en facade-klasse EnumFacade som brukes til å få en enum av ItemType og/eller MarketRole. Dette ville også bidratt med high Cohesion, low Coupling hvis prosjektet skulle gro, men side det for øyeblikket bidrar til flere koblinger lar vi være å bruke det for øyeblikket.



Flyweight

Deler større mengder informasjon mellom klasser.

Flyweight er et design pattern vi ikke bruker i programmet. Grunnen til dette er at vi ikke har mye felles for objektene, og det ikke gir mening å bruke dem. Vi kunne brukt Flyweight sammen med en butikklager-klasse (ShopInventory), for å holde styr på varer og informasjon om butikkenes lagertilstand og vareinnhold, eller om varene som ble opprettet var av samme type.

Multithreading

Ettersom oppgaven spesifiserer at det skal være flere kunder og flere butikker gir det mening å bruke multithreading. Et problem vi kan møte med bruk av tråder er at tråder krasjer og flere enn en kunde kjøper samme vare(race conditions -> locks), noe som oppgaveteksten også nevner.

Foreslått logikk for tråder:

1. Butikk legger vare ut for salg
2. ThreadManager lager tråd for hver vare lagt ut (en per butikk)
3. Et threadpool åpnes så alle kunder kan se vare-trådene
4. Kundene sloss om varen, varen låses og førstemann får den

5. Gjenta

1. Butikker pusher vare til en stack
2. Kunde låser vare
3. Print "(Kunde) kjøpte (vare)"
4. Pop vare

Unit Testing

Unit testing er en metode for å sikre små enheter kildekode individuelt er robuste nok til bruk

Eget prosjekt i solution som tar for seg tester for å teste funksjonene til programmet.

-BizarreBasaarTest(s)

TestAddItem

TestAddShop

Test AddCustomer

TestBuyRemoveItem

TestNameGeneratorNonDuplicatesShop

TestNameGeneratorNonDuplicatesCustomer

TestNullInputFactoryMarketRole

TestNullInputStringGeneratorMarketRole

TestNullInputShopStock

TestNullInputBuyCustShop

Testing av gettere og settere, og lignende er unødvendig siden det tester ikke kode, men heller Visual Studios/C# funksjoner. Desverre oppfyller ikke tråd-klassene Humble Object Test Patternet, så det ikke så mye vi får testet.

Test-klasse er ikke implementert i UML klassediagram på grunn av usikkerhet rundt plassering og koblinger. Skulle den blitt implementert ville den mest sannsynlig være koblet til alle klassene den tester og brukeren, som gjør diagrammet veldig vanskelig å lese. Eventuelt kunne det vært laget et eget UML diagram for koblinger mellom testene, bruker, og klassene.

Parprogrammering

Parprogrammering er en fleksibel programvareutviklingsteknikk hvor to programmerere jobber sammen på en arbeidsstasjon.

