Stella Jovanovic, Nima Khademi, Mashal Hussain

UML og refactoring av SnakeMess

Oppgave 1

Introduksjon

Vi har fått en oppgave som går ut på å refaktorere kode til et klassisk snake spill, hvor vi skal benytte parprogrammering. Vi prøvde å løse oppgaven med tanken på bedre arkitektur og kode, uten å over komplisere prosessen og uten endre på spillets funksjonaliteten.

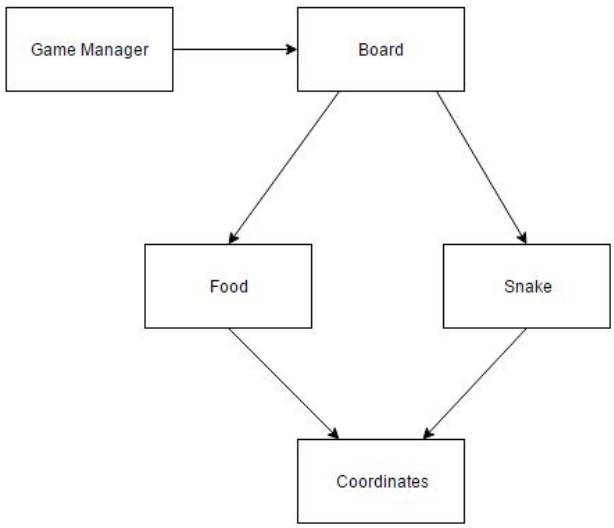
Arbeidsprosess

Vi startet med å se igjennom koden slik den var i utgangspunktet for å prøve å forstå den helt. Ingen i gruppen var kjent med spillprogrammering så dette ble en ganske stor utfordring, vi skjønte fort at det er en logikk som kanskje kan være naturlig for spillprogrammere som ikke er naturlig for oss. Derfor valgte vi å bruke mye tid på å sette oss inn i koden, for å få dannet et bilde av hvordan koden fungerer, før vi begynte med refaktoreringen.

Vi startet med å forandre små ting i koden slik at vi kunne se hva som utgjør forskjell i selve spillet etter koden kjøres og hva forskjellen ble.
Deretter prøvde vi å skille koden inn i det som vi mente var de viktigste objektene og funksjonaliteter, og prøvde å danne oss et bilde av hvordan alle disse komponenter ville fungere i forhold til hverandre.
Først valgte vi å starte veldig enkelt, ved å bytte ut variabelnavn med mer beskrivende varianter. Så prøvde vi å dele opp den originale koden i blokker med metoder som virket logisk og senere begynte vi med å opprette de ønskede klassene og å flytte metoder til klasser hvor det var mest fornuftig at de skulle være. Etter flere forsøk resulterte dette i at koden ikke kunne kjøres, så kom vi til konklusjon denne måten fungerte ikke så bra for oss.

