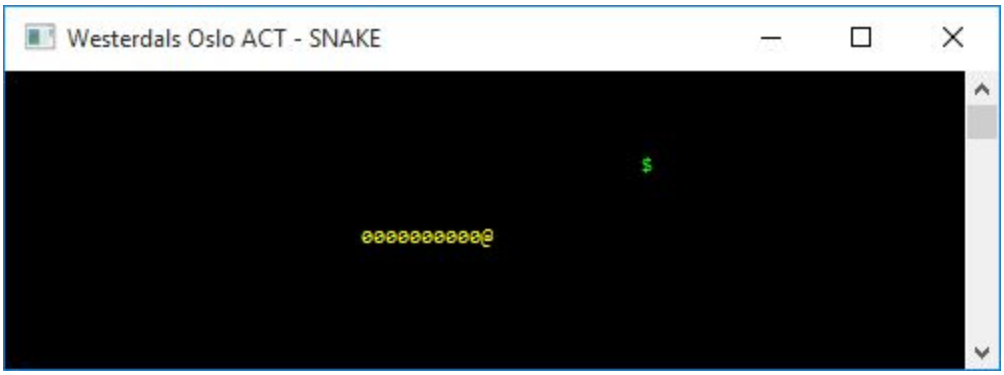
PG3300 Innlevering – SnakeMess

https://bitbucket.org/OlavOlsm/pg3300-innlevering/



Et eksempel på kjøring av spillet SnakeMess

Prosess

For best mulig samarbeid, valgte vi å bruke Git for deling av kode. Til å lagre git repository brukte vi BitBucket, fordi den tilbyr gratis private repositories for opp til 5 personer. Ved bruk av Git sørget vi for at begge har siste versjon til enhver tid og for å ha en backup og mulighet for å gå tilbake til tidligere versjoner. Vi tok i bruk issues på BitBucket for å ha oversikt over oppgavene i innleveringen. Her opprettet vi også bugs vi fant underveis for å huske å rette dem senere.

Gruppen ønsket å jobbe hjemme, så vi tok i bruk Teamviewer, Facebook Messenger, og Slack for å enkelt kunne samarbeide og kommunisere. Teamviewer brukte vi for å se hva den andre personen gjorde på sin pc. Facebook messenger og slack ble brukt for å kommunisere. Det fungerte svært bra, fordi det er lettere å konsentrere seg hjemme uten bakgrunnstøy.

Først forsøkte vi å analysere koden for å forstå funksjonen til de ulike kodesnuttene. Koden kompilerte og kjørte, men var vanskelig å lese, utvide og gjenbruke. Det var nesten ingen kommentarer og navngivingen av variabler og kode var uklare. Nesten all koden var i main, ikke delt opp i klasser eller metoder, utenom klassen Coord for koordinatene.

For å få bedre oversikt laget vi UML-klassediagram ut fra daværende kode. Etter å ha opprettet klassediagrammet, så endret vi inndelingen av klasser og metoder slik vi planla å lage vår implementasjon.