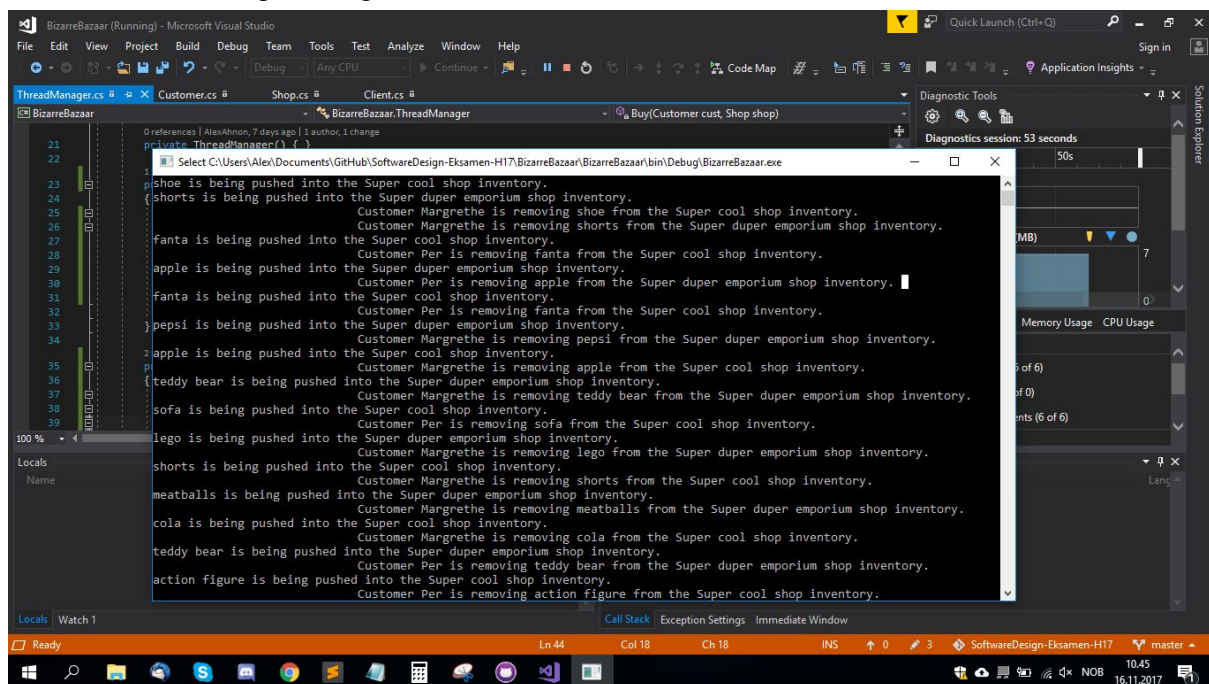
Parprogrammering ble for det meste brukt til å løse problemer rundt bruk av tråder. Dette ble gjort ved å sette en datamaskin med koden i midten av gruppen så alle kunne se den. Programmet ble så kjørt og vi reflekterte over hva grunnene kunne være for at trådene ikke kommuniserte som tiltenkt, hvis resten av programmet virket som det skulle. Vi diskuterte hvor i koden og hvordan det kunne feile, og testet deretter noen endringen for å rette opp problemene. Dette kan i teorien gjentas til problemet er løst.

Denne fremgangsmåten ga flere perspektiver på hva som kunne være problemet, og en diskusjon om hva problemet var og hvorfor det i noen tilfeller er raskere enn å skrive koden og teste det, i tillegg til at når koden blir skrevet er det større sjanse for at den er korrekt og inneholder færre feil. Alt i alt så alle på gruppen verdien i å diskutere kode med en eller flere andre personer, og føler dette er noe vi kan få godt utbytte av å gjøre i senere prosjekter.

Resultat

Etter flere uker med Software design kom vi i mål. Vi har fulgt hovedtemaene godt, og føler at resultatet både fungerer og samsvarer godt med utskrifts-eksempelet fra oppgaven.

Screenshot av ferdig løsning:



Suksess!

Kort Refleksjonsnotat

Nå som Software design er i mål, kan vi endelig si oss ferdig og se tilbake på arbeidsprosessen vår. Under prosjektet har vi jobbet godt sammen på gruppen, der alle har vært gode til å kommunisere, slik at det verken ble uenigheter eller problemer innad i gruppen. Vi har under prosjektet vært flinke til å møte opp når vi skulle, og vi tok oss tid utenfor øvingstimene for å jobbe med innleveringen. Dersom det ble uenighet, løste vi dette med å diskutere oss fram til en løsning alle ble enige om. Emneplanen(oppgaven) har blitt brukt flittig - den ga oss god oversikt over oppgavene vi skulle utføre/ha med, og førte oss i riktig retning. Det har vært jevn jobbing fram til målet, der det har vært nok oppgaver å gjøre. Vi er generelt fornøyde med hvordan innleveringen har blitt løst, og føler at resultatet gjenspeiler tiden og tankene vi har lagt ned i den.

Kildehenvisning

- Sandnes, T. (2017). Forelesninger fra emnet "Software Design", mye om Design Patterns og GRASP som er hentet ut [PDF].
Hentet fra It's Learning.
- Klassifikasjon av UML-diagrammer → <https://www.uml-diagrams.org/>
- C# dokumentasjon → <https://docs.microsoft.com/en-us/dotnet/csharp/>
- Tips og hjelp til koding → <https://stackoverflow.com/>