Bazaar of the Bizarre - Dokumentasjon

Ralf Fajardo

Philippe Rasay

Program og framework

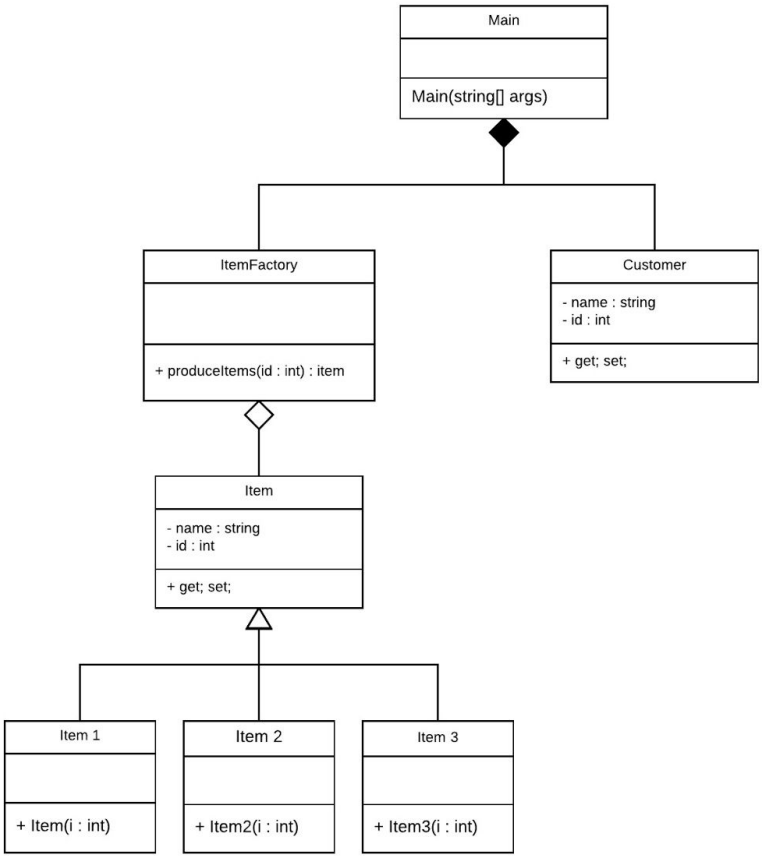
- **Visual Studio 2017**
- **.NET Core 2.0**
- **Lucidcharts (UML diagrammer)**

Vi har fått i oppgave å designe og lage en løsning der hvor et marked med ulike boder selger ulike varer, hvor kundene står i kø for å kjøpe varene. Bodene putter sine varer ut for salg etterhvert som de ankommer lageret, og de holder åpent til de har solgt kvoten sin. Kundene i flertall prøver å få tak i varer i et høyt tempo.

