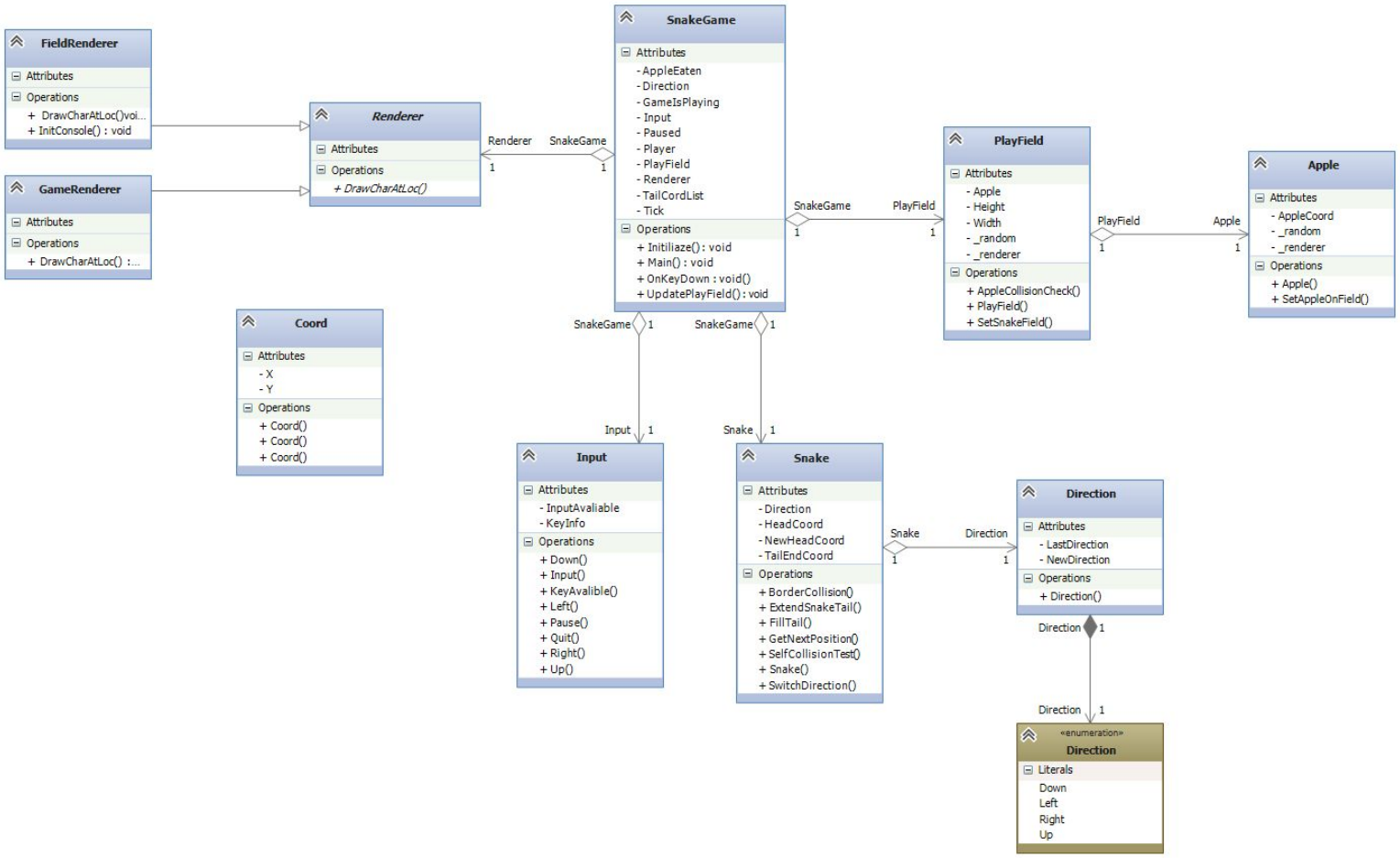