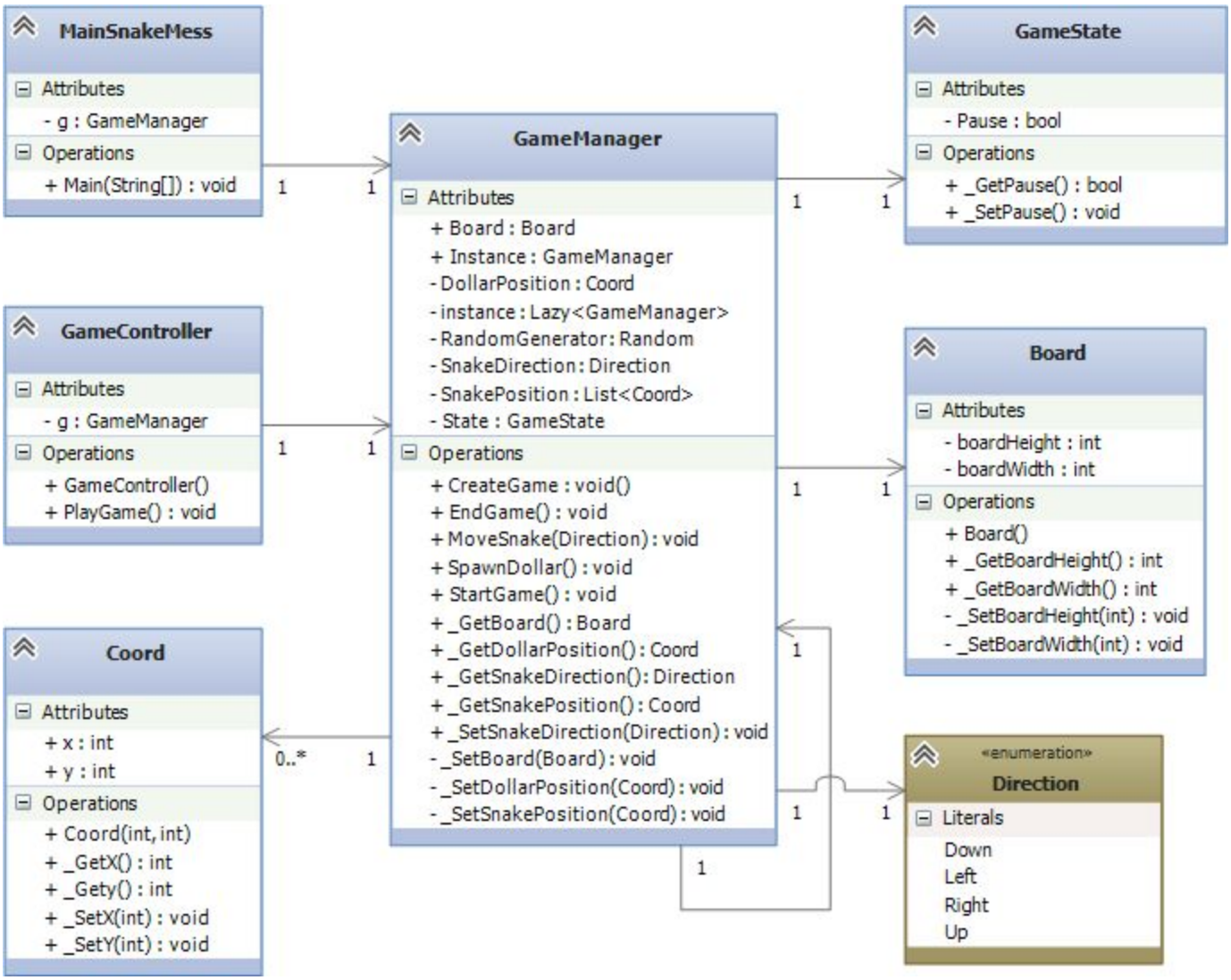
Da vi anså klassediagrammet som ferdig, opprettet vi klasser og metoder for programmet. Underveis i koding av logikken kom vi på ting vi hadde glemt å ta med i klassediagrammet. Dermed ble vi nødt til å endre diagrammet mens vi kodet. Se endelig klassediagram på neste side.

Vi gjorde research på det vi var usikker på underveis i kodingen. Blant annet undersøkte vi mer om design patterns og valgte legge til Singleton slik at kun ett spill kan kjøre av gangen.

Etter programmet var fullført, genererte vi UML klassediagram ut fra koden i Visual Studio 2013, for å få med relasjoner (2015 viser ikke det ordentlig) og sammenligne med vårt eget klassediagram.