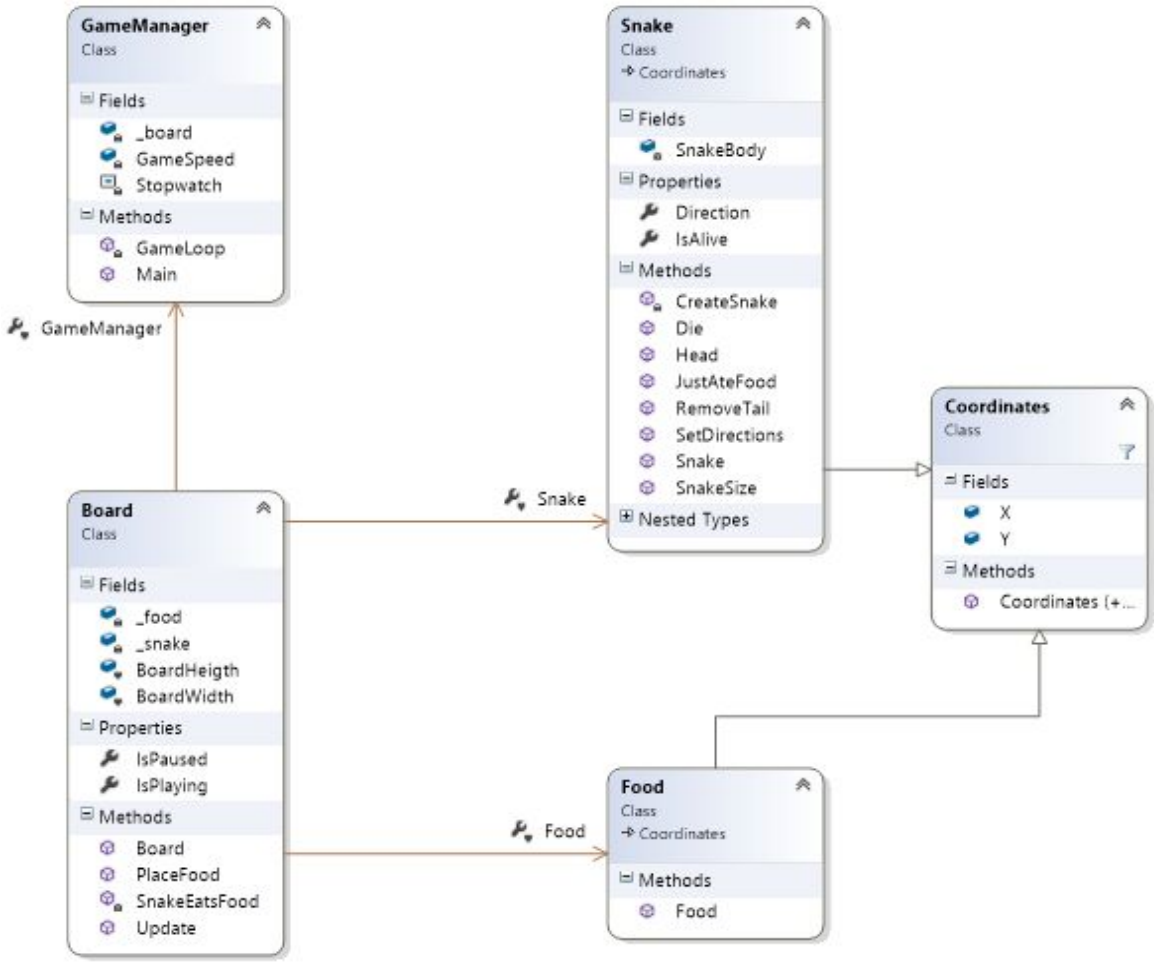
Vi satt oss i gang da med research av hvordan forskjellige snake spill kan kodes for å prøve å lære hva som trengs, etter x antall timer med kode lesing, youtube videoer og tips fra andre studenter følte vi at vi kunne starte med refaktorering av koden.

Dette gjorde at vi ble litt klokere på tankegangen slik at vi kunne begynne å tenke hvordan vi ville at klassediagrammet skulle se ut, med tanken på high cohesion og low coupling.
Deretter følte vi at vi har dannet oss et litt mer oversiktlig bilde av løsningen, og at vi var klare å gå litt dypere inn og at vi kunne begynne å lage en enkel klassediagram uten metodene og attributene (som var litt greiere fordi mange av disse brukte vi fra den originale koden) for å få litt bedre oversikt over hvordan vi skal at klassene skal være, før vi gikk videre med prosessen:



