Projektarbejde: Snake

Anna Ølgaard Nielsen
s144437

Christian Søholm Andersen
s103080

Mathias Enggrob Boon
s144484

Van Anh Tri Trinh
s144449

Introduktion til Softwareteknologi, 02121
Technical University of Denmark

Januar 2015

Contents

Chapter 1

Introduktion

Det kendte spil **Snake**, er et arkadespil, hvor spilleren bevæger en slange rundt på en bane og samler point for at forlænge slangen. Spillet er simpelt, men kan af denne grund have mange forskellige funktioner udover de sædvanlige, hvilket gør hver version ny og unik.

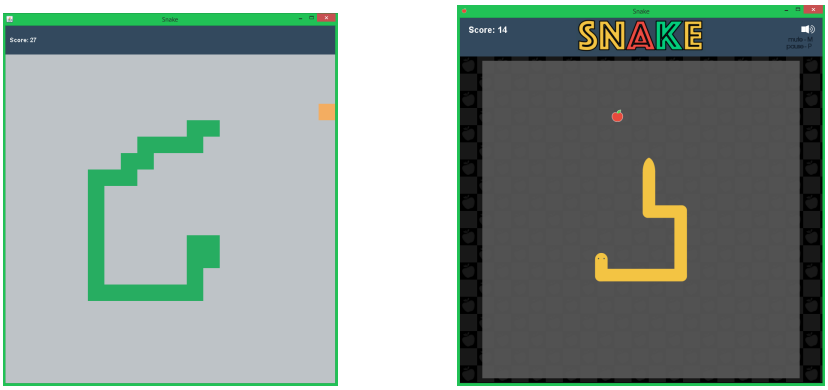
I denne rapport dokumenterer vi designet, implementeringen og evalueringen af vores egen version af Snake. Formålet med projektopgaven er at genskabe Snake som et Java-program, samt at dokumentere hvordan spillet er struktureret og udviklet, og hvorfor vi har valgt visse metoder og algoritmer. Spillet er gengivet i to versioner: **Simple Snake** og **Advanced Snake**.

Simple Snake er en primitiv version af spillet, hvor styring og bevægelse kun foregår vha. input fra spilleren. Spillet opbygges efter *Model-View-Controller*-designet, der gør det let at lave justeringer i spillet, når det senere skal udvides til Advanced Snake. Spillet er som navnet angiver simpelt og skal kun indeholde det mest essentielle for et snake-spil. Advanced Snake bygger videre på Simple Snake og bevarer derfor *Model-View-Controller*-designet, men med tilføjede funktioner, der gør spillet mere udfordrende. Slangen skal kunne bevæge sig af sig selv, brugergrænsefladen skal udvides til at vise andet en blot spil-scenen og control-delen skal udvides til ikke blot at tage tastatur-input, men også musseklik-input. Klasserne er holdt så separate som muligt for at undgå for meget sammenblanding og indbyrdes afhængighed, hvor der ikke skal være det.

I rapporten vil designet samt implementeringen af begge spillets versioner blive forklaret. Først gennemgås den simple version, Simple Snake, hvor projektet først afgrænses til de funktioner og egenskaber, som vi har valgt at fokusere på. Derefter bestemmes i design-delen de metoder, som vi synes er bedst til at løse vores problemer, hvilket letter implementeringen. I implementeringsdelen af rapporten, vil vi gå igennem implementeringen af Model- View- og Controller-delene hver for sig, og kort redegøre for deres sammenhæng. På samme måde gennemgås derefter Advanced Snake. I evaluerings-kapitlerne redegør vi for de tanker, valg og komplikationer som vi støder på undervejs.

Vi brugt github til at strukturere vores samarbejde. Vores repositorie med commit historie og eventuel videre udbygning kan findes her

`https://github.com/RandomInEqualities/Snake`



(a) Simpel Snake (b) Avanceret Snake

Figure 1.1: *Billede af Simple og Advanced Snake.*

Chapter 2

Simple Snake

2.1 Afgrænsning

Simple Snake er opbygget af en bane inddelt i felter og en slange dannet af felter. Spilleren styrer slangen, der ved spillets start består af to felter: et hoved og en hale. Spilleren skal ved hjælp af piletasterne styre slangen ind i et farvet felt på banen, der skal forestille at være mad. Det farvede felt forsvinder derefter, og slangen bliver et felt længere og får et point.

Banen Banen er opbygget som en todimensionel $n \times m$ torus, ved at bevægelse ud af den ene side, er en bevægelse ind fra den modsatte side. Banen er opdelt i felter (rækker, kolonner) som angiver positionerne på banen. Banen skal tilpasse sig vinduets størrelse som er justerbart, og det skal uden problemer være let at ændre størrelsen i programmets kode.

Slangen Slangen består af et antal sammenbundne felter. Den har et hoved og en hale der er de yderste felter af slangen. Slangens hoved har bevægelsesretningerne op, ned, højre og venstre. Gennem input fra tasteturet skal det være muligt at bevæge slangen rundt på banen. Dog må den ikke kunne bevæge sig i modsat retning i forhold til dens hoveds retning. På skærmen skal slangen visualiseres ved nogle firkanter som repræsenterer slangens felter.

Mad På banen skal der være et felt med mad, som slangen kan spise. Når dette sker bliver slangen et felt længere, maden forsvinder og dukker op i et andet felt på banen. Maden skal placeres tilfældigt på et af banens tomme felter. Når maden bliver spist, fås et point som opdateres i et tekstfelt.

Kollision Spillet slutter, når slangen bevæger sig ind i et felt udfyldt af slangen selv medmindre det er halen, som også flyttes, når resten af slangen flyttes. Kolliderer slangens hoved med dens egen krop, har spilleren tabt. Fylder spilleren hele banen ud med slangens krop, har han derimod vundet. Når spilleren enten taber eller vinder, skal spillet fryse, ved at tastatur-input ikke længere kan bevæge slangen.