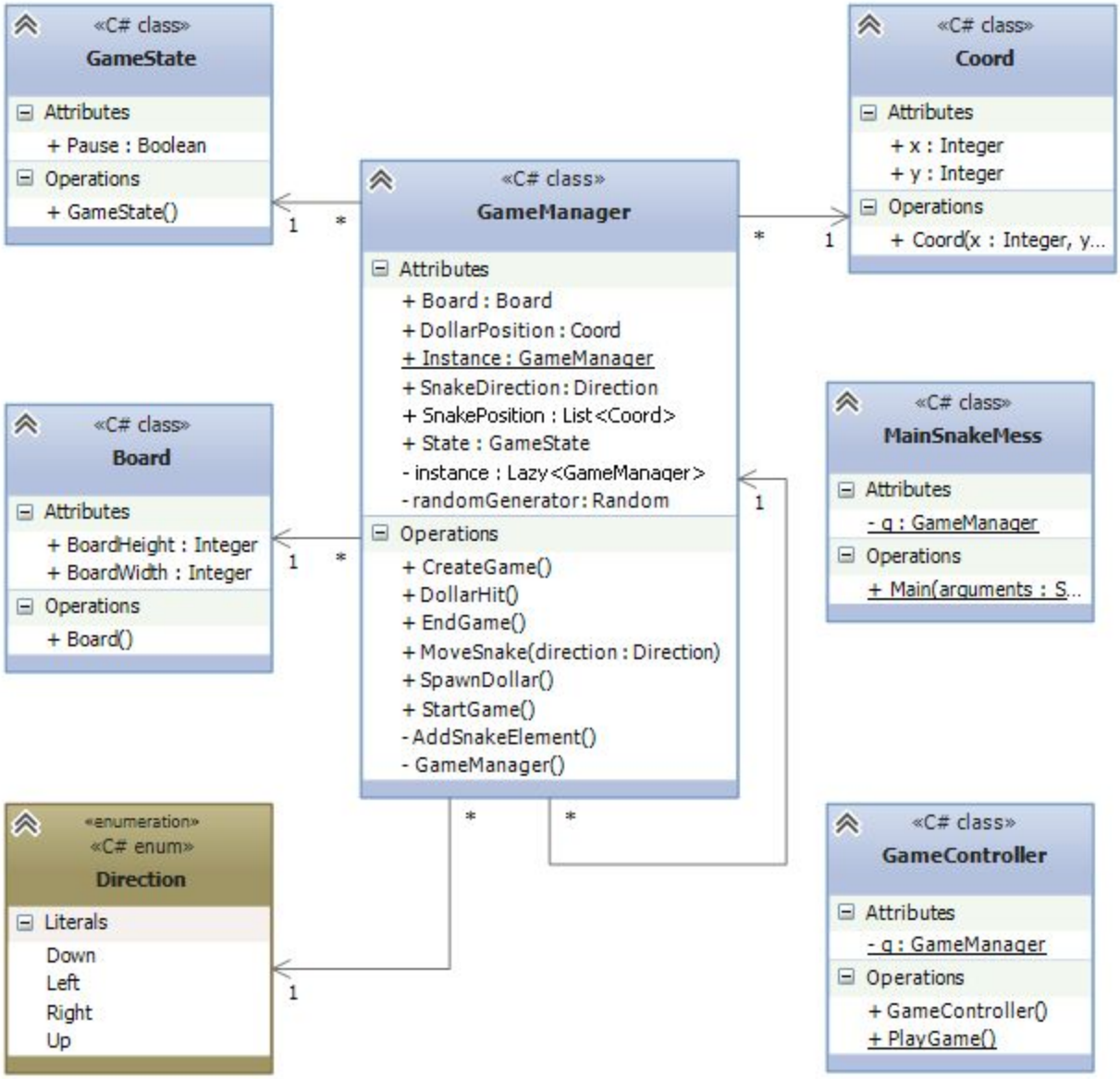
UML Klasse diagram laget før kode

Dette er vårt endelig klassediagram. Vi måtte endre den mye underveis i kodingen, da vi kom på ting vi hadde glemt og annet vi ville endre og legge til for å forbedre funksjonaliteten og strukturen. For eksempel var Singleton design pattern noe vi la til på slutten fordi vi tenkte at det var nødvendig og lurt å ta med. Et klassediagram skal gi oversikt og brukes som et hjelpemiddel, og trenger ikke være helt riktig før man begynner å kode. Når vi gjorde endringer underveis, så vi også lettere om det var lurt eller ikke når vi endret diagrammet. Vi følte at den var til stor hjelp for oss.