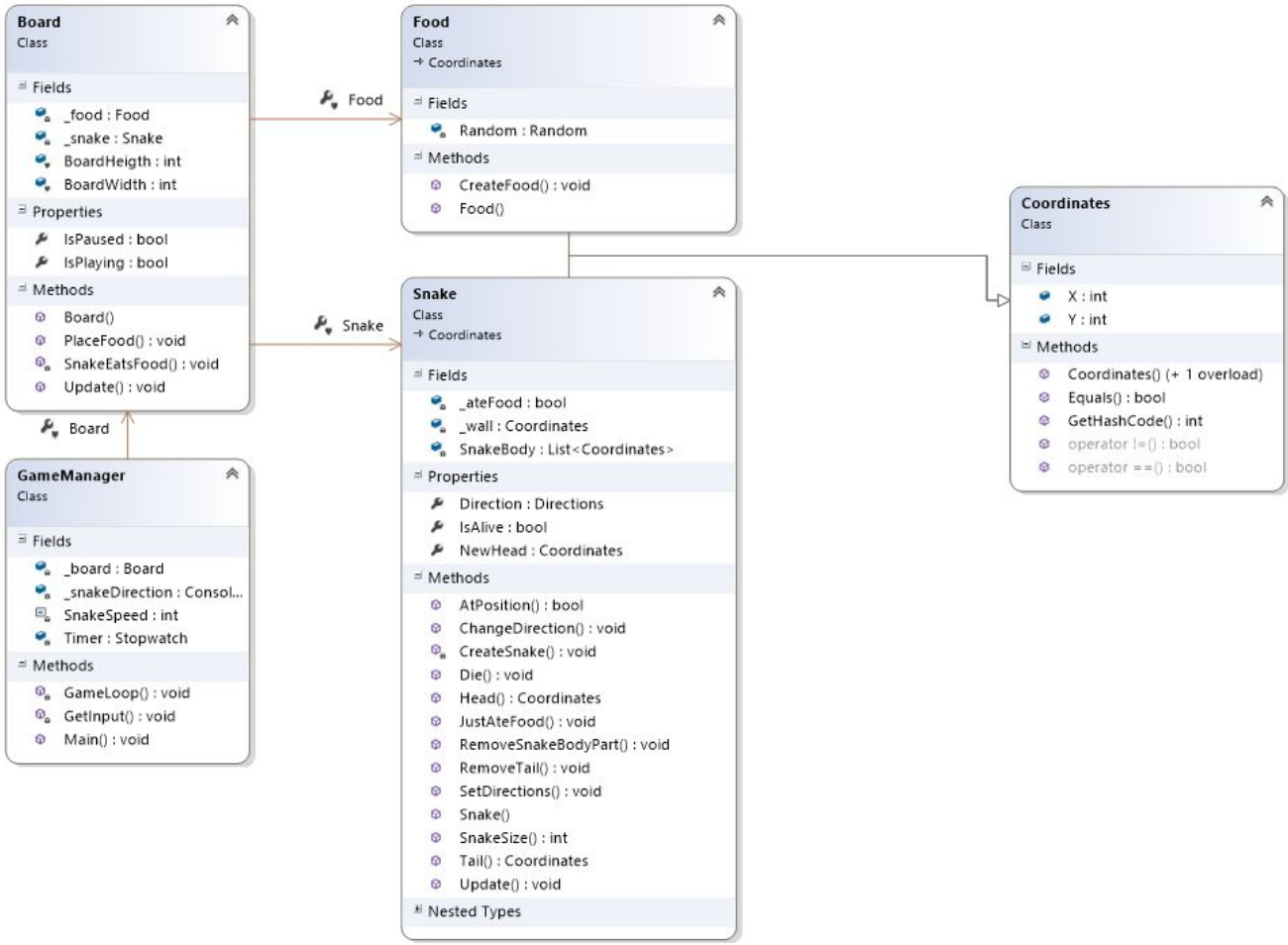
Refaktorering

Etter mye tenkning og skriving av pseudo kode følte vi at vi har dannet oss et litt mer oversiktlig bilde av løsningen, vi var klare å gå litt dypere inn og at vi kunne begynne å lage en klassediagram med metoder som vi kan da bruke som en referanse under refaktoreringen:



Med grunnlag i UML klassediagrammet startet vi med refaktorering av SnakeMess, med tanken på at vi skal opprettholde den originale funksjonaliteten og spillbarheten.
Det var viktig å få alle i gruppen til å samarbeide så vi startet med parprogrammering, hvor en skrev kode og de to andre sto bak og ga forslag mens koden ble skrevet. Vi prøvde å skifte roller jevnt.
Vi brukte Git for versjon kontroll, og i tillegg gjorde dette enklere for oss å dele opp koden slik at hver av oss kunne jobbe med en av de klassene og til slutt kunne vi samle dem opp til et fungerende program, her fikk vi også mye hjelp fra debugging verktøyet ved å sette opp breakpoints, for å få alt til å fungere.
Vi begynte enkelt med å sette opp Coordinates og deretter Board klassen. Food klassen ble også ganske enkel så det ble neste på listen. Etter dette var gjort ble vi veldig ivrige i å få til Snake klassen slik at vi kunne begynne å teste spillet. Da gjenstod det bare GameManager for å få koblet alt sammen.

