

**FORSIDE FOR INNLEVERING**

**TILGJENGELIGHET**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Fri** | **x** | **Begrenset** |

|  |  |  |
| --- | --- | --- |
| **Emnekode og emnenavn:** | PG3300 Programvarearkitektur | |
| **Tittel norsk:** | NITH Standard Forside for Innleveringer | |
| **Tittel engelsk (Hovedprosjekt):** | null | |
| **Eventuell oppdragsgiver:** | null | |
| **Utleveringsdato:** | 8. oktober 2012 | |
| **Innleveringsdato:** | 11. november 2012 | |
| **Antall sider:** | 9 | |
| **Antall ord:** | 2136 | |
| **Sammendrag (maks 100 ord):**  Dokumentet tar for seg en kort introduksjon av oppgaven og dokumentasjon av GUI, UI, og designvalg på kodenivå i innlevering 1 i emnet PG3300 Programvarearkitektur. Innholdet er delt inn etter deloppgaver. | | |
| **Gruppenummer:** | null | |
| **Studentnavn:** | **Studentnummer:** | **Signatur:** |
| Martin Lehmann | 700766 | null |
| Simen Bekkhus | 700532 | null |
| **Studentens signatur er også en bekreftelse av at hun/han har gjort seg kjent med, og fulgt, NITHs retningslinjer for intellektuell redelighet (tilgjengelig på intranett).** | | |

# Innholdsfortegnelse

[Innholdsfortegnelse 2](#_Toc340438777)

[Oppgave 1 a): UML 3](#_Toc340438778)

[Oppgave 1 b): Implementasjon i Visual C# 3](#_Toc340438779)

[Oppgave 1 c): Dokumentasjon av og begrunnelse for løsningen 3](#_Toc340438780)

[GUI 3](#_Toc340438781)

[UI 3](#_Toc340438782)

[Designvalg på kodenivå 3](#_Toc340438783)

[Beskrivelse av og begrunnelse for eksistensen til klasser 3](#_Toc340438784)

[Klasse-, metode-, og variabelnavn 4](#_Toc340438785)

[Objektorientering 5](#_Toc340438786)

[Arv 6](#_Toc340438787)

[Kjente «bugs» 6](#_Toc340438788)

[Oppgave 2 7](#_Toc340438789)

[GUI 7](#_Toc340438790)

[UI 7](#_Toc340438791)

[Designvalg 7](#_Toc340438792)

[Beskrivelse av klasser 7](#_Toc340438793)

[Objektorientering 8](#_Toc340438794)

[Låsing 8](#_Toc340438795)

[Problemer som har oppstått 9](#_Toc340438796)

Deloppgave 1 av 2: SnakeMess

# Oppgave 1 a): UML

Bilder ligger vedlagt.

# Oppgave 1 b): Implementasjon i Visual C#

Kode ligger vedlagt.

# Oppgave 1 c): Dokumentasjon av og begrunnelse for løsningen

## GUI

Alt av GUI var gitt i deloppgaven. Vi har gjort vårt ytterste for å gjenskape alt av visuelt innhold (inkludert at det blir generert et linjeskift hvis man «dør» helt nederst i høyre hjørne, selv om dette nok er å regne som en bug).

## UI

Brukeren kan benytte piltastene, som definert i oppgaven; WASD; eller NumPad 1 og 9 eller 3 og 7 for å navigere, tilsvarende det «originale» Snake-spillet. Brukeren kan også trykke Space eller Pause for å pause spillet, og Escape eller End for å lukke spillet. Dette er, som også spesifisert i beskrivelsen av InputHandler-klassen, den eneste funksjonaliteten vi har lagt på (som dermed ikke er å regne som refaktorering av koden).

## Designvalg på kodenivå

Her vil vi beskrive og begrunne eksistensen til de forskjellige klassene; begrunne klasse-, metode-, og variabelnavn; og forklare valgene vi har tatt rundt objektorientering og arv.

### Beskrivelse av og begrunnelse for eksistensen til klasser

#### Generelt

##### SnakeGame

En abstrakt base for selve spillet. Definerer vindusstørrelse og framerate. For å utvide programmet, må det skrives nye klasser som arver fra SnakeGame.

##### ConsoleSnakeGame

Arver fra SnakeGame, og tar seg av å sette opp spillet spesifikt for konsollen.

##### Program

Definerer hvilken plattform programmet skal kjøre på og starter «Game Loop»-en.

#### Spillobjekter (Game Objects)

##### DrawableGameObject

Alle tegnbare objekters mor; inneholder felles egenskaper for alle tegnbare objekter.

##### Snake

Selve slangen i form av en liste av alle kroppsdeler. Oppdaterer posisjonen til hver enkelt kroppsdel for å «flytte» slangen.

##### SnakeHead

Slangens hode. Har sin egen klasse i stedet for å være den samme klassen som SnakeBodyPart for å slippe å gjøre konsollspesifikke kall fra andre steder enn ConsoleView.

##### SnakeBodyPart

Hver enkelt av individuelle kroppsdeler. Har ansvar for posisjon, tekstur, og eventuell farge. Har sin egen klasse i stedet for å være en del av SnakeHead for å slippe å gjøre konsollspesifikke kall fra andre steder enn ConsoleView.

##### ConsoleBlank

Skriver over det bakerste elementet på slangen ved oppdatering for at slangen skal «bevege seg».

##### Loot

«Eplet» slangen skal «spise». Skrevet som en «singleton» for å optimalisere ytelse ved å flytte det samme objektet rundt i spillvinduet i stedet for å opprette et nytt objekt hver gang «eplet» blir «spist».

#### Arkitekturspesifikt

##### Model

Inneholder all data for spillet: i denne oppgaven (før eventuell utvidelse) Snake og Loot.

##### View

En abstrakt base for forskjellige typer visninger programmet kan ha. Før eventuell utvidelse er dette fullstendig meningsløst, men vi har valgt å løse det på denne måten for å gjøre programmet mer utvidbart, samt lettere å vedlikeholde. For å implementere flere visninger, må det skrives nye klasser som arver fra View.

##### ConsoleView

Arver fra View, og tar seg av den konsollspesifikke tegningen.

#### Verktøy (utilities)

##### InputHandler

Tar i mot og abstraherer faktisk input, og gir Input-klassen de abstraherte instruksjonene. Dette er det eneste punktet der vi har lagt på funksjonalitet i tillegg til å refaktorere, som kun består i å la brukeren kontrollere slangen med WASD og NumPad 1 og 9, eller 3 og 7, i tillegg til piltastene; å avslutte spillet med End i tillegg til Escape; og å pause/gjenoppta spillet med Pause i tillegg til Space. For å utvide programmet til andre plattformer, må det skrives nye metoder tilsvarende CheckConsoleInput() : bool.

##### Input

Kontrollerer slangen basert på de abstraherte instruksjonene fra InputHandler.

##### Point

Todimensjonalt punkt for å definere posisjonen til alle tegnbare objekter, og å definere spillvinduets størrelse.

### Klasse-, metode-, og variabelnavn

Vi har fulgt konvensjoner for klassenavn, som alle indikerer nøyaktig hva klassen er eller gjør, med unntak av Program-klassen; her har vi beholdt standard navn, og ser ikke dette som problematisk for kodens lesbarhet, da dette jo *er* standard navn på klassen som inneholder main-metoden. Eksempler er Point (som representerer et todimensjonalt punkt), DrawableGameObject (som representerer et abstrahert tegnbart objekt som er en del av spillet), og Snake (som representerer slangen i form av en liste av DrawableGameObjects).

Vi har fulgt konvensjoner for metodenavn, som alle indikerer hva metoden returnerer, eller gjør hvis void. Et eksempel er Loot.MoveLootToRandomAcceptablePosition() : void som flytter Loot-instansen til et annet godkjent punkt (ikke oppå slangen eller utenfor spillvinduet).

Vi har fulgt konvensjoner for variabelnavn (både fields, properties, og lokale variabler inne i metodescope), som alle indikerer hvilken verdi variabelen representerer. Eksempler er SnakeGame.Paused : Boolean, som representerer hvorvidt spillet er pauset, og DrawableGameObject.Position : Point, som representerer posisjonen til objektet.

### Objektorientering

Vi er klare over at løsningen vår er en ekstrem overdrivelse for dette lille prosjektet, men vi har løst oppgaven på denne måten for å vise kompetanse da dette er i skolesammenheng, og ikke til fulle vil reflektere en «reell» situasjon i bedrift.