UML Klasse diagram generert fra kode

Etter vi var helt ferdig med koden, genererte vi UML klasse diagram fra koden, for å sammenligne med klassediagrammet vi laget selv. Visual Studio 2015 viste ikke relasjoner ordentlig, så vi valgte gjøre det i Visual Studio 2013. Klassediagrammet lignet på den vi laget selv og gav oss bekreftelse på at koden er slik vi hadde tenkt. Vi er usikker på om relasjonene i klassediagrammet Visual Studio opprettet er riktig. Vi tenkte det var relasjon fra MainSnakeMess og GameController til Gamemanager fordi instansen til GameManager hentes og det kalles på metoder i den. Dette klassediagrammet viser også mange forhold på alle koblingene på siden til GameManager. GameManager er singleton så det kan bare være en instans av den, så det ser ut som Visual Studio gjør feil på dette, eller at det betyr noe annet.



Valg for refaktorering av kode

Vi valgte å bruke C# konvensjoner / standarder for navngiving av variabler, metoder og klasser. Til dette brukte vi Resharper som hjelpemiddel. Noen valg har vi gjort for å holde koden bakoverkompatibel, slik at den også kan kjøres i tidligere versjoner av C# og ikke bare C#6. Selv om visual Studio 2015 har c#6 som standard er det også mulig å kjøre tidligere versjon der. For eksempel har vi skrevet { get; private set; } i properties der { get; } gjør det samme i C#6. I GameController kunne vi brukt using static for å fjerne redundant "GameManager.Direction." og "ConsoleKey." for å gjøre koden mer lettlest. Men dette er bare på noen få linjer av vår kode og vi valgte derfor heller beholde bakoverkompatibilitet.

Design pattern

For å hindre at GameManager kan opprettes i flere instanser har vi tatt i bruk design patternet Singelton. Dette valgte vi å gjøre fordi det ikke bør være flere enn ett spill kjørende på en gang. Vi brukte en god løsning fra csharpindepth.com; Opprette en privat static readonly instanse ved å bruke System.Lazy, gjøre konstruktør private, og opprette en public static property for å returnere instansen.

Klasser

Vi splitter koden i forskjellige klasser for å gjøre det mer oversiktlig. Hver klasse legges i sin egen .cs fil. Her er noen av klassene vi har opprettet:

- MainSnakeMess: Programmets main metode oppretter instans av GameManager, og kaller metodene CreateGame og PlayGame for å starte spillet. Den setter også retningen på slangen til å gå nedover når spillet starter.
- GameStates: En egen klasse for status i spillet. Har en C# property som heter pause, med get og set metode. Selv om denne bare har en variabel, gjør det koden mer oversiktlig og gjør det lett å videreutvikle programmet i ettertid.
- Coord: Lagrer koordinater til elementer i spillet ved å opprette to properties, x og y.
- Board: Setter opp spillerbrettet (konsollen) med innstillinger, som at cursor ikke skal være synlig og tittel på vinduet.
- GameManager: Er spillets "creator". Den har hovedansvaret for oppretting av objekter som tar seg av spillets funksjoner. Her er metoder for alt som skal gjøres i spillet, blant annet flytte slange, gjøre slange lenger, plassere ny dollar i tilfeldig posisjon ved hjelp av Random-funksjoner integrert i C#. Vi kan derfor si at klassen har en "high cohesion" fordi objektene kun gjør relaterte oppgaver. Denne klassen har gjort det mulig for oss å ta i bruk "low coupling" i og med at den har hovedansvaret for oppretningen av objekter og metoder for viktige funksjoner i spillet. Dette sørger for at ikke blir mange koplinger opp mot de andre klassene.