Klassediagram etter refaktoreringen:



Konklusjon

Etter vi ble ferdig ser vi at det blir mye mer kode enn hvordan det var originalt som ofte kan være et resultat fra refaktorering. Men den er nå mye mer oversiktlig og bedre strukturert. Vi har også hatt flere GRASP guidelines i tankene mens vi drev på med refaktoreringen Alle klassene har fått sine egne arbeidsoppgaver og har ingen andre metoder enn det som var naturlig å ha et annet sted, altså low coupling og high cohesion.
Vi tror at vi gjorde en god jobb til slutt, og føler at vår refaktoreringen av oppgaven oppfyller de kravene som oppgaven stiller.

Multithreading

Oppgave 2

a) Hva er race conditions? Hvordan kan disse oppstå?

Race condition brukes i sammenheng med multithreading.
Race condition er når to eller flere tråder prøver å endre data (variabel) samtidig.
Race conditions kan oppstå når koden ikke er tråd sikker, og flere tråder kan aksessere/ endre data samtidig.
Eksempel på Race Condition:
Varelager i nettbutikk. Det er bare 1 eksemplar av en vare igjen på lager.
Forventet atferd:
To kunder (2 tråder) prøver å kjøpe varen, som betyr at en kunde skal få kjøpe varen, mens den andre må få beskjed om at varen er ikke lenger tilgjengelig.

Tråd 1	Tråd 2		Antall varer
			1
Hent antall varer		←	1
Antall: 1 Kunden kjøper en vare, antall senkes til antall - 1			
Sett oppdatert antall		→	0
	Hent antall varer	←	
		Antall: 0 Kunden kan ikke kjøpe en vare. Kunden får beskjed om at lageret er tomt.	

Hvis koden ikke er trådsikker, kan dette føre til at uforutsette feil kan oppstå:
To kunder (2 tråder) prøver å kjøpe varen, og hvis koden ikke er tråd sikker vil det oppstå et race condition tilfelle hvor begge kundene skal få kjøpe varen, selv om det var bare en vare igjen på lager.

Tråd 1	Tråd 2		Antall varer
			1
Hent antall varer		←	1
	Hent antall varer	←	1
Antall: 1 Kunden kjøper en vare, antall senkes til antall - 1			

	Antall: 1 Kunden kjøper en vare, antall senkes til antall - 1		
Sett oppdatert antall			0
	Sett oppdatert antall		0

Dette problemet kan vi løse med locks. Det fungerer slik at når to tråder prøver å få tilgang til det samme objektet, vil bare en tråd kunne få tilgang av gangen. På denne måten unngår vi at race conditions oppstår.

b) Hva er locks? Hva slags objekter er egnet som locks? Hva er forskjellen på static og local locks? Hvordan benytter vi locks i C#?

Når vi vil styre hvilke tråder som skal få tilgang til å endre data, kalles dette locking.
Det fungerer slik at når to tråder prøver å få tilgang til et og samme objekt, så kan bare en tråd få tilgang av gangen ved at den låses. Når vi låser, hindrer vi andre i å bruke det samme låseobjektet. På denne måten unngår vi at race conditions oppstår.
Beste praksis er å definere et privat objekt for å låse på, eller et privat statisk objekt variabel for å beskytte data som er felles for alle instanser.
Spesifisering av locks: vi skiller mellom static locks og non-static locks, som er en viktig del av multithreading. Static locks brukes for hele instansen, mens non-static locks er “thread safe” bare for denne instansen.
Locks brukes i flere programmeringsspråk, og I C# benytter vi locks på følgende måte: lock(expression) {...}
Eksempel på static og non-static locks:

```
public class LockDemonstration{
        private static Object lockAllObjects = new Object ();
        private Object lockInstanceObject= new Object ();

        public void TestLocks() {
                lock (lockAllObjects ) {
                        // Thread safe for all instances.
                }
                lock (lockInstanceObject) {
                        // Thread safe for this instance.
                }
        }
}
```

Eksempel på en tilfelle hvor det er lurt å bruke locks:
Vi vil lage en metode som trekker en sum fra en bankkonto, og da er det viktig at 2 tråder ikke skal kunne få tilgang til det låste objektet samtidig, fordi da vil man risikere å kunne overtrekke kontoen.