2.2 Design

Snake-spillet er lavet efter et *Model-View-Controller*-design (MVC). Styringen, spil-logikken og den visuelle repræsentation holdes adskilt i tre dele. Med dette bygges et meget modulært program. Spil-logikken er *Model*, styringen er *Control* og den visuelle repræsentation er *View*. Disse tre dele skal kode-mæssigt holdes adskilt. Spil-logikken må ikke kende til styringen eller den visuelle repræsentation, mens den visuelle repræsentation godt må kende til spil-logikken, men ikke til styringen. Styringen må kende til både spil-logikken og den visuelle repræsentation (se Fig. 2.1).

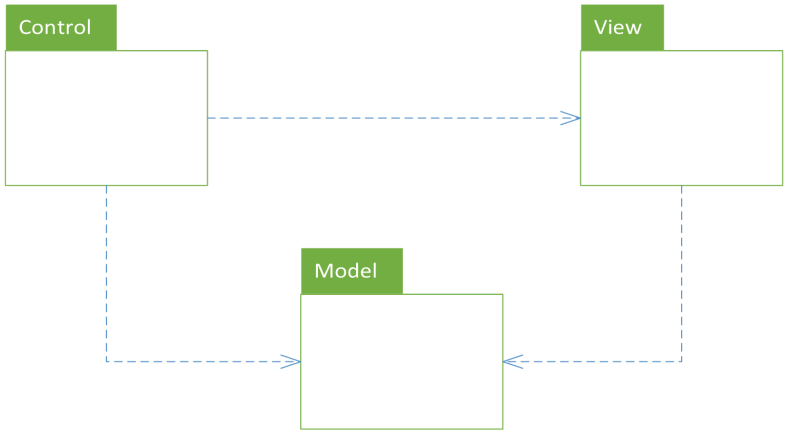


Figure 2.1: *Oversigt over Model-View-Controller-designet.*

På denne måde interagerer brugeren kun med den del af programmet, der er dedikeret til styring. Styringen manipulerer programmets tilstand i *model*-koden, som visualiseres i *view*-koden. Dette betyder derfor også, at alle funktioner, der påvirker programmets tilstand, skal holdes i *model*-koden. *Control* modtager kun input fra brugeren, og sender dette videre til *View* og *Model*. Det er muligt gennem en Observer at "observere" ændringerne i spillet, som derefter opdateres i *View* (se Appendix A for UML klassediagrammer).

Simple Snake er designet, så spillets logik ligger i klassen *Game* i model-pakken. *Game* får da spillets objekter fra andre klasser i *model*-pakken, f.eks. *Food* og *Snake*. Enumerations-klassen, *Direction*, forsimpler kommunikation med *Game*-klassen, idet det er nemmere at tilkoble en masse kommandoer og statements til hver retning, hvis der eksplicit er en *Direction*-klasse, som begrænser retningsmulighederne. Vi har valgt at gøre *Game* til en Observable klasse, da view-klasserne derved kan observere model-delen og opdatere brugergrænsefladen, så snart spillets tilstand ændrer sig. *Snake*-klassen afhænger af *Game*-klassen, da 'move'-metoden i *Snake* er nødt til at signalere til *Game*, at slangen har bevæget sig. Dette gør den ved at kalde på den 'snakeHasMoved'-metode.

I view-pakken ligger *View*, der er vinduet, som spilleren ser spillet igennem. Det er her *BoardPanel*-klassen, der viser den nuværende spil-tilstand. Denne er en Observer, som notificeres hver gang banens tilstand ændres i modellen. Når dette sker, eller når spillet startes, tegnes banen, slangen og maden fra *Game*-klassen, vha. klasserne *View* og *BoardPanel*. Scoren holdes i *Game*-klassen og vises i view-klassen *ScorePanel*. Dette panel er placeret over *BoardPanel* i

View-vinduet. Den observerer *Game*, og opdaterer sig selv, når scoren ændrer sig.

Control-klassen kalder relevante *Game*- og *Snake*-metoder når spilleren trykker på tastaturet. Klassen nedarver fra KeyListener og bestemmer hvornår og hvorhen slangen skal bevæge sig.

For at starte spillet bruges klassen *Driver*, som opretter et nyt *Game*-, *View* og *Control*-objekt.

2.3 Implementering

I Simple Snake er der ikke brug for nogen menu, og det er derfor muligt at holde styringen af spillet meget simpel. Ved at definere de fire piletaster i *Control*-klassen og en funktion der gør det muligt at bevæge sig på en bane som en torus, er alle styringskrav til Simple Snake opfyldt. Resten af spillet håndteres da i model-koden, mens view-koden gengiver spillet visuelt.

2.3.1 Model

Spillets model består af en række klasser, der tilsammen udgør spillets funktioner og objekter. Klassen *Game* er selve spillets logik.

Klassen *Field* bruges til at definere et punkt, som repræsenterer et felt på banen. Den fungerer på samme måde som *Point*-klassen, hvor koordinatsystemet starter oppe i venstre hjørne. Forskellen mellem *Point*- og *Field*-klassen, er hvordan man kommer frem til et punkt. *Point*-klassen går først ud ad x-aksen og derefter ned ad y-aksen, hvorimod *Field*-klassen gør det i omvendt rækkefølge. Da hele spillepladen er delt op i rækker og kolonner, frem for en x- og y-koordinater, er *Field* lettere at bruge for at undgå forvirring, da to-dimensionelle arrays også er opdelt i rækker og kolonner, hvor den første værdi repræsenterer rækker, mens den anden værdi repræsenterer kolonner. Vores *Field* har også den fordel, at man ikke dynamisk kan ændre rækker og kolonner. Disse værdier er som datatypen String, låst fast, og nye koordinater kan kun opnås, ved at oprette et nyt *Field*-objekt.