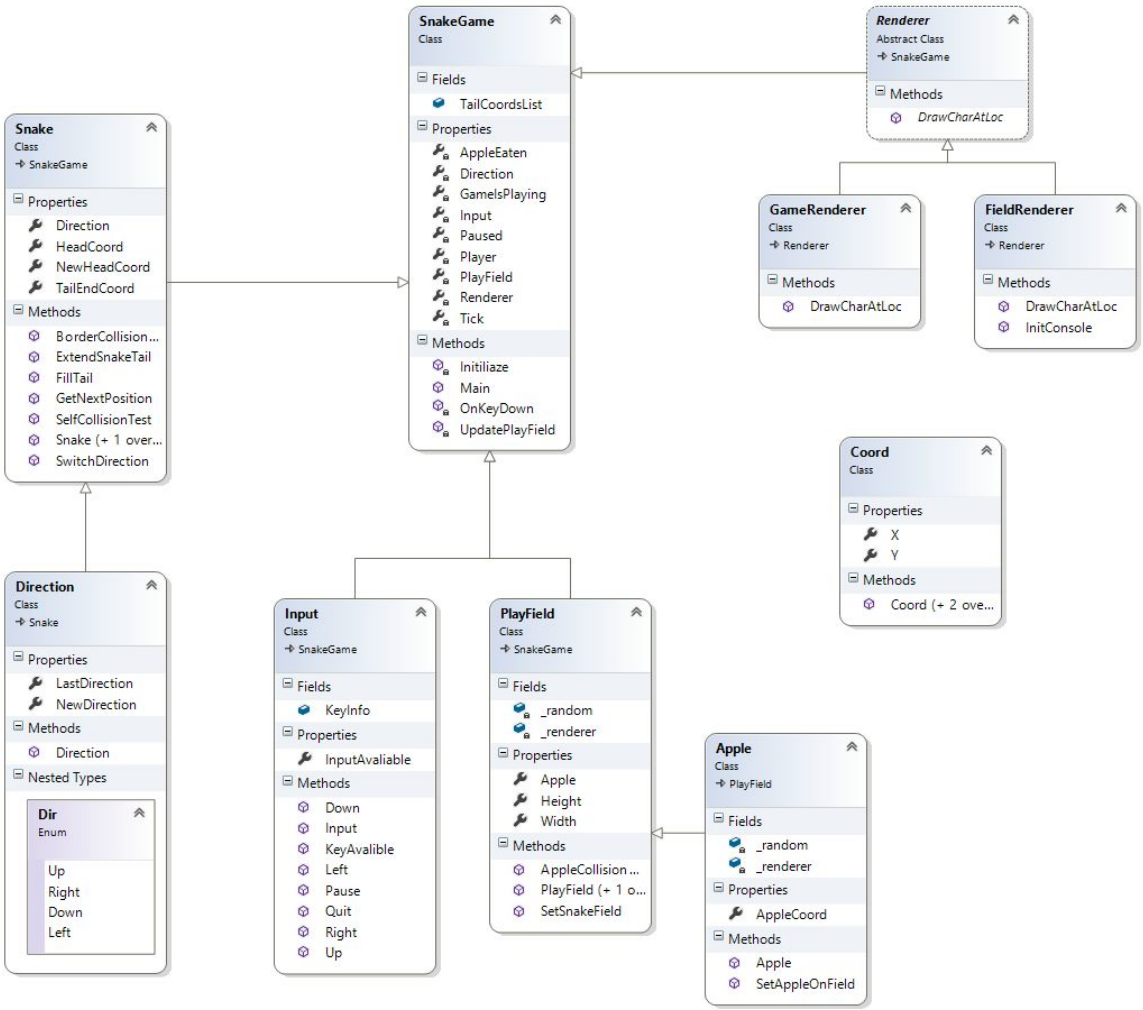
# Snake Refactor Dokumentasjon