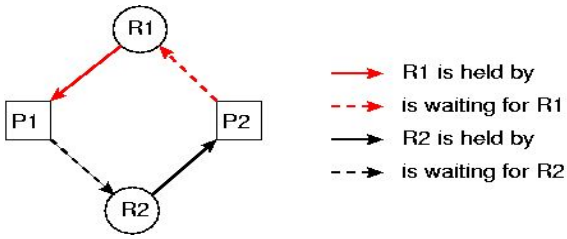
```
private Object _lockObject = new Object();
public void Withdraw(int amount) {
        lock(_lockObject) {
                _account.balance -= amount;
        }
}
```

c) Hva er deadlocks? Hvordan kan disse oppstå?

Deadlock er en beskrivelse av en situasjon som kan oppstå når to eller flere hendelser/ prosesser med locks prøver å aksessere og bruke samme ressurs.
Dette kan skje ved at en prosess A som tar i bruk lock statement, og som bruker en ressurs X, venter på at ressurs Y som er brukt av en annen prosess B og som også tar i bruk en lock statement, skal bli fullført, og samtidig venter B på ressurs X som blir brukt av A. Resultatet blir at begge prosessene A og B venter på hverandre.

Eksempel på deadlock:

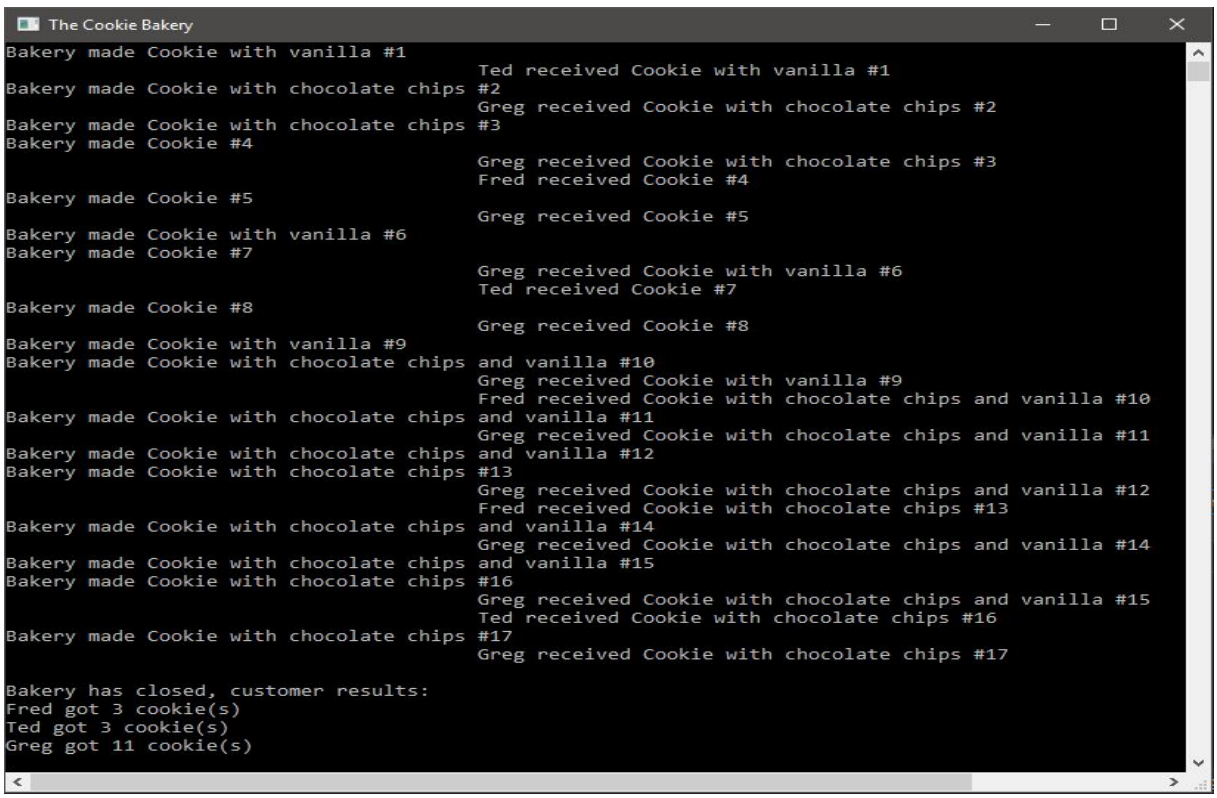
Vi har 2 tråder, A og B. Tråd A låser objekt X, og tråd B låser objekt Y.
A vil aksessere objekt Y og B vil aksessere objekt X.
Begge objektene er låst til en tråd og begge trådene prøver å aksessere det andre objektet, og da oppstår det en Deadlock.