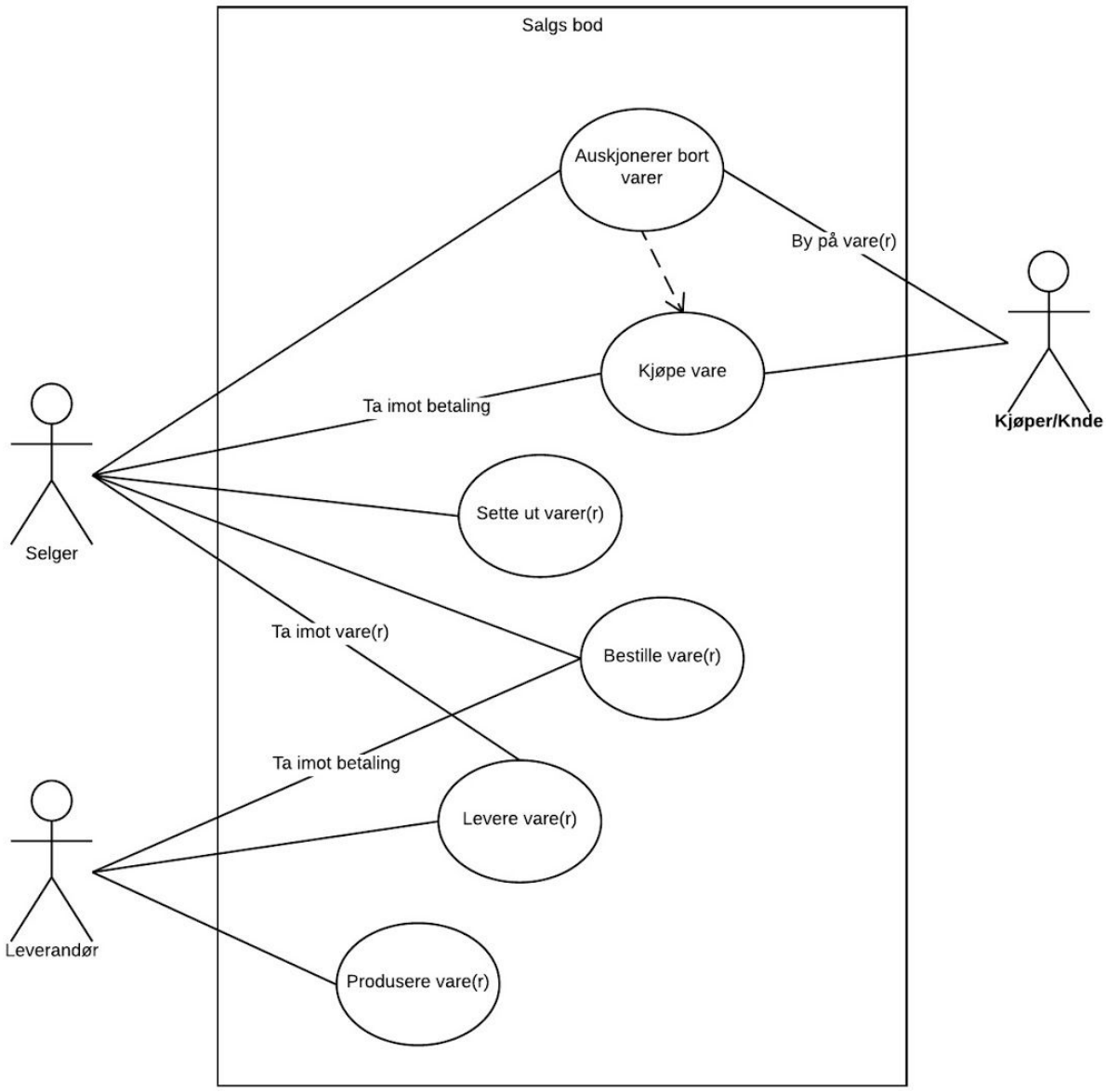
Her er noen kriterier for å lage en god løsning:

- Flere kunder kan ikke kjøpe samme vare
- Kunder kan ikke kjøpe varene før de legges ut for salg

Det vi startet med var å diskutere og brainstorme om hvordan vi skulle løse oppgaven. Vi begynte å tegne ulike klasser og metoder vi tenkte vi skulle ha med, og lagde et diagram av hvordan vi tenkte at løsningen vår skulle se ut.



Deretter lagde vi et use case diagram for en bazaar bod, for å prøve å få litt mer oversikt over hva som faktisk skjer i programmet. Og få inspirasjon for å få en strukturert og ryddig kode.



Da vi hadde fått et oversikt bedre oversikt over strukturen begynte vi å implementere koden.

Parprogrammering

Siden vi kun er to som er gruppen har vi alltid sittet ved siden av hverandre og jobbet sammen. Helt i starten har vi brukt en god del på parprogrammering, hvor vi begge satt

sammen og jobbet på samme kode. Vi jobbet slik at et kodet, mens den andre satt ved siden av og kom med innspill, så byttet vi på rollene. Dette føler vi har hjulpet å forbedre koden vår, på grunn av at det skjedde ofte at den som satt og kodet satt fast, og den som satt ved siden av så feilen ganske fort. Derfor har vi valgt å bruke parprogrammering til store deler av oppgaven, istedenfor at vi begge jobbet med hver vår oppgave. Fordelen vi så med parprogrammering at vi føler vi skrev en bedre kode, fordi vi var hele tiden to som kom med innspill til en og samme kode. Og ofte så man en feil eller en forbedring som den andre ikke så. På den andre siden tok implmenteringen av koden lengre tid enn forventet, siden begge jobbet med samme oppgave istedenfor å fordele oppgavene. Etter at vi fikk implementert det meste av strukturen, begynte vi å fordele mindre oppgaver hver for oss. Hvor vi ikke benyttet parprogrammering, men heller satt på hver vår pc og jobbet ved siden av hverandre.