Oppgave 1) a.
Dette er UML Class Diagrammet vårt for Snake Refactor oppgaven. Har også lagt ved en autogenerert class diagram fra ferdig refaktorering




*UML Class Diagram for SnakeMess*

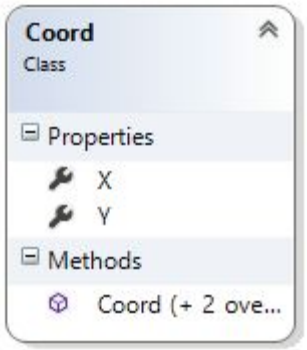
De forskjellige klassene:

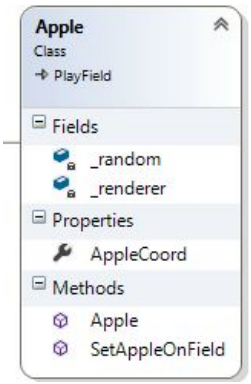
- Snake
- Direction
- SnakeGame
- Input
- PlayField
- Apple
- Renderer
- GameRenderer
- FieldRenderer
- Coord

Har latt være å kommentere koden, har istedet skrevet ganske detaljert hvordan ting henger sammen i dokumentasjonen for å holde koden mer ryddig og oversiktlig.

**Coord**

Denne klassen holder rede på koordinatene til “objekter” / “chars” på spille brettet, vi valgte å lage den slik fordi den ligner på det som var i den originale filen, og vi så ingen grunn til å bytte den ut, Det eneste vi endret på i denne klassen var å legge til en ekstra konstruktør Coord(int x, int y); Og selfølgelig å ta vekk unødvendig const string ok. Pluss at vi endret public int x og y, til C# properties.



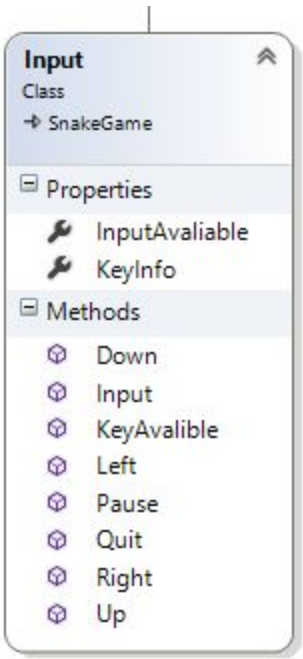

**Apple**

Apple klassen skal ta hånd om all logikk til eplet, først en Coord property for Apple Koordinatene, slik at andre klasser kan få ut hvor eplet befinner seg.
Private _random og privat _rendere, slik at vi kan legge eplet ut på brettet tilfeldig, og også kunne “rendre” det dvs i dette tilfellet skrive en char på console, men samme prinsipp som rendering.
Constructoren til apple klassen setter et nytt eple til plass 0,0. så har den en metode som heter SetAppleOnField, den gjør som metoden tilsier, at den setter et eple tilfeldig på en plass på det angitte brettet. derfor må den ha parameter på hvor stort brettet er, denne metoden er da mer gjenbrukbar fordi, hvis vi hadde hatt ett brett til med en annen størrelse måtte vi ha skrevet om hele programmet, dette er eksempel på low coupling slik at hvis vi f.eks skulle hatt med et annet nivå med større bredde og høyde eller mindre, må vi ikke gjøre noe annet enn å kalle på denne metoden med de nye verdiene vi har fra det brettet.
Metoden vil så finne en plass som ikke er på halen eller hodet til slangen så derfor må vi ha med en liste over alle koordinatene til slangen som vi sjekker opp mot den tilfeldige posisjonene vi genererte, og hvis det går galt vil den generere en ny random plass helt til den har funnet et sted å sette eplet. Når den har funnet en plass kaller vi på render objektet sin DrawCharAtLoc som vil tegne en $ på den ledige plassen. Apple klassen arver av PlayField fordi playfieldet er teoretisk den som eier et eple, fordi det eneste stedet et eple kan tilhøre er på play fieldet, derfor er det playfield som er creator av apple objektet fordi gir det mest mening at playfieldet skal ha kontroll over hvor eplet er. Det går under GRASP conseptet creator.