The Cookie Bakery

Oppgave 3

Skjermbilde av programmet når den kjøres:

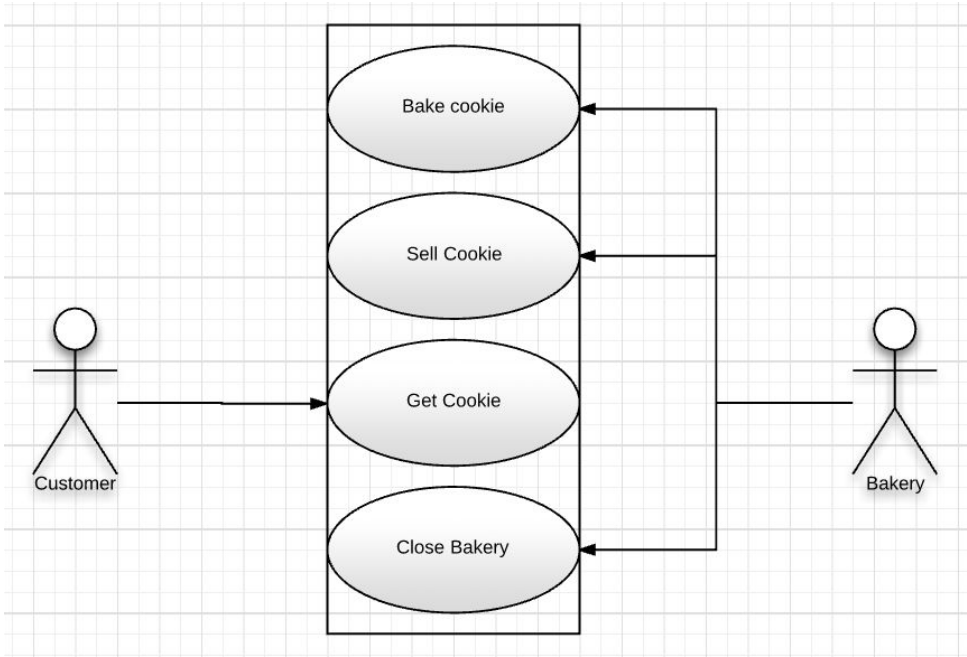


Introduksjon

Vi har fått en oppgave som går ut på å lage en Cookie Bakery som lager cookies og som har tre faste kunder hvor alle prøver å få tak i cookies samtidig. Vi må sørge for at koden skal være tråd sikker slik at bare en kunde kan få tak i hver cookie som blir laget. Løsningen vår implementerer mulighet til å kunne lage flere typer cookies. Vi prøvde å løse oppgaven med tanken på bedre arkitektur og kode, og vi prøvde å følge design guidelines, vise at vi behersker multithreading og at vi tar i bruk design patterns. Som i Snake Mess har vi også brukt parprogrammering, og vi brukte Git for versjon kontroll. Når det gjelder bruk av variabelnavn, metodenavn og klassenavn brukte vi ReSharper som hjelpemiddel.

Arbeidsprosess

Vi valgte å møtes på skolen hver gang vi skulle jobbe. Vi startet med å diskutere oppgaven og kjørte en liten brainstorming av hvordan vi tenker at alt skulle gjøres.
Vi var alle enige om at det var var lurt å lese mer om multithreading og design patterns før vi gjorde noe mer, så vi satte i gang med å studere og utforske tema. Etter at vi følte oss mer trygge på funksjonalitetene som skulle trenges for å utføre oppgaven. For å få litt bedre oversikt over hvordan programmet skal se ut, lagde vi en UML Use Case Diagram, før vi begynte med kodingen.