Field-klassen har fået defineret en 'equals'-metode, der bruges til at undersøge om to objekter ligger på samme felt, f.eks. slangens hoved og maden. Maden er et *Food*-objekt, som er defineret ved et *Field*-objekt, idet den fylder præcist ét felt. Denne bliver oprettet i klassen *Game*, hvor datafeltet 'position' afgør dens position. *Food* og *Field* har kun to værdier i sig - en række og en søjle. Som tidligere nævnt, kan disse ikke ændres efter at de er blevet oprettet. Dette gør, at de kan returneres i getter-metoder, hvor det sikres, at udefrakommende klasser ikke ændrer dem. Nogle gange kan det være vanskeligt at resonere omkring kode, hvis alle objekter vilkårligt kan ændre hinanden. Immutable klasser sikrer, at objekter ikke har ændret sig. Dette gør, at det er umuligt for control-pakken at snyde spil-logikken.

Banen er oprettet i *Game*-klassen, hvor der i konstruktøren også tjekkes om længden og højden af banen er af gyldige størrelse - dvs. værdier mellem 5 og 100. Her initialiseres spillerens score også, og der oprettes et Dimension-objekt, der skal repræsentere banen, et *Snake*-objekt og et *Food*-objekt. Hver klasse har de funktioner, der er relevante for dem, f.eks. *getPosition* i *Food*, der giver madens position i række/kolonne-koordinater. I *Game*-klassen er også oprettet enum-værdierne for State: RUNNING, WON og LOST. Disse bruges af view-delen til at afgøre, hvad der skal vises i vinduet.

Slangen selv er defineret ved klassen *Snake*. Slangens krop består af en række felter, hvoraf det første er hovedet, og de resterende er kroppen med halen til sidst. Koordinaterne for disse Fields er gemt som elementer i en ArrayList kaldet 'positions'. Det første element er slangens første led, hovedet, det andet element er slangens andet led osv.

Slangens flyttes ved at kalde dens 'move'-metode. Metoden signalerer internt til *Game*-objektet, at slangen har bevæget sig. Det gøres med metoden 'snakeHasMoved'. 'Move' er et enum, som betegner slangens handling: om slangen ikke kan bevæge sig, spiser maden, er død eller bevæger sig, henholdsvis EAT_NECK, EAT_FOOD, EAT_BODY og NORMAL. Først tjekkes, at slangen ikke bevæger sig ind i sin hals. Hvis det sker, signaleres EAT_NECK, og der returneres og ventes igen på nyt kontrol-input. Det undersøges derefter, om der er mad på samme felt som slangehovedets nye placering. Før slangen faktisk bevæger sig, beregnes altså det felt som spilleren flytter hovedet ind i, *når* slangen rykker sig. Dette gøres når 'move'-metoden kalder på 'findMovePosition'-metoden, der tager en Direction som input, og beregner den nye position ud fra hovedets nuværende position. Hvis den nye position ligger inde for banen, returnerer metoden blot et felt med de koordinater. Ligger et koordinat uden for banens grænser, returneres i stedet koordinatet på den modsatte side af banen. Dette beregnede felt kan nu sammenlignes med *Food*-objektets koordinater, og er disse ens, signaleres EAT_FOOD, et nyt hoved tilføjes i 'snake'-ArrayListens index 0 med hovedets nye koordinater, maden forsvinder, og der returneres, hvorefter der igen ventes på nyt kontrol-input. Hvis ikke maden har samme koordinater som hovedets nye felt, undersøges der, om den nye placering indeholder slangens krop. Dette gøres ved at undersøge om snake-ArrayListen - pånær det sidste element - indeholder et element der har de samme koordinater som hovedet. Hvis en af listens elementer har samme koordinater som hovedet, signaleres EAT_BODY. Hvis ikke, tilføjes et nyt hoved på det nye felt og det sidste element i ArrayListen fjernes. Dette giver en illusion af at slangen har rykket sig et felt frem.

Signalerne fra 'Move'-enum, kan nu bruges til at bestemme spillets tilstand. Dette gøres i *Game*-klassens 'snakeHasMoved'-metode, der sætter State til WON eller LOST, eller opretter nyt mad, hvis det gamle er spist. 'findFoodPosition'-metoden i *Game*-klassen sikrer, at maden altid placeres på et gyldigt felt, dvs. et felt der ikke er udfyldt af slangen. For at sikre dette, placeres maden først på et tilfældigt felt inden for banens rammer, hvorefter der undersøges om et af slangens led i 'snake'-ArrayListen har samme koordinater som madens felt. Er dette tilfældet, gives maden et nyt felt, indtil det lander på et felt uden slangens krop. Er slangen tilpas stor, er dette dog ikke effektivt, da der er stor sandsynlighed for at ramme et felt, der er optaget af slangen. Af denne årsag undersøger metoden først, om slangen fylder mere end halvdelen af banen. Er dette tilfældet, laves der i stedet en liste med alle tomme felter vha. en for-løkke, der løber gennem alle banens rækker og kolonner, hvorefter et tilfældigt element i listen vælges som madens position.

2.3.2 View

Brugergrænsefladen er samlet i view-klassen *View* der nedarver fra JFrame. I *View*s konstruktør oprettes et *ScorePanel*-objekt, der placeres øverst i vinduet, og et *BoardPanel*-objekt, der placeres under det i et BorderLayout. Board-panelet viser selve banen med slangen og maden, der begge er vist ved farvede firkanter, mens score-panelet, viser scoren.

ScorePanel-klassen nedarver fra JPanel og implementerer Observer. Klassen har en 'update'-metode, som signalerer at panelet skal optegnes pånyt, hver gang noget ændrer sig *Game*, da denne klasse har fået tilføjet score-panelet som Observer. 'paintComponent'-metoden benyttes til at tegne score-tekstfeltet. *BoardPanel* nedarver også fra *JPanel* og implementerer Observer, for at den også kan registrere ændringer i *Game*-klasse. Den består af en række 'draw'-metoder, samt en 'paintComponent'-metode, der kalder på 'draw'-metoderne for at tegne alle spillets komponenter på spillets bane og spillets bane selv. Disse tegnes med Graphics2D, der tegner et

rektangel pr. felt.