GRASP

Etter vår brainstorming har vi valgt å bruke prinsippene i GRASP.
Her er en beskrivelse og litt eksempler på hvordan vi har brukt de prinsippene i GRASP vi har benyttet oss av.

Creator/Information expert

Til vår creator har vi valgt å lage klassen program. Denne klassen har ansvaret for å opprette instanser av klassene Bazaar og Customers. Program klassen er også den som delegerer tilgjengelighet til et gitt objekt.

Siden Program klassen har informasjonen til å opprette instanser av de to andre klassene, gjør det også program klassen til information expert.

Low coupling

Vi har valgt å ha veldig lav kobling mellom klassene. Der Item har tre subklasser sword, runestone, ring. Våre klasser avhenger ikke av mange andre klasser for å fungere, det er som regel kun en kobling mellom klassene vi har lagd. Noe som gjør det lettere å gjenbruke koden vår, og/eller bygge på den.

High cohesion

Her har vi laget klasser i forhold til hva de skal gjøre. Item klassen henter navn og id i subklassene(sword, runestone, ring). De fleste klassene vi har lagd har få oppgaver de gjør. Istedenfor at klassene skal ha mange små oppgaver som ikke henger sammen, har vi delt det opp i flere klasser som fokuserer på en eller to oppgaver.

Polyformisme

En item kan ha flere former basert på hvilke id som blir brukt. Det kan enten være sword, runestone eller ring.

For å summere hvordan GRASP fungerte i vår løsning fant vi ut at ved å bruke GRASP sin design pattern klarte vi å abstrahere opprettelsen av objekter og klarte å få en fin design på løsningen vår.

Design patterns

Factory

Vår factory klasse er itemFactory, istedenfor at Bazaar klassen vår skal lage item objekter, så kaller den på itemFactory som opretter item objekter for den. Siden det er så mange item objekter som skal opprettes bestemte vi oss for å lage en factory klasse for å abstrahere opprettelsen av objekter.

Facade

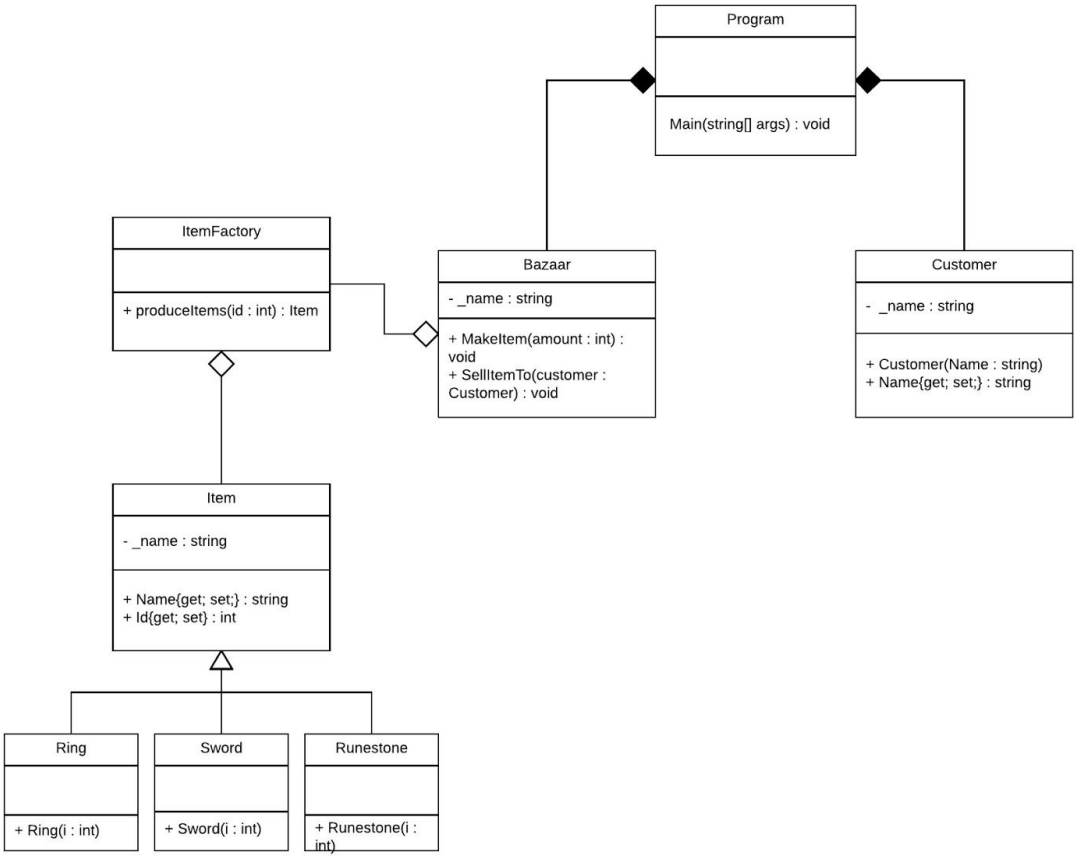
Vi har brukt Facade i for eksempel klassen Program, har vi “gjemt” mye av logikken i de andre klassene. I program oppretter vi bare klassobjekter og kaller på de ulike metodene derfra. Noe som gjør at det er litt lettere å forstå hva som skjer i løsningen.