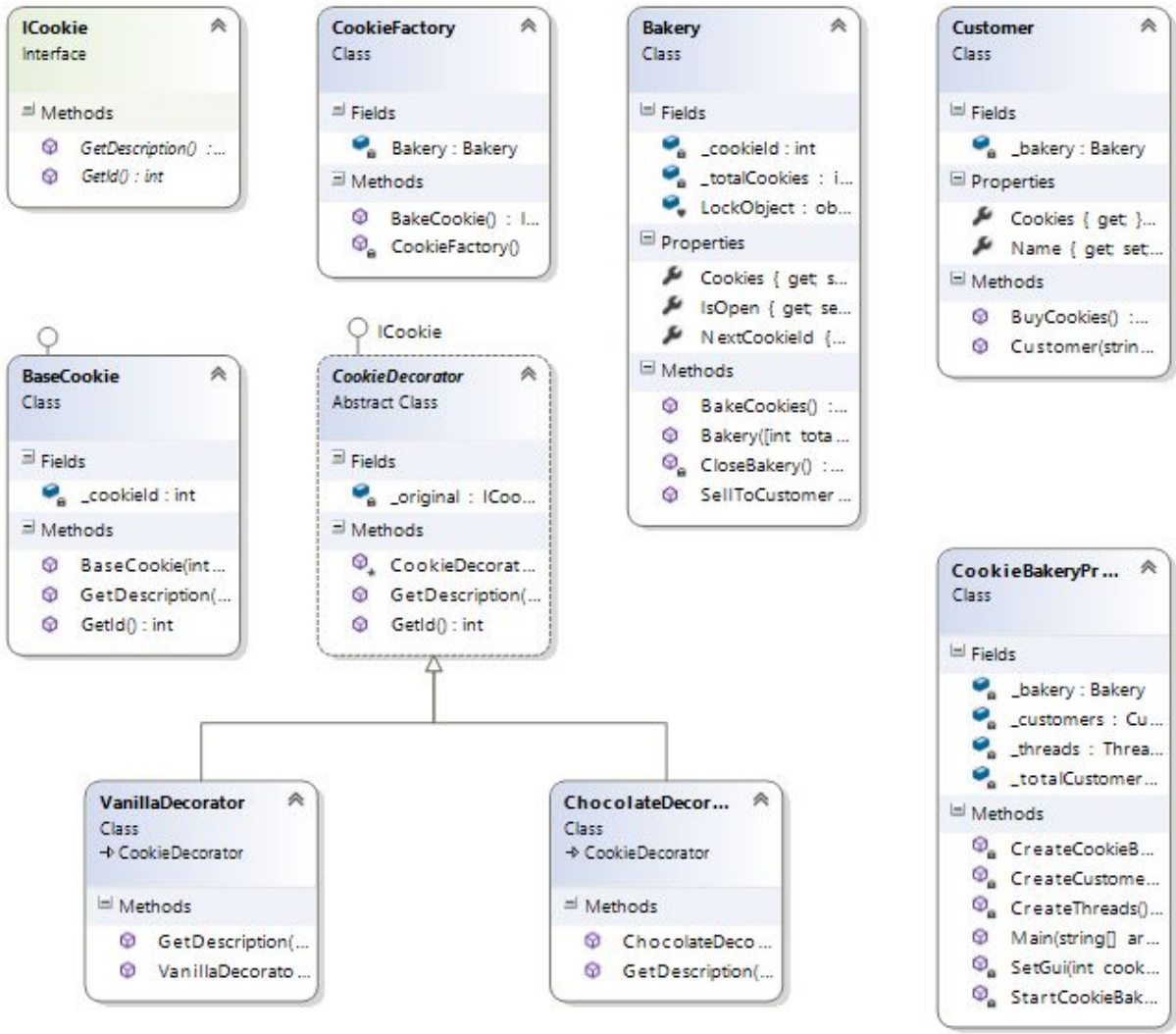


Use case diagrammet viser hvordan vi se for oss prosessen for et ferdig produkt.
Bakeriet lager cookies som og selger dem til kundene som prøver å få tak i dem, og når alle cookies er solgt stenger bakeriet.
Klassene og metodene i The Cookie Bakery gjør kun relaterte oppgaver, og hver klasse/metode har ansvar for en spesifikk oppgave. Kort oversikt over klasser:

Cookie Interface	Interface med getDescription og GetId for cookies
Base Cookie	Arver fra cookie interfacet og brukes som en base for andre typer cookies
Cookie Decorator	Abstrakt hjelpeklasse brukt av andre decorator klasser til å lage forskjellige typer cookies
Vanilla Decorator	Arver fra decorator, overskriver getDescription