- GameController: Er selve "Controlleren"i programmet. Den bestemmer hvilke handlinger som skal skje ut i fra hva som skjer / hvilke inputs spillet får ved bruken av "PlayGame" metoden. Istedenfor å gjøre arbeidet med handlingene selv, så delegerer den det til de andre klassene.

Metoder

Vi navngir også metoder med standard C# konvensjon og metoder er derfor skrevet i PascalCase.

- CreateGame: Oppretter nytt brett (Board) og legger inn første element som @ (hodet) i slangen i posisjon 10 med gul farge. Til slutt kalles SpawnDollar for å legge inn en dollar.

- PlayGame: Det sjekkes om en knapp er trykket på tastaturet, og kaller metoder for hva som skal skje når disse trykkes. For eksempel når du trykker space blir pause true, og da kjøres continue for å hoppe tilbake til start i while løkka.
  Det sørges for at hver runde i spillet tar 100 ms ved å bruke variabelen timer av typen Stopwatch. Dersom det har gått under 100 ms siden sist timeren ble nullstilt hopper programmet tilbake i loopen med continue.
  Deretter blir MoveSnake kalt for å flytte slangen i nåværende retning.

- DollarHit: Kaller på metoden AddSnakeElement og SpawnDollar.

- EndGame: Bruker Environment.Exit, fordi den skal brukes for å avslutte konsoll app.

- MoveSnake: Flytter slangen i retningen som den har fått i parameter, ved å tegne ny @ for hodet og ny 0 for første element i kroppen, og fjerner siste elementet (halen) ved å skrive ut mellomrom " ". De første 3 rundene legges det til et ekstra element på slangen, siden slangen skal begynne med 4 elementer (inkludert hodet).

- SpawnDollar: Plasserer ny dollar i tilfeldig posisjon, men kjører igjen hvis den traff slangen.

- AddSnakeElement: Legger til et element bakerst i listen over koordinatene til slangen, for å gjøre slangen et element større.

- CheckIfSnakeHitElements: Sjekker om hodet til slangen har truffet noen av elementene i spillet; Hvis dollartegnet ble truffet kalles metoder for å gjøre slangen større og ny dollar opprettes.
  Hvis kantene av vinduet ble truffet eller den traff kroppen (seg selv), avsluttes spillet.

Variabler

Vi har brukt C# konvensjoner / standarder for navngiving av variabler. Vanlige variabler skrives i camelCase, men properties skrives i PascalCase.

Det var flere variabler vi ikke trengte som var i originalkoden, som ikke gjorde noe eller hvor vi har laget metoder for å gjøre funksjonaliteten isteden, som EndGame istedenfor å ha en variabel gameOver. Disse valgene gjør at det blir færre if setninger og koden blir mer lettlest og forståelig.

C# gir deg også mulighet til å deklarere variabler som "var", men vi velger å bruke den orginale variabeltypen for å gjøre det mer lettlest.

Vi endrer navnene på variablene til mer relevante ord som korresponderer med dens funksjon.

- snake → snakePosition
- head og newH → headPosition
- rng → randomGenerator
- app → dollarPosition

Vi fjerner variabler som er redundante og/eller ikke har noen verdi for programmet.

- OK, found, gg: fordi de ikke blir brukt
- gameOver og spawnDollar blir fjernet og gjøres om til metoder
- Vi fjerner while-loopen med spot variabelen fordi den bare legger ut dollartegnet en gang når spillet starter. Dette er helt unødvendig og vi tar bare innholdet av while-løkken og setter koden utenfor loopen.

Kilder

- Vi har brukt msdn for å se etter tilgjengelige features i C#: https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx
- C# Coding standarder: http://www.dofactory.com/reference/csharp-coding-standards
- Singleton løsning: http://csharpindepth.com/Articles/General/Singleton.aspx