**Input**

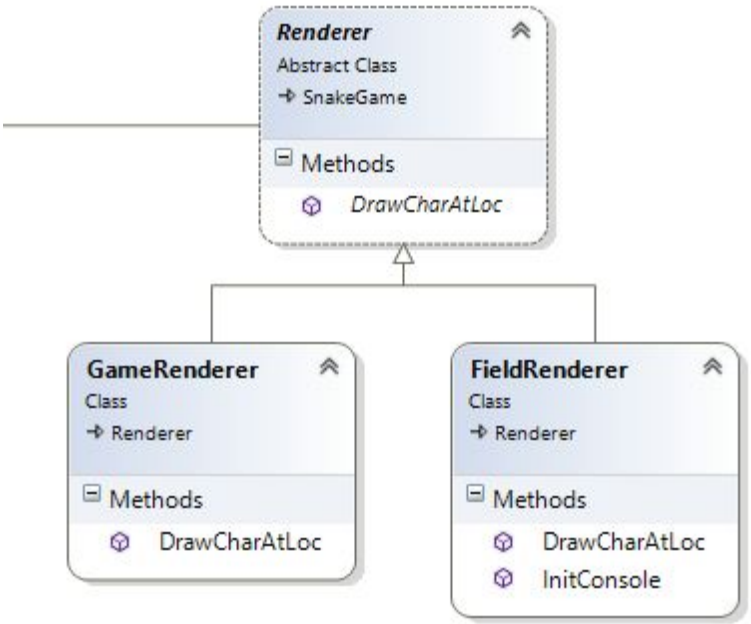
Input klassen holder info om de finnes en key tilgengelig ved hjelp av metoden KeyAvaliable så returnerer den true/false. Hvis det er en key tillgenelig så vil den sette av public property InputAvaliable til true, slik at andre klasser som skal bruke input, vet at det har skjedd et taste trykk. I konstruktøren vil vi sette ConsoleKeyInfo som holder info om hvilken key som ble trykket, til den samsvarende tasten som ble trykket. Når vi har all info vi trenger, kan creator klassene kalle på alle de forskjellige metodene de måtte ønske. F.eks så ser man med en gang at metoden Pause som vil pause spille har taste trykk “space”, og nå kan man bare endre denne ene == variablen i return linjen så har man satt en ny knapp til pause, f.eks på kan man sette den til “P” og man trenger ikke grave dypt ned i funksjonaliteten til koden for å endre “Key Bindings”, hvis man vil bygge enda mer på dette kan man lage en config fil som holder all slik informasjon slik at man enda lettere kan endre og lagre keybinds.

Det er litt mer komplisert for piltastene hvordan de returnerer hvilken retning slangen skal gå. Den vil se hvilken vei du går og prøver du å gå i direkte motsatt vei av det du allerede gjøre vil den ikke endre retning, men om du går i en 45 grader vinkel til en av sidene vil den la deg gjøre det. Denne klassen kan nesten ses på som pure fabricaton, grunnen til at jeg sier nesten er at den bruke Direction.Dir enum til å returnere hvilken retning den går i. Om jeg hadde laget den litt anderledes til å returnere et eller annet objekt som representerer en retning så ville det kunne brukes i langt flere tilfeller, og dermed oppnå mer Pure Fabriction, som vil si at det ikke representerer en direkte løsning til dette spesifikke prosjektet men at det mer en generell “service” som hånterer en veldig vanlige problemer.