Chocolate Decorator	Arver fra decorator, overskriver getDescription
Cookie Factory	Ansvar for opprettingen av forskjellige tilfeldige cookies.
Bakery	Legger ut cookies som skal selges til Customers helt til alle cookies er solgt, så stenger bakeriet.
Customer	Hver tråd representerer en kunde som skal kjøpe cookies og kaller på buyCookies metoden.
Cookie Bakery Program	Eksekveringen av selve programmet

Klassediagram:



Forklaring på løsningen

Design guidelines

Vi hadde flere GRASP guidelines i tankene mens vi drev på med kodingen av The Cookie Bakery. Vi har fulgt C# sine konvensjoner for navngivning av variabler, klasser og metoder, og alle klassene har fått sine egne arbeidsoppgaver og har ingen andre metoder enn det som var naturlig å ha et annet sted, altså low coupling og high cohesion. High cohesion vil at klassene ikke skal ha for mye ansvar og heller ikke ha ansvar for noe som ikke har noe med klassen å gjøre, for eksempel en cookie klasse skal ikke trenge gjøre noe med en Customer klasse. Low coupling innebærer å ha så få koblinger mellom klasser som mulig, det må være noen koblinger, men dette skal holdes til et minimum.

Creator

Vi har laget klassen CookieBakeryProgram som creator, det er denne klassen som har ansvar for oppretting av objekter som trengs for å kjøre programmet.

Design patterns

For å videreutvikle koden valgte vi å lage et Cookie interface til cookiene våre.

Decorator

Vi valgte også å bruke decorator design pattern som arver av interfacet til å gi cookiene forskjellige smaker. Vi valgte å bruke denne pattern siden den gir oss akkurat det vi trengte med tanke på at alle cookiene skulle være like men bare at de kunne ha forskjellige smaker. Bakeriet lager cookies med tilfeldig smak, Base Cookie arver fra Cookie Interface og brukes som grunnlag for flere smaker, og med flere typer som kan legges til ved å opprette flere decorator klasser.

Factory

Vi ville at det skulle være en egen klasse som opprettet de forskjellige cookiene for oss, vi tenkte da at det passet å bruke en Factory design pattern, og lagde Cookie Factory.

Multithreading

Vi tok i bruk multithreading, siden hver kunde skulle behandles som egen tråd. Siden alle skulle kalle på samme metode tok vi i bruk locks, som løser problemet med race conditions, som kan påvirke det endelige resultatet. Med locks rundt sellToCustomer metoden, sørger vi for tråd sikkerhet slik at race conditions ikke skulle skje, med andre ord at cookie må fortsatt finnes når kunden mottar den. Dette sikrer at når kundene prøver å få cookies samtidig at bare en kunde kan få tak i hver cookie som lages.

Konklusjon

Vi føler at vi gjorde løsningen på en bra måte, fordi alle funksjonalitetene som var krav har vi implementert i vår løsning, som viser at vi har forståelse for hvordan vi skal ta i bruk design guidelines, f.eks vi har en Creator, vi lagde klasser med tanken på high cohesion og low coupling.
Når det kommer til design patterns så har vi en Decorator og en “Factory”.
Vi tok i bruk multithreading for å vise at vi forstår hvordan denne funksjonaliteten fungerer, og at vi kan lage et tråd sikkert program som bruker locks.

Kilder:

1. Deadlock:http://images.google.de/imgres?imgurl=http%3A%2F%2Fwww.cs.fsu.edu%2F~baker%2Fcop5611.S03%2Fgraphics%2Fbasicdeadlock2.gif&imgrefurl=http%3A%2F%2Fwww.cs.fsu.edu%2F~baker%2Fcop5611.S03%2Fdeadlock.html&h=209&w=446&tbnid=6pq9QL4MDNHkBM%3A&docid=IVBvQfYb_9xG1M&ei=1UYaWMieLcWrsAHmnIPIDQ&tbm=isch&iact=rc&uact=3&dur=431&page=1&start=26&ndsp=25&ved=0ahUKEwiI3eua74rQAhXFFSwKHWbOANkQMwhPKCwwLA&bih=760&biw=1536
2. https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx