Markus Nyland, Vinh Nguyen, Daniel B. Stoa 20. nov 2016

Innlevering PG3300 - Software Design

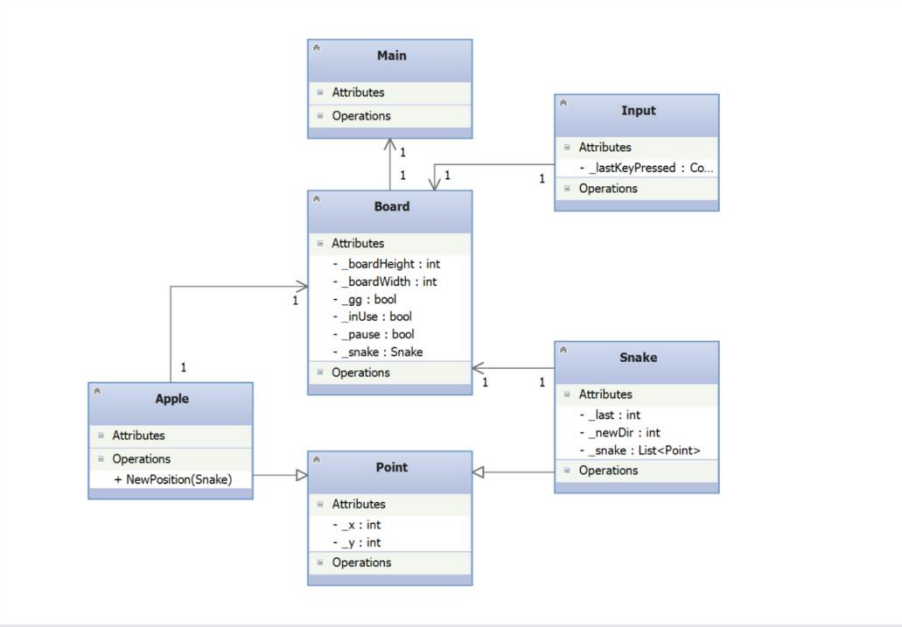
Innledning

Oppgaven går ut på å refaktorere SnakeMess, svare på spørsmål knyttet til multithreading og å lage et bakeri som selger kjeks ved hjelp av multithreading og locks. Vi går nærmere inn på hvordan vi har gått fram med oppgavene, og beskriver diverse design patterns vi har brukt der hvor det passer.

Oppgave 1 - SnakeMess

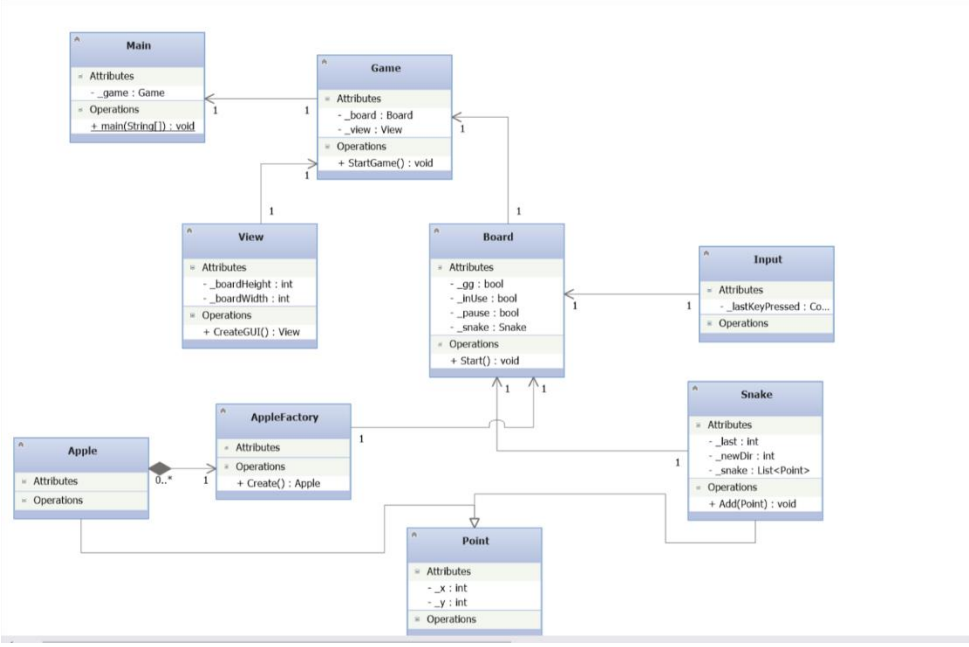
Fremgangsmåte

Vi begynte med å se på hvordan koden kjørte i sin helhet. Kjørte programmet noen ganger, så på hva man kunne gjøre og hva som skjedde om visse kriterier ble oppfylt. Deretter så vi dypere på koden, prøvde å skjønne hva som faktisk skjedde. Vi satte kommentarer på steder det var litt ekstra uklart, så vi kunne lettere få et overblikk over hva koden gjorde.