Da spillepladen skal være mellem 5×5 og 100×100 felter, kan det skabe problemer, hvis spilleren ikke kan justere størrelsen på vinduet eller banen. Vinduet kan være for stort til at passe på en gennemsnitlig computerskærm eller for lille til at gøre hele banen synlig. Det er derfor nødvendigt at gøre spillets vinduesstørrelse og selve bane- slange- og æble-størrelsen fleksibel. Derfor beregnes en passende feltstørrelse med metoden 'getWindowRectangle' i *BoardPanel*, der beregner den størst mulige længde på hvert led - afhængig af vinduets størrelse og antal felter på hvert led af banen. Når feltets bredde og højde er udregnet, kan denne rektangel nu bruges til at tegne banen, slangen og æblet. På Fig. 2.2 kan man se, at længden og højden af et felt tilpasser sig vinduesstørrelsen og udvider sig på det led, som vinduet strækkes på. Det kan derfor risikeres, at felterne og banen bliver aflange.

Figure 2.2: *Simple Snake med udstrakte felter*

2.3.3 Control

I Simple Snake er der ikke brug for nogen menu, og det er derfor muligt at holde styringen af spillet meget simpel. *Control*-klassen nedarver fra KeyAdapter og implementerer derfor metoden 'keyPressed', som kaldes hver gang, der tastes på tastaturet. Hvis tasten er en af de fire piletaster, kaldes metoden 'move' i *Snake*-klassen. For at *Control*-klassen registrerer tastatur-input, tilføjes den som keyListener til *View*-klassen.

2.4 Evaluering

Opbygning af banen

Først overvejede vi at designe banen som et to-dimensionelt array. Et [10][5]-array vil f.eks. give en bane med 10 rækker og 5 kolonner. Hvert element i arrayet bestemmer da, hvad der befinder sig på netop dennes plads. Elementerne i arrayet kan f.eks. være et blankt felt, mad, et led af slangen osv. Dette gør det nemt at introducere nye spilelementer i fremtiden, f.eks. bonus-point, vægge og miner, idet der blot skal tilføjes nye værdityper. Visualisering af spillet foregår ved at definere et billede for hvert spilelement, og få programmet til at tegne objektet på arrayets plads.

Vi valgte dog ikke denne metode, idet det to-dimensionelle array med banens størrelse skal sendes til 'paintComponent'-metoden, hver gang banen skal tegnes. For en bane på f.eks. 100×100 felter betyder det, at der i alt loopes over 10.000 elementer, hver gang banen skal tegnes - dvs. hver gang slangen bevæger sig. Vores anvendte metode sender derimod kun de relevante felter, dvs. madens og slangens position til view-koden. Denne behøver derfor ikke at tage hensyn til de andre felter, som er er tomme og uændret. En anden ulempe er, at det er mere kompliceret at bruge dette array. Hvis man f.eks. vil vide hvor slangen befinder sig, skal man loop igennem hele arrayet.

Slangens dele

Da slangen i snake-spillet består af en række felter, som alle har netop et koordinatpar, er en effektiv måde at gemme slangens position på en ArrayList, idet dennes datastruktur er fleksibel i størrelse og passer til formålet. Når slangen vokser i størrelse, bliver dette dog mindre effektivt, idet slangens hoved altid sættes som element 0, hvorved hele listen skal flyttes for at gøre plads. En anden mulighed ville være at bruge en anden datastruktur, f.eks. LinkedList, eller lade hovedet være defineret som elementet i index positions.size()-1 - altså sidst i listen. Et problem ved at bruge LinkedList er dog, at denne datastruktur ikke tillader vilkårlig adgang af værdier, men derimod altid bevæger sig fra første eller sidste element, indtil den når ind til det ønskede element. Dette problem kan løses ved at bruge en iterator, men blev fravalgt, idet det skabte problemer, når listen skulle bruges på tværs af klasserne. Alt i alt er det en mikro optimering at diskutere ArrayList vs LinkedList, da slangen sikkert ikke vil blive stor nok til at konsekvenserne af valget kan måles.

Score

Til implementationen af scoren blev to løsninger foreslået. Enten lades scoren være et datafelt i *Game*-klassen, ellers lades den være en klasse for sig selv. Ved at lade scoren være et datafelt, bliver implementationen simplere. At lade scoren være en klasse for sig selv har derimod fordelen, at der kan tilføjes en Observer til *Score*-klassen, som dermed kun opdateres, når scoren ændrer sig. Scorepanelet tegnes ikke i samme klasse som banen, og kan derfor holdes separat, så scorepanelet kun gentegnes, når scoren ændrer sig, mens banen med slangen gentegnes hver eneste gang slangen har bevæget sig. Hvis scoren derimod er et datafelt, tegnes scoren hver gang spillet opdateres og altså efter hver tur - også selvom scoren er uændret. Vi implementerede først scoren som en klasse, men ændrede dette senere da, det var for meget at operette en klasse, der kun holdte øje med en enkelt integer.

Tegning af spillet

For at visualisere spillet er det nødvendigt at kunne optegne banen, slangen og maden. Dette gøres i klassen *BoardPanel*, som har metoderne 'drawLevel', 'drawSnake' og 'drawFood', som hver især tegner deres tilsvarende element. Metoden 'paintComponent' kalder de førnævnte metoder, således at baggrunden tegnes først, så banen, så maden og til sidst slangen. Alting gentegnes hver gang spillet opdateres, hvorved der "tegnes over" det gamle billede. Det kan her argumenteres for, at baggrunden for eksempel ikke bør gentegnes, når slangen bevæger sig, ligesom store dele af banen forbliver uændret, og derfor ikke behøver gentegning. Alternativt kunne man lave et system, som undersøger hvilke områder der skal gentegnes, og hvilke der er uændret, og derfor ikke skal gentegnes. Dette kan dog i sidste ende betyde, at de kræfter der spares på tegning i stedet blot bruges på at afgøre, hvad der skal tegnes.