Multithreading

Vi valgte å implementere multithreading i løsningen vår for å få flere kunder til å kunne delta i budrundene. Men vi kom over et problem der flere kunder kunne få tak i samme items fra samme bazaar. For å forhindre dette benyttet vi oss av **lock funksjonen** for å sikre at en kunde ikke kan kjøpe samme item som en kjøper allerede har kjøpt. Dette forhindret at bazaarene ikke solgte samme item til flere kunder samtidig.

Code factoring

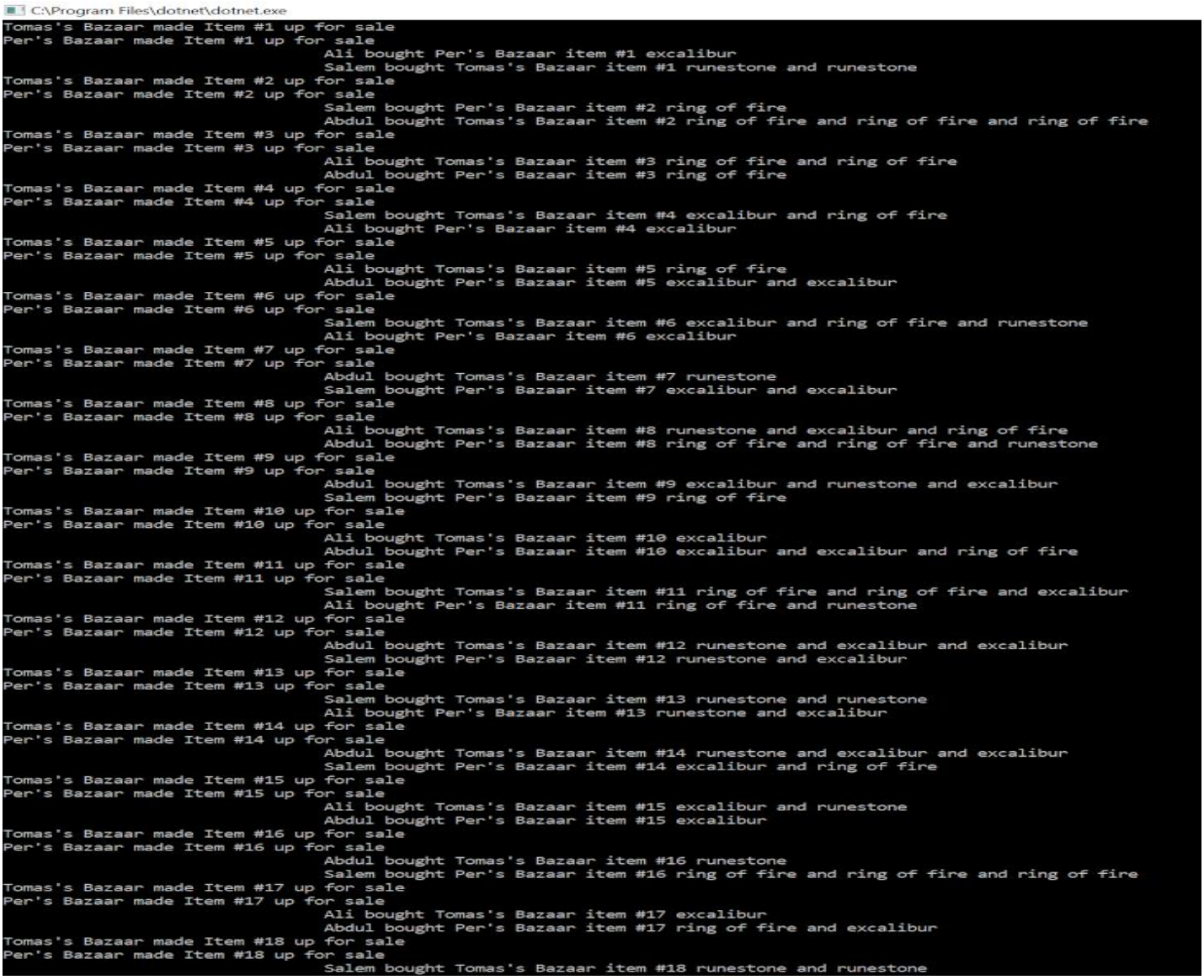
Etter at vi begynte å implementere koden fant vi ut at vi måtte refaktorere litt. Etterhvert som vi kodet fant vi på bedre navn til metoder og variabler. Vi så også at noen metoder passet bedre i egen klasse, så det er lettere å lese koden. Vi startet først med at Program klassen er den klasse som skulle kalle på ItemFactory for å oprette objekter. Vi bestemte oss for å lage en egen klasse (Bazaar) for dette, som både kaller på itemFactory og selger itemsene til kunden, istedenfor at alt det her skulle skje i Program klassen.