**Renderer**

Renderer er en abstrakt klasse med en abstrakt metode som hetere DrawCharAtLoc siden vi prøvde å finne steder som hadde den samme metoden med forskjellig implementasjon var dette det ensete klare tegnet på at dette kunne gjøres om til en abstrakt klasse. Siden det var en bit av koden som tegnet en gul character og det var en annen som tegnet en grønn lagde vi 2 under klasser en som heter FieldRenderer som skal ta seg av alt som gjøres på brettet (console) og en GameRenderer som kanskje kan renames til SnakeRenderer, fordi det er det den hovedsaklig driver å rendrer, men siden man ofte skal legge på flere ting i spillet så må man sikkert lage flere metoder som kan gå under GameRenderer. En annen måte vi kunne gjort dette på er å ha color som parameter, men siden vi ville vise at vi kunne bruke abstrakte klasser og arv så gjorde vi det slik, noe som sto i oppgaven at vi skulle ta i bruk alt det vi hadde lært selv om det ikke ga altformye mening i en realistisk løsning. Og hvis man skal bygge på spillet er det mye bedre å ha det slik, fordi da kan du bare legge til en ny rendering metode hvis du f.eks skal ha en vegg eller andre ting som skal tegnes på brettet.




**Snake**

Ganske enorm klasse holder all logikken til snake og hvordan snake skal oppføre seg om den kolliderer med epler, halen eller kantene av brettet. Vi kunne sikkert ha delt denne klassen enda mer i f.eks en SnakeCollider, som holder styr på all kollisjonene som snake kan krasje i, men siden det ikke var så altfor mye å gjøre kollisjons sjekk på (3 metoder) så lot vi det være slik. Vi har 3 Koordinater som holder styr på posisjonen til hode, posisjonen til det nye hodet neste

update, som vil gå i retning til Direction, og koordinatene til slutten av halen. Slik at vi kan ta slutten av halen å sette til whitespace, og sette et nytt hode på NewHeadCoord plassen. Direction vil ha en enum Dir, som vil være 0 = Up, 1 = Right, 2 = Down og 3 = Left
Det første konstruktøren vil gjøre er å legge til 4 elementer i TailCoordList som holder koordinatene til halen, det legges til en et nytt element til halen for hver gang et eple blir spist. Vi tenkte kanskje å bruke dekorator her men siden det ikke lager altformye mening, i tillegg til at det kan bli vanskelig å implementere en decorator som legger på 0'er.

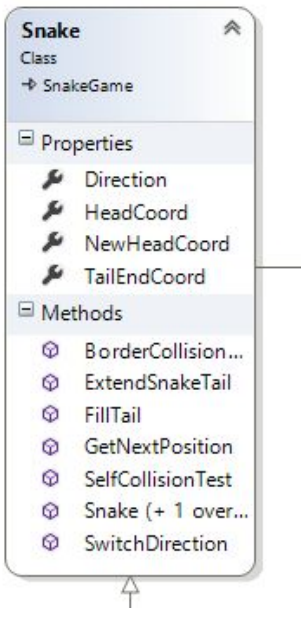
Metoden SwitchDirection(enum Dir newDir) skal iogforseg se etter hvilken retning slangen går, og da inkrementere / dekrementere de riktige koordinatene, som blir lagret som NewHeadCoord, med denne kan vi da før det skjer,  sjekke om slangen treffer en vegg eller et eple.

Denne klassen arver av snake game fordi det er snakegame som oppretter den og er Creator, alt som har å gjøre med snake er i SnakeGame.

BorderCollisionCheck(int height, int width) er metoden som sjekker om Slagen ikke har krashet i et av veggene eller at den er innen for de visse boundsene som brettet har, ikke < 0 og ikke > height / width returnere true/false og et simpelt kall på denne metoden vil være i SnakeGame noe som dette: GameIsPlaying = BorderCollisionCheck(width, height) hvor witdth og height må komme fra et PlayField objekt. denne metoden er lette å sette seg inn i og er derfor med på å skape High Cohesion.

AppleCollisionCheck, før jeg begynte med å skrive dokumentasjonen var AppleCollisioncheck i Snake klassen, men etter å ha sett hvor mange parameterverdier jeg kunne ta bort ved å sette den i PlayField, så satte jeg den der for bedre low coupling, det blir kanskje litt mindre high coheision fordi da vil collision metodene være spredt, men jeg gikk fra 6 parameter verdier til 2 så jeg syntes det var vert å flytte denne metoden.

SelfCollisionTest, sjekker først om et eple har blitt spist, dvs om AppleCollisionCheck har returnert true / false, hvis den er true vil den ikke sjekke om den har krashet i seg selv siden vi vet allerede at den krasjet i eplet, den kan ikke krasje i begge to samtidig. Gjenbrukbar og oversiktlig metode som ikke krever lang tid å sette seg inn i får å kunne forstå.

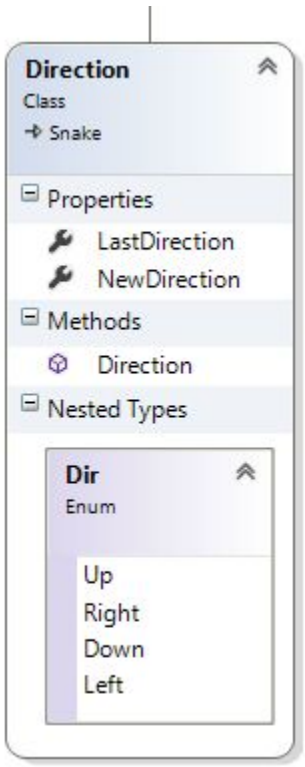



ExtendSnakeTail sier seg selv legger til et nytt element i listen som vil forlenge halen til slangen veldig gjenbrukbar og oversiktlig. Hører til der den skal.

**Direction**

Direction klassen har en enum dir som er Up, Right, Down og Left, disse blir brukt for å se hvilken retning slagen går. Vi må holde rede på to ting den nye veien slangen kommer til å gå hvis inputen endrer retning, og den gamle retningen den går slik at man forsetter å gå i samme retning om input ikke har endret den nye retningen, eller om input har gitt en pil i motsatt retning slik at den kan ignorere inputen. Denne direction klassen extender snake fordi det er bare snake som kan ha en retning og om det kommer f.eks andre objekter som kan ha en retning vil dette føre til at de også kan lage et retning objekt derfor er dette noe men kan igjenbruke om man skal f.eks ha en annen snake ai eller, epler som flytter seg. Den arver av snake fordi der er snake som er creator, selv om det ikke er helt nødvendig og bruke extends fordi det er ingen direkte overrides eller bruke av super klassen (base klassen i c#) vil det se mer ryddig ut. Hadde det vert flere klasser som brukte direction ville jeg nok ha gjort det på en annen måte å isteden gjøre som Coord, å bare ha en objekt klasse hvor de som trenger den kan bruke den.

Konstuktøren til dette objektet, lager en ny direction og setter den til down, så slangen vil alltid gå ned først.



**SnakeGame**
Til slutt er det snake game som utfører alt sammen fra de andre klassene, knytter de sammen og utfører selve spill logikken, det første som skjer er at main operetter et new SnakeGame objekt, deretter vil vi kalle på Initialize som starter spillet å får det til å kjøre helt til GameIsPlaying = false.
I initialize setter vi alle boolske variabler som vi trenger til de riktige verdiene. Vi lager en ny tailcoord liste objekt som skal holde på alle tail Coord objektene. Lager et nytt PlayField med navn PlayField, Lager en tick timer som skal være en stopwatch slik at uansett hastigheten på PCen din vil du ikke kunne gjøre flere oppdateringer per sekund enn andre folk men annen hardware, Setter også opp en ny renderer som skal tegne slangen på bretter etter vær update. Kaller så på Player = new Snake();
som vil initialisere ett nytt snake objekt som vil gjøre alt den den trenger får å kunne ha en Slange på banen.
Jeg har også laget et direction objekt fordi ellers må jeg gå via player.Direction.newDirection osv når jeg skal se hvilken vei slangen går i input logikken, og jeg syntes de lagde for mange connections, alternativt kunne jeg ha satt OnKeyDown metoden i Direction klassen eller i snake klassen men siden snake klassen var allerede veldig full og dette relaterte seg ikke mye til Direction klassen fant jeg ut at det så best ut å ha de i SnakeGame Klassen med et nytt Direction objekt som ble passert der hvor det trengs.