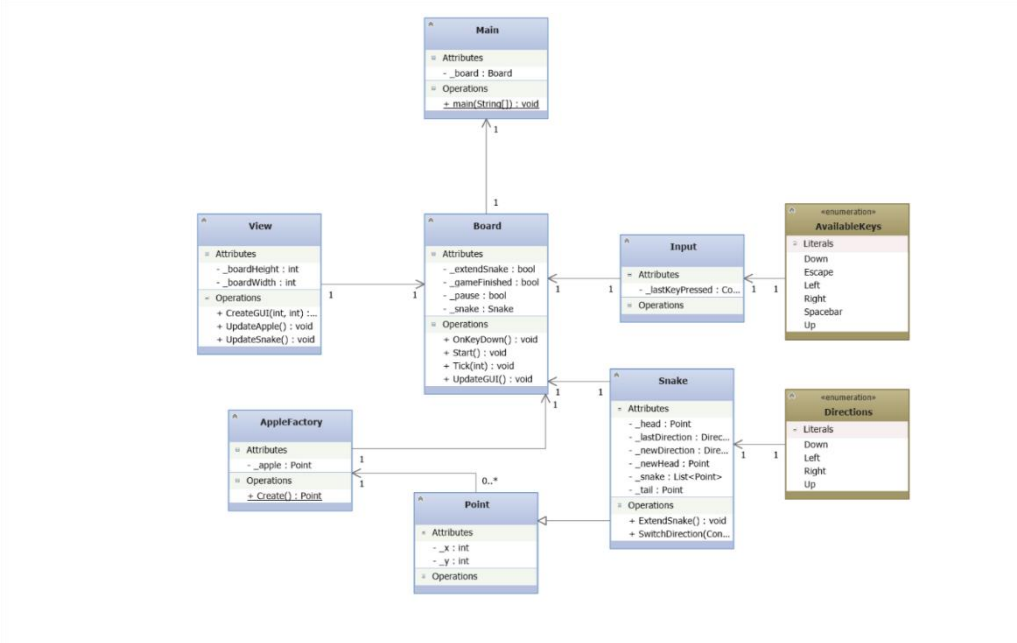


Vi begynte så med klassediagrammet. I første omgang prøvde vi å sette inn fields som SnakeMess allerede brukte, for å se hvilke klasser de kunne tilhøre til. Dette ble da vårt første utkast til et