Skalering af banen

Gives hvert felt en absolut størrelse, ser spillepladen altid pæn ud, og kommer ikke til at afhænge af alle mulige vinduesforhold. Ulempen er bare, at alt efter hvor stor banen er, så kan vinduet blive for stort til computerskærmen, og hvis den er for lille, kan banen ikke forstørres. På denne måde er det en "for simpel" løsning bare at sætte faste værdier ind, for ikke at få en "grim" bane. Problemet bliver løst i Advanced Snake ved konstant at beregne den optimale feltstørrelse og dermed bevare forholdet mellem feltets længde og højde, mens resterende plads bliver udfyldt af baggrunden.

Brug af *Game*-klassen

Efter vi havde delt de fleste funktioner ud i klasserne, besluttede vi os for at samle nogle metoder i *Game*-klassen frem for deres oprindelige klasse, f.eks. *findFoodPosition*-metoden og banens oprettelse. Vi satte ikke 'move'-metoden ind i *Game*-klassen frem for *Snake*-klassen, da den ikke som sådan skal bruge informationer fra andre klasser. Slangen skal bare bevæge sig afhængig af *Control*-klassen. Logisk giver det også mening at slangen bevæger sig som en uafhængig enhed, og at det ikke er *Game* som flytter slangen. Det giver os dog den ekstra afhængighed at *Snake* er nødt til at kende til *Game*-klassen, så den ved hvor at maden ligger og hvor stor banen er.

Chapter 3

Advanced Snake

3.1 Afgrænsning

Den avancerede del af spillet er en udvidelse af det grundlæggende spil Simple Snake, hvor vi her har lagt fokus på brugervenlighed, valgfrihed og bedre struktur i programmets opbygning.

Multiplayer

To personer skal kunne spille mod hinanden i et lokalt multiplayer-spil. Den ene spiller bruger de samme tastatur-inputs som den oprindelige spiller bruger i singleplayer-spillet, mens den anden bruger tasterne W, A, S og D. Spillerne skal ikke blot bevæge sig på samme bane, men også kunne påvirke hinanden ved kollision.

Gameplay

Som typiske Snake-spil, skal slangen kunne bevæge sig af sig selv. Den skal kunne bevæge sig i periodiske intervaller, mens den med input fra tastaturet skal skifte retning eller sætte farten op i det moment der trykkes. Ved at implementere sværhedsgrader kan slangens hastighed ændres og tilpasses efter spillerens ønske. Desuden skal slangen accelerere ved at hastigheden forhøjes, når spilleren har samlet et vis antal æbler. Spilleren skal også have mulighed for at afbryde spillet og vende tilbage til hovedmenuen eller sætte spillet på pause vha. tastatur-inputs.

Menuer

Implementation af menuer, skal give spilleren adgang til forskellige scener i spillet. Når spillet startes, vises først hovedmenuen, som giver adgang til alle andre scener. I denne findes en 'singleplayer'-knap, en 'multiplayer'-knap, en 'controls'-knap og en 'quit'-knap. Trykkes på 'singleplayer'-knappen eller 'multiplayer'-knappen, tages spilleren til indstillingsmenuen, der giver spilleren mulighed for at indstille spillet efter personlig preference med hensyn til banens størrelse, slangens farve og sværhedsgraden. 'Controls'-knappen tager spilleren til en scene med vejledning til hvordan spillet spilles, mens 'quit'-knappen lukker vinduet.

Grafik og lyd

I stedet for 'paintComponent's tegnede rektangler, opgraderes grafikken ved at erstatte mad-rektanglet med et æble og lade slangen blive tegnet op af importerede billeder, der udover at vise hvilken ende der er hale og hvilken ende der er hoved, skal beskrive slangens form og retning, når den drejer. Derudover tilføjes en pause- og en game-over-skærm, der gør spil-tilstanden tydelig for spilleren og giver spilleren hurtig adgang til at spille igen eller vende tilbage til menuen. Når vinduet skaleres skal brugergrænsefladen også tilpasse sig den nye størrelse uden at miste sin form, sideforhold og placering af elementer i vinduet - samtidig med, at vinduets størrelse udnyttes optimalt. Udover grafik importeres baggrundsmusik og lydeffekter, som afspilles, når spillet starter, slutter og når slangen spiser et æble.

3.2 Design

3.2.1 Klasserelationer

Den avancerede versions design er også baseret på *Model-View-Controller*-konceptet, og indeholder derfor pakkerne, *model*, *view* og *control*, der hver især indeholder klasser tilhørende model, view og controller. Klasserne i control-pakken oprettes og tildeles i view-klasser, der giver kontrol over spillets progression og brugergrænsefladens udseende (se Appendix B for et relations klasse diagram). Dette bryder med MVC-konceptet men koden en del kortere. For eksempel operetter panel klasserne deres egne kontrollere. De interagere ikke med kontrollerene ud over at oprette dem og sørge for at de lytter til de rigtige komponenter.

I *Driver*-klassen, hvori main-metoden for programmet ligger, oprettes kun et *ViewFrame*-objekt, der nedarver fra JFrame, og derefter sætter vinduets synlighed til sand. I denne klasses konstruktør oprettes alle andre relevante klasser, som findes i view-pakken. Disse nedarver fra JPanel, og tilføjes og fjernes til og fra JFramen afhængig af hvor i programmet spilleren befinder sig. Er spilleren f.eks. i hovedmenuen, bruges *MenuPanel*-panelet, mens *HeaderMultiplayerPanel* og *BoardMultiplayerPanel* tilføjes, hvis spilleren er i multiplayer-spillet. På denne måde har hver scene i spillet en eller to tilhørende klasser der nedarver fra JPanel.