Her er class diagrammet vårt etter at vi refaktorerte koden

Unit testing ble ikke så mye fokusert på, men vi har tatt med litt testing på løsningen vår.

Her kan du se et utkast av resultatet fra programmet.



Her ser vi Per og Tomas's Bazaar og items de selger. Som nevnt tidligere så selger bazaarene tre items som ring of fire, excalibur og runestone.
Dere kan også se kundene som kjøper varer og at varene blir kjøpt etter at varen har blitt lagt ut for salg og ikke tidligere.

Vi har valgt en **queue** for å holde på varene som kundene har i kurven. Ettersom en kunde kan kjøpe flere varer samtidig har vi laget flere **queue** ettersom hvor mange varer kunden "ønsker" å kjøpe. Her har vi valgt at en **queue** kan ha en, to eller tre varer samtidig. (Vet ikke helt om dette er den riktige måten å gjøre det på, men vi føler at vi får samme output som i oppgave beskrivelsen.)
Som dere kan se i bildet kan Ali, Abdul og Salem kjøpe varer fra bazarene uten å måtte krangle om hvem som kjøpte varen først. Noe som gjør det til en bra bazaar! Her er det førstemann til mølla!

Vi håper at løsningen vår faller i smak og at dere ser våre design patterns i koden. Implementeringen av GRASP og Design patterns har gjort til at koden vår ser mer ryddig ut og gjør det lettere for andre å kunne bygge på det.(Mener vi i hvert fall)

Kilder

Slides fra forelesninger i PG3300
https://www.youtube.com/watch?v=9Y2mZger8kE&t=310s
https://www.youtube.com/watch?v=ovzDLPfK_T0
https://sourcemaking.com/design_patterns/abstract_factory
https://sourcemaking.com/design_patterns/factory_method
https://sourcemaking.com/design_patterns/facade
https://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/rao.pdf
http://www.dotnetfunda.com/articles/show/1492/software-patterns-grasp-part-iii-creator
http://www.dotnetfunda.com/articles/show/1518/software-patterns-grasp-part-v-low-coupling
http://www.dotnetfunda.com/articles/show/1522/software-patterns-grasp-part-vi-high-cohesion
http://www.dotnetfunda.com/articles/show/1548/software-patterns-grasp-part-vii-polymorphism
https://www.tutorialspoint.com/csharp/csharp_multithreading.htm

https://www.codeproject.com/Articles/1083/Multithreaded-Programming-Using-C

Vedlegg

Vedlegg 1: Kildekode

Vedlegg 2: Klassediagram før implementering av kode

Vedlegg 3: Klassediagram etter implementasjon av kode

Vedlegg 4: Use case

Vedlegg 5: Use case narratives