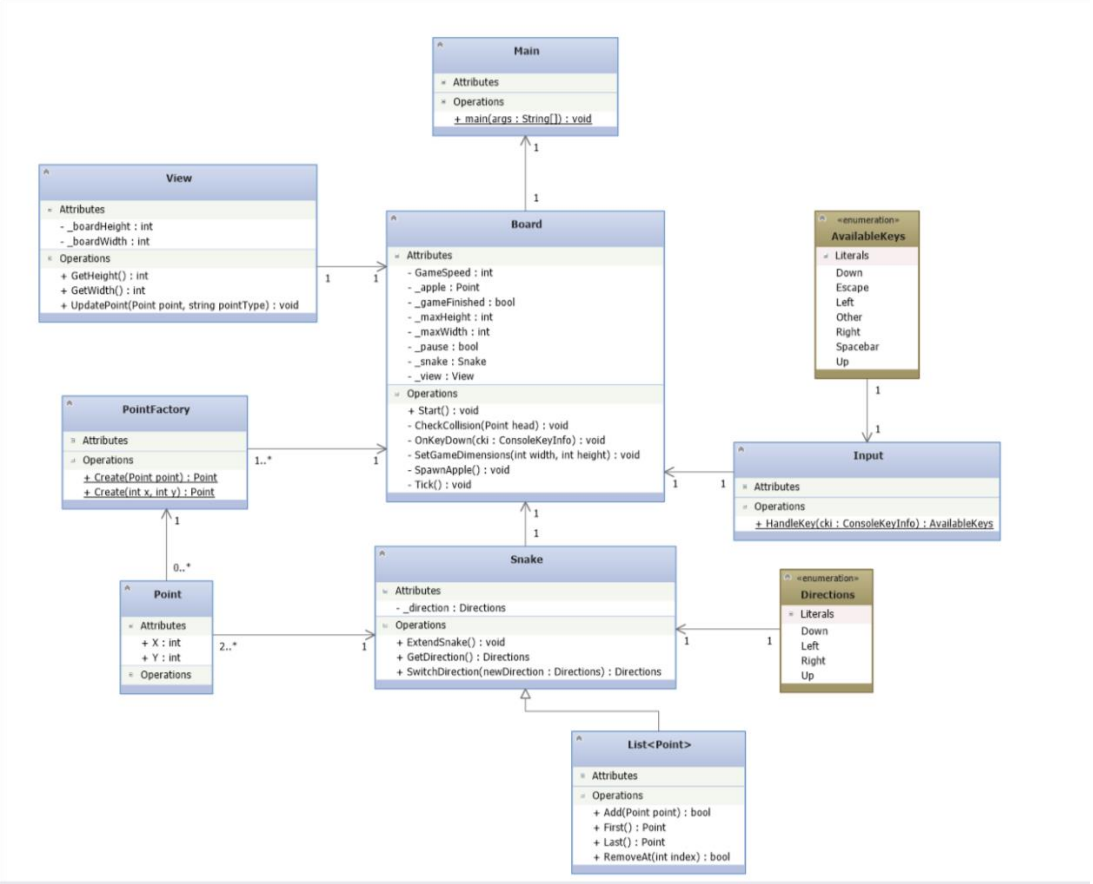
klassediagram, hvor Board fungerer som en controller som holder styr på alle de andre klassene og objektene.



I figur 2 bestemte vi oss for å splitte ansvaret som Board hadde i flere klasser. Vi lagde en View klasse som oppretter et vindu som lar oss spille på, og Game kobler logikken til Board og grensesnittet til View sammen, så Main for kjørt metoden sammen. Videre fant vi ut av vi ville implementere flere design patterns for å vise forståelse av faget. Vi inkluderte da et AppleFactory, som har en statisk metode Create() som lager et nytt eple. Vi er klar over at å endre posisjonen på eplet framfor å lage nye Apple-objekter ville vært lurere ytelsesmessig, men for å vise at vi forstår Factories gjorde vi det på denne måten.



Etter å ha kommet med et forslag på klassediagram satte vi oss ned og parprogrammerte. Vi gjorde dette fordi vi følte vi hadde et ganske greit klassediagram, og ville se hva som kom til å funke. Dersom vi hadde gjort noen feil i klassediagrammet kom vi til å se disse, og oppdatere klassediagrammet med små endringer. Vi så tidlig at _x og _y i Point måtte være properties. Dette var for å kunne endre og hente ut verdiene til punkter. AppleFactory kunne heller ikke bruke en statisk metode med et field, så vi sløyfet fieldet og hadde to int-verdier som parameter til metoden.



Over ser vi den ferdigstilte løsningen på vårt klassediagram. Det ble litt forskjellig fra hvordan det så ut før vi begynte å programmere, fordi vi så flere lure løsninger og at vi kunne dele ting inn i metoder. Et merkbart eksempel er at Snake nå arver fra List-klassen, slik at Snake ER en liste med Points i motsetning til å være en klasse som har en liste med Points.

Problemer

Vi oppdaget tidlig et problem med oppgaven, som var at SnakeMess var veldig uforståelig veldig lenge. Vi prøvde tidlig å finne ut av hva deler av koden gjorde, men vi måtte dobbelsjekke flere ganger og prøve å tyde hva som faktisk skjedde. På grunn av dette tok det litt lenger tid å lage klassediagrammene, fordi vi ville bruke eksisterende kode og måter å gjøre ting på framfor å lage et helt nytt program. Dette viste seg å være ganske vanskelig på grunn av hvor horribel koden var, så det ble en del forandringer.

Som følge av den vanskelige koden slet vi også med å få skrevet ut riktig på skjermen. Vi så ikke før etter vi selv hadde prøvd å lage vår versjon at slangen går “baklengs”, altså hodet er det siste leddet i listen. Vi trodde dette var for å villlede oss fra oppgavens side, men da vi skjønte det falt ting litt mer på plass. Likevel tok det lang tid å finne en måte å få riktig output på. Vi så at spillet vårt funket i en mer “tekst-basert” visning hvor vi ble vist koordinatene til punktene på slangen og eplet, som funker som vi ville. Det gikk bra til slutt, selvom vi heller ville gjort det på en litt annen måte fordi det ser litt stygt ut.

Hva funker bra

Vi har delt løsningen inn i flere klasser som har hvert sitt eget ansvarsområde. Hvis vi ville hatt en del klasser til hadde det vært mulig å splitte Board-klassen til noen flere klasser som har enda mer konkrete arbeidsoppgaver, som kollisjon og et game-tick, men vi valgte å holde det på denne måten fordi Board er løsningens Controller. Vi vil si at løsningen vår følger MVC-design patternet, hvor View er view, Board er Controller og resten er Models. Ingen av modellene vet at de blir brukt til et spill, Board har kontrollen på hva som skjer og gir input til View, mens View lager et vindu og har metoder for å displaye punkter til konsollen.

Løsningen vår følger og Low Coupling og High Cohesion. Mange klasser med klart definerte arbeidsoppgaver var noe vi bestemte oss for å ha med fra starten av. I tillegg har vi med Factory design patternet, som lager punkter gitt et annet punkt eller x og y koordinatene til det nye punktet.

Oppgave 2 – Multithreading

I denne delen av oppgaven skal vi beskrive multithreading og begrepene race conditions, locks og deadlocks.

a) Race conditions

Race condtions går ut på at to forskjellige tråder skal lese fra og endre på en variabel samtidig. Gitt at vi har to tråder tråd1 og tråd2 og en variabel x = 5:

Dersom tråd1 prøver å hente ut x, får tråd1 tilbake 5 som er den opprinnelige verdien. Tråden kaller på en metode som gjør x = 8. Tråd2 henter ut x, som er lik 8, og kaller en metode som endrer x til 10. Dette fungerer fint, dersom trådene gjør instruksjonene i denne rekkefølgen. Dette er ikke en selvfølge, siden det er usikkert hvilke av instruksjonene i trådene som blir kjørt på hvilken tid. Gitt samme variabler som over:

Dersom tråd1 henter ut x får tråd1 tilbake 5, som over. Men plutselig kan tråd2 også hente ut x, som gir 5, og kalle på sin metode som gjør x = 10. Deretter kan tråd1 kalle på sin metode, som gjør x = 8.

Hvilken av versjonene over som blir kjørt er umulig å vite i første omgang. Dette fordi instruksjonene på hver tråd kan kjøre en etter en, eller alle på en gang eller noe i mellom før den andre tråden får gjøre sin oppgave. Man kan forhindre dette ved å bruke locks, som fører oss til neste deloppgave.

b) Locks

En «Lock»-statement i C# sørger for at kun én tråd kan aksessere og kjøre en kodebit i et objekt om gangen. For å løse race conditionen vi hadde i oppgave 1a, hvor det var 2 tråder som hadde adgang til 1 felles objekt, kan man da bruke en lock-statement for å avgjøre hvilke av trådene som skal ta eierskap over kodebiten. Tråden som blir gitt eierskap må gi den fra seg før en annen tråd kan ta over – Og med dette løser vi race condition.

Locks kommer i diverse former og her skal vi se på forskjellen mellom «static» og «local. Static lock bruker vi gjerne dersom vi ønsker å låse alle instanser i en statisk metode. Om vi derimot vil låse en spesifikk instans, bruker vi da en local lock istedenfor.

For å demonstrere hvordan vi tar i bruk av locks i C#, så kan vi ta en nærmere titt på denne kodebiten;

```
Class Items
{
   int ItemsInStock;
   private Object thisLock = new Object();

   private void ItemsSold(int amount){
            lock(thisLock){
            ItemsInStock -= amount
            }
}
```

I dette eksempelet bruker vi locks gjennom nøkkelordet thisLock. I denne metoden ser vi dersom en vare blir kjøpt, vil antall varer på lageret dekrementeres. For å unngå at denne hendelsen skal skje samtidig i to tråder (som kan føre til at systemet selger flere varer enn det den har.), har vi brukt thisLock på ItemsSold, dette vil resultere med at den første tråden som kjører ItemsSold vil da få låse seg for å forhindre at andre tråder skal få tilgang til den.

c) Deadlocks

En deadlock er et fenomen som oppstår når to eller flere tråder blokkerer hverandre fordi begge to har låst en ressurs som den andre tråden spør etter. Forestill deg to ressurser A og B som blir brukt av tråd 1 og 2. Tråd 1 skal gjøre noe og tar A, deretter prøver både tråd 1 og 2 å bruke B, denne «kampen» vinner tråd 2. Tråd 2 trenger nå ressurs A, men denne er låst av 1 samtidig som 1 trenger B som 2 nå har låst. Dette er en deadlock.