Spilleren navigerer vha. JButtons eller tastatur-input der modtages i control-klasserne. Control-klassen *ViewFrameListener* oprettes ligeledes i *ViewFrame*s konstruktør, hvorefter den som KeyListener tilføjes til alle panelerne. Denne klasses funktioner er globale, da den står for 'mute'- og 'return to menu'-funktionerne, der konstant skal være tilgængelige for spilleren uafhængig af de viste paneler.

Hvert panel opretter også i deres egne konstruktører deres tilhørende Listener fra control-pakken. Disse Listeners står for kontrollen, der er unik for deres panel. Alle panel-klasserne oprettes med *ViewFrame* som parameter, der derefter igen bruges som parameter, når panel-klassens control-klasse oprettes. Control-klassen kan derefter bruge *ViewFrame*, til at skifte paneler ud, når knapper eller taster trykkes. Specielt for *MenuListener*, der hører til hovedmenu-panelet, oprettes i konstruktøren også et *GameSingleplayer*- og et *GameMultiplayer*-objekt, da disse skal bruges, når der klikkes på 'Singleplayer'- eller 'Multiplayer'-knapperne.

GameSingleplayer og *GameMultiplayer* er underklasser til *Game*-klassen, og står for spil-delens oprettelse ved at bruge de andre objekt-klasser i model-pakken: *Board*, *Food* og *Snake*. Derudover findes der i model-pakken hjælpeklasserne: *Field*, *Direction*, *Event* og *Player*. *Game* nedarver fra Observable-klassen, der gør det muligt for view-klasserne at blive notificeret, når der foretages ændringer i spillet, og følgelig tilpasse sig ændringerne i brugergrænsefladen.

Al grafik, som ikke er fra Swing-biblioteket, er importeret i klassen *ResourceImages*, der giver enhver view-klasse adgang til at få fat i alle billeder. På samme måde er selv-definerede farver oprettet i klassen *ResourceColors*.

3.2.2 Multiplayer

Multiplayer ligner singleplayer på den måde at de begge foregår med slanger på en bane. Problemet med at integrere multiplayer ind i vores singleplayer-klasse er, at deres forskelle ligger meget spredt.

Singleplayer og multiplayer kunne dele omkring 90% af koden, mens de resterende 10% er flettet ind i de 90%. F.eks. har multiplayer et lidt anderledes score-system, idet der er to score-værdier. Det samme gælder for slangens farve, da der nu er to slanger og dermed to farver at vælge. Multiplayer-spillet har også et lidt anderledes gameplay, idet spillerne skal spille imod hinanden.

Vores mål for multiplayer-designet var forståelighed og genbrug af mest mulig kode fra singleplayer-spillet. Singleplayer- og multiplayer-spillene er lavet som to separate game-klasser, som arver fra en abstrakt game-klasse. Fordelen ved dette, er at vi har en adskillelse af spillene. Det som forgår i singleplayer-klassen har ingen indflydelse på multiplayer-klassen og omvendt. Med dette design er spillene nødt til også have hver deres header-panel, der holder på score-værdierne, og hver deres indstillingsmenuer og control-klasser. Som spillet, findes der altså for alle disse ting, en singleplayer- og en multiplayer-version, der begge nedarver fra en fælles superklasse. Underklasserne kan så frit override og/eller udvide implementationerne i deres egne klasser.

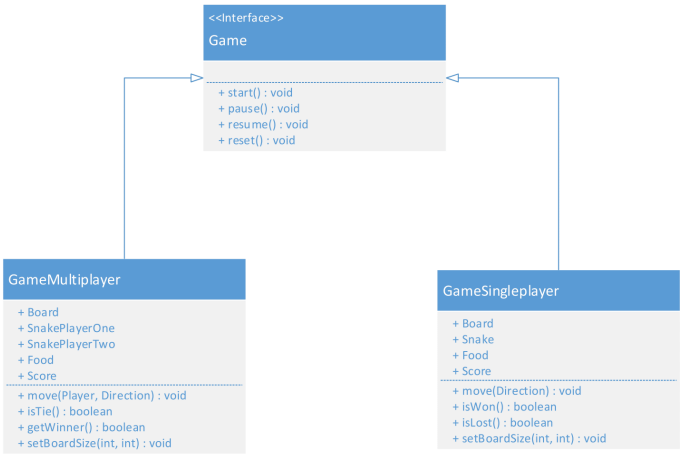


Figure 3.1: *Klassediagram over Game-klasserne.*

3.3 Implementering

3.3.1 Model

Ændret siden simpel: Timed bevægelse. Lokalt multiplayer-spil. Indstillingsmenu med valg af banestørrelse, farve og sværhedsgrad.

Modellen er nu lavet så det er muligt at vælge mere end én spiller. I både *GameSingleplayer* og *GameMultiplayer* findes en timer, som automatisk rykker slangen frem, hvis spilleren ikke gør det selv. Det er dog kun i Singleplayer-spillet at timerens opdateringsinterval formindskes efter hvert femte samlet æble. Slangen er stadig defineret som en ArrayList i klassen *Snake*, hvor bl.a. dens retning defineret ved en 'Direction'-enum er og en 'move'-metode.

Slangens bevægelse

I game-klasserne oprettes en eller to *Snake*-objekter og et *Board*-objekt med en given højde og bredde. Der oprettes også et Timer-objekt, der opdaterer slangens bevægelse med et vis tidsinterval imellem hver opdatering. Begge game-klasser implementerer ActionListeners, der bruges til at oprette Timer-objektet sammen med en konstant, der bestemmer opdateringsintervallet. Disse klasser skal nu også implementere 'actionPerformed'-metoden fra ActionListeneren. Heri kaldes game-klassernes 'move'-metode, der får slangen til at bevæge sig automatisk, når der er gået den tid, der er givet som opdateringsinterval.