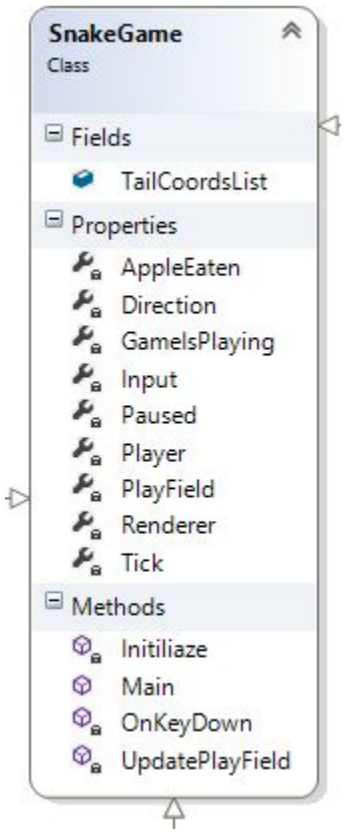
Før den uendelige game løkka så starter man stopwatchen Tick, og dermed går inn i while(GameIsPlaying) løkka. det første som skjer er input og det vil alltid sjekkes selv om spillet er pauset, for ellers kunne man ikke unpause. OnKeyDown brukte originalt hardkodede verdier 0 1 2 3 for direction men det ser mye bedre ut med en enum bak alt. Det gjør at andre kan lettere sette seg inn i det å forstå hva som foregår uten å måtte bruke masse kapasitet for å forstå hva newDirection = 3; betyr.

I selve game loopen vil den forsette å bevege seg så lenge !Pused som vil si så lenge man ikke har trykket på pause, vil den forsette å gå frem, trykker man på pause “space” så vil løkken bare sjekke om en ny tast har blir trykket, slik at man da ikke fryser spiller helt og ikke kan komme tilbake igjen.

Vi vil så se om det har gått nok tid siden sist oppdatering, har det ikke gått nok tid vil man ikke få gjort den neste oppdateringen og må vente til det har gått litt mer tid.

Den vil så hente ut posisjonene til playeren ved hjelp av metoden GetNExtPosition
Etter det vil vi se hvor slangen kommer til å ende oppe hvis vi flytter den i neste retning, da må vi utføre et par tester for å se hva som kommer til å skje videre i spillet, f.eks om spilleren har gått over brettet eller har krashet med seg selv. Vi har implementert et if(Conititon) continute; statement og det det gjør er å gå ut av game løkken hvis den evaluerer til true.

Har den kommet seg igjennom alle testene vil den rendre en 0 på brettet og hvis og utifra om et eple har blitt spist eller ikke vil den sette whitespace på tailenden, og så vil den legge til et ekstra element i tail listen, som den vil remove igjen et annet sted i koden om den ikke har spist et eple, så den vil bare bli lenger om den har spist et eple men kan da forstsatt flytte seg som en slaks stack “last in first out”.




**Generelle valg bak alt.**
Det først jeg prøve å gjøre var å lage massevis av klasser å så koble alle sammen ved hjelp av statiske objekter øverst i klassen jeg fant fort ut at dette ikke ville samsvare med low coupling prinsippene fordi det ble veldig mange koblinger og veldig rotete klasser. Jeg skrapte hele proskjetet å startet fra bunn av igjen hvor jeg fokuserte på å få ting som hørte sammen i sine engene klasser, som vil få high cohesion, etter det så jeg at det var mye enklere å gjøre koblinger mellom dem ved å ha en SnakeGame klasse som ikke gjorde noe annet enn å binde sammen alle de andre klassene slik at man kunne utføre spill logikken.
Alle objektene i SnakeGame er private som følger med i GRASP sitt Information Expert. Som betyr at de som ikke trenger å vite informasjonen skal ikke ha tilgang til den.

Vi har hatt fokus på å lage enkle metoder som er lett å sette seg inn i samtidig som å prøve å få de så gjennbrukbare som mulig. Det vil si de fleste metodene tar parameterverdier eller returnerer noe.