Oppgave 3 - The Cookie Bakery

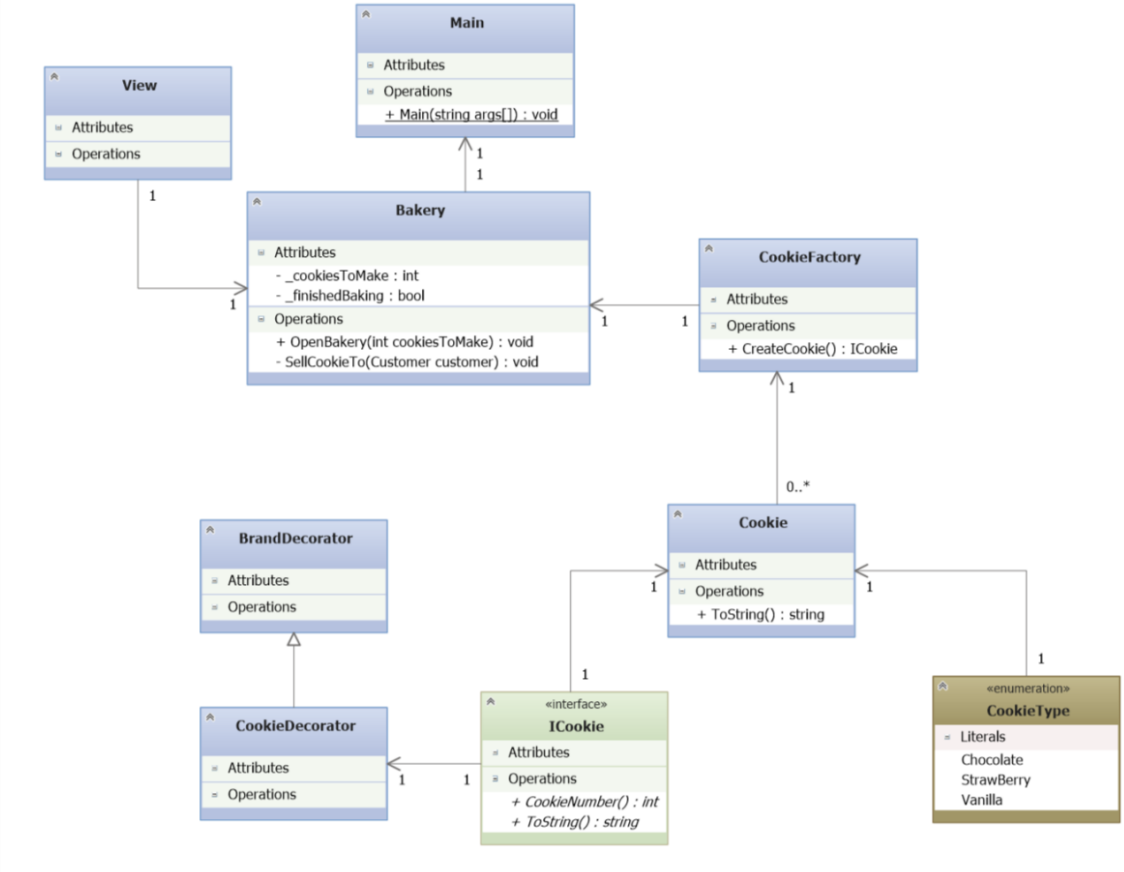
Fremgangsmåte

Prosessen har vært veldig lik oppgave 1. Vi jobbet som regel på skolen etter forelesninger, hvor vi fikk muligheten til å parprogrammere. Vi bestemte oss for å gjøre ekstra arbeid hjemme dersom det var nødvendig, hvor vi tok kontakt gjennom Facebook.

Denne oppgaven la mye vekt på threading og siden vi splittet oppgave 2 slik at alle i gruppen kunne besvare på 1 oppgave hver, satt vi oss ned for å se på hva hver og en av oss hadde skrevet, slik at vi fikk et bedre innblikk på hvordan vi skulle gå fram med denne oppgaven. Deretter satt vi sammen en UML use case for å få bedre oversikt over programmet. Da vi ble fornøyde med use case, gikk vi over til å lage første utkast av klassediagrammet før vi begynte å programmere. Vi satt flere ganger fast,

men med litt undersøkelse på nettet kom vi sakte, men sikkert framover. Vi la til Decorator helt ved slutten for å vise at vi hadde en forståelse på Design Patterns. Etter at vi var ferdige med å programmere lagde vi et fullstendig klassediagram.

Klassediagram

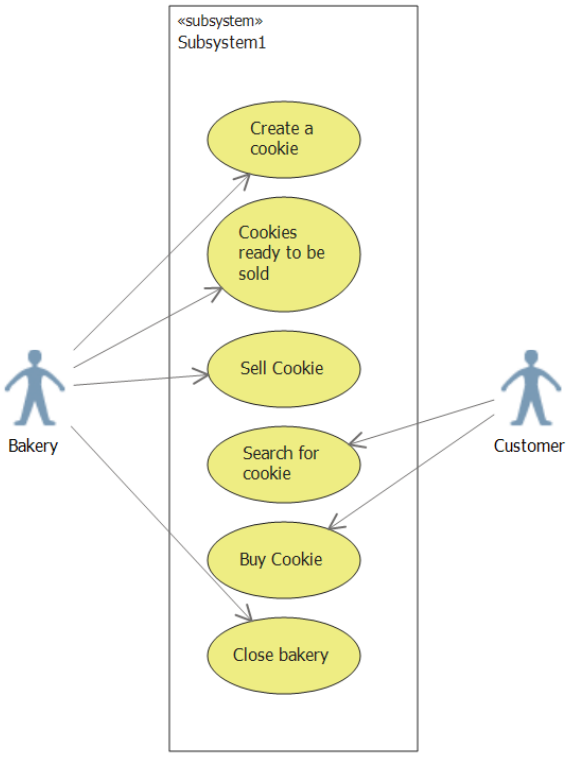


Use Case Diagram

Under viser vi hvordan vi ser for oss bakeriet:

- Bakeriet lager først en kjeks og legger den ut for salg.
- Kunden ser først etter om det er kjeks før han kjøper.
- Bakeriet selger kjeksen dersom det er kjeks igjen.

- Dersom det er solgt ut, stenges bakeriet for dagen.



Multithreading

I oppgaven brukte vi Multithreading hvor hver kunde skulle være en tråd. Oppgaven spurte om 3 kunder, og vi var nødt til å lage tre kunde-objekter som tilsvarte 3 tråder. Som vi forventet(etter oppgave 2) skapte dette race conditions siden alle trådene kunne aksessere én metode samtidig(dette førte til at 2 kunder kunne få samme kjeks som påvirket sluttresultatet). For å håndtere dette problemet fant vi fort ut at vi måtte bruke lock-statements. Vi pakket CheckCookies() metoden i Bakery i locks. Dette forhindret race conditions med å oppstå.

Design Pattern

Vi gjorde bakeriet vårt litt mer spennende ved å implementere Decorator. I tillegg til de vanlige cookie smakene(chocolate, vanilla, strawberry), inkluderte vi også tilfeldige premium kjeks.

Vi har også brukt Factory patternet, som gir oss en kjeks med forskjellig smak. Her lages det mange objekter som ser veldig like ut, så ved å kombinere Factory sammen med Decorator vil vi få en klasse som håndterer både lagingen av ny kjeks via en statisk metode, og dekoreringen av kjeksen i form av premium kjeks.

Her er da Bakery vår controller som holder orden på klasser og håndterer input av brukeren. Dette da ved hjelp av en annen klasse som gir oss tilbake hvilken knapp som ble trykket på. Vi har også

oppnådd low coupling og high cohesion, hver klasse har et bruksområde og koden er forståelig og ganske uavhengig av andre klasser.

Problemer

Et problem i koden er at vi kun får solgt til en og en kunde selv om det er to kjeks ute for salg samtidig. Vi prøvde flere forskjellige alternativ, hvor flere av de ga oss Exceptions uten hell. Vi kom dessverre ikke på en løsning som tok hånd om dette, selv om det ser ut som at locken skal fungere.