Når en af piletasterne (Player 1) eller WASD (Player 2) trykkes ned, kaldes game-klassens 'move'-metode med en 'Direction' som parameter. Dette sker i control klasserne, *BoardSingleplayerListener* og *BoardMultiplayerListener*. Denne 'move'-metode tjekker om slangen spiser sig selv igennem dens metode 'isNeckDirection'. Returnerer 'isNeckDirection' sand, gås der ud af 'move'-metoden, og spillet kører videre i den samme retning. Returnerer den falsk fortsætter 'move'-metoden med metoden 'isBoardFull', der tjekker om det er den sidste frie plads i banen, som man bevæger sig ind i, ved at sammenligne banens størrelse og slangernes størrelse. I dette tilfælde har spilleren vundet.

Der oprettes et felt, 'newHeadPosition', ved at kalde på 'Snake'-klassens metode 'getNextHeadPosition', som tager 'direction' og 'board' som parametre. Her beregnes den nye række og kolonne for hovedet, hvis den bevæger sig i den givne retning. 'board' kalder på sin 'wrap'-metode, der tager den nye positions række og kolonne som parametre. Her tjekker den som i Simple Snake, om den givne række og kolonne ligger inden for banens størrelse. Hvis ikke, returnerer den et nyt Field med en række eller kolonne, der ligger i den modsatte del af banen. Ligger feltet inde på banen, returnerer den blot et nyt felt med de samme koordinater.

Der undersøges om det nye felt har samme koordinater som 'Food'-objektet. Denne boolean bruges i *Snake*-klassens egen 'move'-metode. I denne metode tjekkes først om den ny-beregnede hovedposition har samme koordinater som en del af kroppen. I dette tilfælde returnerer metoden sand, timeren stopper og spilleren har tabt. Ellers fjernes det sidste element i slangens ArrayListe (halen), hvis slangen *ikke* spiser et *Food*-objekt, mens den tilføjer et nyt hoved på index 0 uanset om den spiser *Food*-objektet eller ej.

Event system

Vi har oprettet et event system til model klasserne. *GameSingleplayer* og *GameMultiplayer* nedarver fra *Observable*. *Observable* kan i *Observer* 'update' metoderne sende et ekstra object med sig. Vi har lavet en *Event* klasse som indeholder alt det som der kan ske i spillet. Når for eksemple slangen spiser, så sendes et *Event* object af type *EAT* med som det andet argument i 'update' metoderne. Det kan vi bruge til at implementere lyde i view pakken.

Multiplayer

Specielt for multiplayer-spillet tjekkes der også, om den ene spiller støder ind i den anden. Denne har et *Player*-objekt, som definerer vinderen, der enten er ONE, TWO eller NONE. I 'move'-metoden her er tilføjet et if-statement, der undersøger om 'newHeadPosition' har samme koordinater som et af modstanderens positioner. Hvis dette er sandt og det ikke er modstanderens hoved, sættes 'winner' til modstanderen, mens 'winner' sættes til 'NONE', hvis det *er* hovedet. Alt efter hvilken spiller der vinder, printer spillet nu ved spillets ende en tekst der fortæller hvem vinderen er og hvor mange æbler denne har samlet.

3.3.2 View

Ændret siden simpel: Tilføjede scener. Brug af importerede billeder. Optegning af slange efter bevægelse og farve. Ny algoritme for justering af banestørrelsen.

Når *View*-klassen konstrueres, opretter den som tidligere nævnt panel-objekterne fra samme pakke. Derudover opretter den også view-klassen *Audio*, der står for spillets lyde, som spilles efter slangens handlinger i spilscenen, hvis klassens boolean felt 'muted' er falsk. Disse importeres som 'Clip'-objekter.

Når spilvinduet åbnes, kalder *ViewFrame* først på metoden 'showMenu', der kalder på metoden 'setFrameComponents', som bruges hver gang vinduet skal skifte paneler ud. Metoden fjerner først alle sine komponenter, tager to nye komponenter som parametre, og placerer den førstnævnte øverst i sit BorderLayout, og det andet komponent i midten - dvs. nedenunder. Derudover bruges metoden 'requestFocus', til at give center-panelet fokus. Dette er nødvendigt hvis man vha. en JButton skifter scene, idet panelet ved denne handling mister fokus og derved ikke opfatter tastatur-input. Når 'showMenu' kaldes, sættes *HeaderBasePanel*-objektet som det øverste panel, mens *MenuPanel* er panelet under. *HeaderBasePanel* viser en topbar, som indeholder spillets titel, tastatur-genveje og en JButton, som viser om lyden er slået til eller fra og som også kan bruges til at slå lyden til eller fra. Klasserne *HeaderSingleplayerPanel* og *HeaderMultiplayerPanel* nedarver fra denne klasse, og tilføjer i deres 'paintComponent'-metoder, optegning af spillerens/spillernes score, som de får fra game-klasserne. *ViewMenu*-panelet indeholder JButtons som giver spilleren adang til spillets scener: Singleplayer, Multiplayer og Controls. Den sidstnævnte fører til hjælpescenen *ControlsPanel*, der blot viser et billede med vejledning til spillet. Desuden har den også en *Quit*-knap, som fungerer som vinduets kryds i hjørnet. Når denne trykkes kaldes på *ViewFrame*s 'closeWindow'-metode, der starter WindowEventet, som lukker vinduet.