Vi har valgt å gå relativt langt bort fra den opprinnelige arkitekturen vi modellerte i UML-diagrammet fra oppgave 1, dette i stor grad fordi vi har måttet benytte nødløsninger grunnet omstendigheter beskrevet under «Problemer som har oppstått» nederst i dokumentet. Dette er også grunnen til at vi ikke kan dokumentere alle valg som velreflekterte og gjennomtenkte, da mye har skjedd «on-the-fly», noe vi selvsagt må ta på egen kappe. Vi synes likevel at valgene vi har tatt har vært gode, og den grunnleggende arkitekturen fra den opprinnelige planen består, selv om vi har gått bort fra en ren MVC-implementasjon. Vi vil benytte deler av den muntlige presentasjonen til å presentere eventuelle forbedringer vi ville gjort, og håper på forståelse her.

Vi har valgt å benytte en MVC-inspirert arkitektur for programmet, der vi har én klasse kun tar seg av tegning og én som tar seg av inputhåndtering (View). Vi lar SnakeGame ta seg av oppdateringer av Snake-objektets posisjon og å lytte etter kollisjon mellom Snake-hodet og spillets ytterkanter og slangens egne kroppsdeler (selv om metoden for oppdatering ligger i Snake-klassen). SnakeGame lytter også etter kollisjon mellom slangens hode og Loot-instansen, og oppdaterer eventuelt Loots posisjon.

Vi har gjort programmet enkelt utvidbart til andre plattformer og/eller visninger på samme plattform gjennom å benytte konsolluavhengige (abstrakte) klasser (View, ConsoleSnakeGame) og metode (InputHandler.Input() : void), som tar i mot og prosesserer data fra og/eller sender instruksjoner til de konsollspesifikke klassene. På denne måten er «alt» en trenger å gjøre for å utvide programmet til andre konsoller å skrive én konsollspesifikk klasse for initialisering av spillet, én for tegning av objekter til den nye visningsformen, og én for håndtering av input, samt å endre eller legge til kall fra de konsolluavhengige klassene til de nyskrevne konsollspesifikke, heller enn å skrive om *alle* klassene som benytter egenskaper eller metoder i klassene som tar seg av initialisering, tegning, og inputhåndtering.

Vi har i all hovedsak benyttet designmønstre for å gi andre som skal jobbe videre med koden «least privilege»: Snake og Loot er begge Singletons; og vi har benyttet access modifiers for å gjøre all kode utilgjengelig der det ikke er helt nødvendig å få tak i den. Vi har også unngått å benytte nøkkelordet «var» for implisitte variabeldeklarasjoner der det ikke har hatt noen nytte, for å i hvert fall forsøke å tvinge de som eventuelt skal jobbe videre med koden til å beholde typene (viser til samtale under forelesning mandag 5. november).

Vi har, i tillegg til polymorfisme og singleton-mønsteret, aktivt benyttet designprinsippene Creator for å la riktige klasser opprette instanser av andre klasser, og High Cohesion for å oppnå Low Coupling. Vi har også i stor grad løst oppgaven med SOLID-prinsippene i tankene: hvert objekt kun har ett enkelt ansvar; entitetene har attributter, som gjør det mulig å endre dem uten å modifisere kildekoden; alle objekter som benyttes i andre entiteter kan trygt byttes ut med mer spesialiserte objekter som arver fra objektet som benyttes i utgangspunktet gjennom å utelukkende la arvende objekter utvide «parent»-objektet; og vi lener oss på abstraksjon heller enn konkretisering, som beskrevet over. Vi har, fordi vi ikke har implementert noen interfaces, ikke benyttet oss av «Interface segregation principle» i SOLID.

### Arv

Alle de konsollspesifikke klassene arver (naturlig nok) fra de konsolluavhengige. I tillegg til denne arven, har vi generalisert alle objekter som skal tegnes i den abstrakte klassen DrawableGameObject, som holder alle felles egenskaper for SnakeHead, SnakeBody, Loot, og den Console-spesifikke klassen ConsoleBlank (som i konsollen overskriver det bakerste tegnet – dette er ikke et problem en vil møte på i andre omgivelser). Dette gjør at vi kan benytte designmønsteret polymorfisme i Snake-klassen: slangen består av en liste med DrawableGameObjects, som i praksis er SnakeHead og SnakeBody med hver sin posisjon, tekstur, og farge.

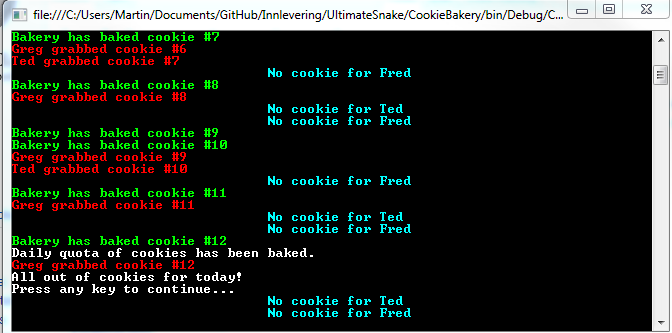
## Kjente «bugs»

Vi har ikke funnet noen bugs eller feil i programmet som ikke ble utbedret før levering.

Deloppgave 2 av 2: Cookie Bakery

# Oppgave 2

## GUI



Figur 1: Screenshot av konsollvinduet under kjøring av programmet

Vi har, som vist i figur 1, valgt å gå delvis bort fra det angitte designet av output ved å skille mellom «baking» og «grabbing» med farger, og å returnere utskrift også når «kunder» forsøker å grabbe en Cookie, men ikke får den fordi det ikke er noen cookies «up for grabs». Ut over dette

## UI

Dette programmet tilbyr ingen form for brukerinteraksjon.

## Designvalg

### Beskrivelse av klasser

#### Person

En enkel representasjon av en person, med navn som eneste egenskap.

#### Cookie

Privat innerklasse i Bakery. Eksisterer kun for å øke kodens lesbarhet.

#### Bakery

En helt statisk klasse bestående av et heltall som indikerer antall Cookies som skal bakes i løpet av en dag; en liste over bakte Cookies; og et heltall som indikerer antall Cookies som er blitt solgt i løpet av dagen. Klassen inneholder også den private innerklassen Cookie. Håndterer salg av Cookies, og sørger for at problemer rundt flere tråder ikke oppstår gjennom å «låse» lista hver gang denne skal oppdateres.

#### Program

Definerer ventetid mellom hver Cookie som blir bakt, og tiden hver person må vente før de kan forsøke å «grabbe» neste Cookie. Starter tre tråder som hver representerer én person, én tråd som representerer bakeriet, og tar seg av å holde hovedtråden opptatt under kjøring så det skal være mulig å la konsollen «leve» selv etter at programmet er ferdig med å eksekvere, gjennom å lytte etter tastetrykk.

### Objektorientering

Vi har valgt å holde koden enklest mulig i denne oppgaven, fordi dens formål er å vise kompetanse til å låse objekter.

### Låsing

#### SellCookieTo(Person customer) : void

Denne metoden må med nødvendighet låse lista før den gjør endringer til attributtene for å hindre at den samme Cookie-en blir «solgt flere ganger», altså at flere tråder gjennomfører sjekken for om det er flere Cookies i lista enn det har blitt solgt før den første som gjennomførte denne rekker å oppdatere variabelen som representerer antall solgte Cookies.

En alternativ løsning til liste over ferdige Cookies (som vil kreve mye RAM med *mange* cookies i lista) kan være å benytte en Stack eller Queue, og Push-e og Pop-e objektene og heller holde styr på hvor mange Cookies som har blitt bakt i sin egen heltallsvariabel.

#### BakeCookie

Vi har valgt å låse lista også her, fordi det i mellomrommet mellom at en Cookie blir lagt inn i lista og dette blir gjort kjent gjennom utskrift til konsollen kan hende at Cookie-en som akkurat ble laget allerede har blitt «grabbet», noe som vil føre til at utskriften vil indikere at Cookie-en ble solgt før den ble bakt. Dette vil (logisk sett) ikke være tilfelle, men ettersom å låse lista for å legge inn en ny Cookie og skrive ut resultatet har tilnærmet ingenting å si for programmets ytelse så lenge det er tidsbegrensninger på hvor ofte kundene kan «grabbe» Cookies, har vi valgt å gjennomføre dette for å forhindre forvirring. Uten tidsbegrensningene vil det ta omtrent tre (3) ganger lenger tid å kjøre gjennom programmet (testet med 5000 Cookies).

Generelt

# Problemer som har oppstått

Vi har, som diskutert, hatt problemer med at ett av tre gruppemedlemmer ikke har møtt «commitment»-ene sine i prosjektet, som har ført til at mye tid har blitt kastet bort, som igjen har ført til at vi for eksempel ikke har hatt tid til å lage noe domenemodelldiagram eller sekvensdiagram fordi problemene oppsto allerede helt i begynnelsen av prosjektet.

Vi burde selvsagt tatt tak i dette tidligere enn vi gjorde, men vi hadde heller ikke lyst til å kaste ham ut av gruppa. Vi håper på forståelse for dette, og at det vil bli mulig å gjøre opp for manglende diagrammer, samt eventuelle mangler i kode og/eller dokumentasjon på den muntlige presentasjonen av prosjektet.