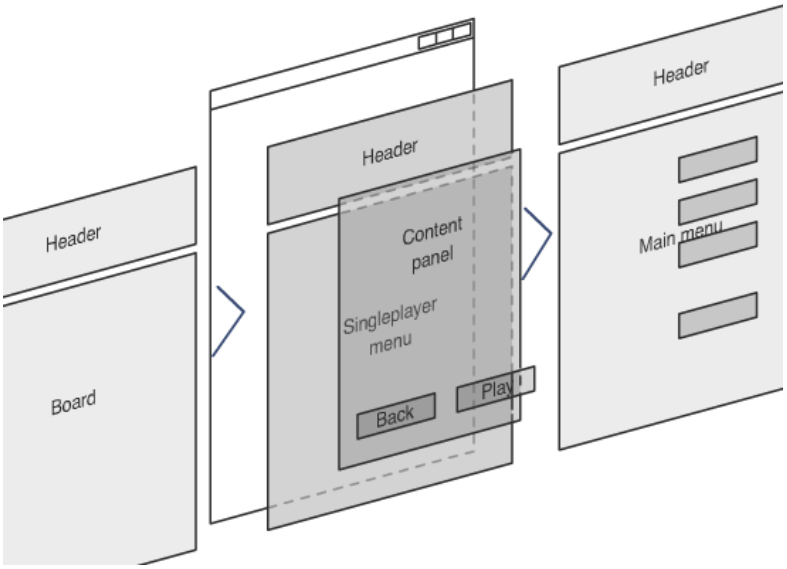


Figure 3.2: *'setFrameComponent's udskiftning af paneler.*

Vinduesskalering

Da *ViewFrame* bruger BorderLayout, er det simpelt at placere et header-panel øverst, og det ønskede scene-panel nedenunder. Alle andre view-klasser er efterladt med standard-layoutet *FlowLayout*, da deres komponenter skal placeres i specifikke koordinater. Dette gøres med *setLocation* der definerer komponentets præcise placering ved x- og y-koordinater. Når spilleren ændrer vinduets størrelse, er deres koordinater dog ikke altid bevaret, hvis komponenterne f.eks. altid skal ligge i midten af vinduet, eftersom punktet (0,0) ændres, idet den følger vinduets venstre hjørne. Disse kald på *setLocation*-metoden er derfor placeret i panelernes *paintComponent*-metode, da denne køres hver gang vinduet skaleres. Koordinaterne beregnes derved pånyhver gang vinduet skaleres, og komponenterne kan derfor altid få de rigtige koordinater. Denne metode er brugt til alle billeder, JComponents og al tekst der oprettes eller tegnes.

I klassen *OptionsBasePanel* tegnes grafikken til spil-delen. Her er lagt specielt vægt på, at felternes størrelse passer til spillerens ønskede bane-størrelse og spillerens ønskede vinduesstørrelse samtidig med at bevare dens kvadrate form, hvilket jo ikke var tilfældet i den simple version af spillet. Metoden 'getFieldSideLength' beregner ud fra disse to størrelser den størst mulige længde og højde af et enkelt felt og returnerer derefter den mindste af de to værdier. På denne måde kan feltet være kvadratisk og passer med sikkerhed på begge led af vinduet. Denne 'getFieldSideLength'-metode kan nu bruges til at bestemme størrelsen af og tegne banen, slangen og æblet, så de passer til vinduets størrelse. Da disse tegnemetoder bliver kaldt fra 'paintComponent'-metoden, der køres igennem konstant under spillet og når vinduesstørelsen ændres, kan spilleren udvide vinduet for at se spillet i en større version, der uafhængigt af vinduesskaleringen, skaleres lige meget på begge led.

Tegning af slangen

Udover bane-størrelsen er det også muligt at vælge slangens farve. Når en farve er valgt tjekkes der i 'drawSnake'-metoden i *OptionsBasePanel*-klassen om farven har været valgt før. Hvis ikke, tilføjes den nye farve til ArrayListen 'snakeColors' i metoden 'addSnakeColor'. Denne metode farver også alle billederne, som viser slangens dele og tilføjer dem til delens egen ArrayListe. F.eks. er 'snakeHeadUp' en ArrayListe, som indeholder alle de brugte farveversioner af 'snakeHeadUp'-billedet. Farvningen af billederne sker igennem 'colorSnakeImage'-metoden, der tager et BufferedImage- og et Color-objekt som parametre. Her oprettes der et WritableRaster-objekt, der gør det muligt at manipulere med et billedes pixels. Dette gøres ved at kopiere billedet med BufferedImages 'copyData'-metode, og parameteren null, hvilket opretter en kopi af billedets areal som en rektangel og ind i et passende WritableRaster-objekt. I for-løkken undersøges om den valgte pixel har farven på øjnene, da den så ikke skal farves. Dette gøres med billedets 'getRGB'-metode i et if-statement og den fundne farvekode for øjnene. Farvekoden er bestemt ved at printe alle farvekoderne for et billede af slangens hoved, og derefter finde den farvekode, som forekommer mest sjældent, idet dette må være øjnenes farve. I if-statementet farves slangen ved at der oprettes et array af heltal, som skal holde på R-, G- og B-værdierne for den valgte pixel. Denne defineres ved at bruge 'getPixel'-metoden på WritableRaster-objektet, der giver et heltals-array med størrelsen tre, der nu kan tildeles den nye farves RGB-værdier og derefter bruges til at farve den valgte pixel, med metoden 'setPixel', der tager imod koordinaterne for den valgte pixel og farve-arrayet. Endeligt, returner 'colorSnakeImage'-metoden et nyt BufferedImage, der nu er farvet og kan tilføjes til kropsdelens ArrayList.

Har den valgte farve været valgt tidligere, bestemmes dens index i 'snakeColors'-ArrayListen, som kan bruges til at hente alle de rigtige farveversioner af slangens dele. På denne måde behøver billederne ikke at blive farvet en bestemt farve mere end én gang. Efter at have fundet billederne for slangen i den rigtige farve, skaleres alle billederne, så de passer til bane-størrelsen.

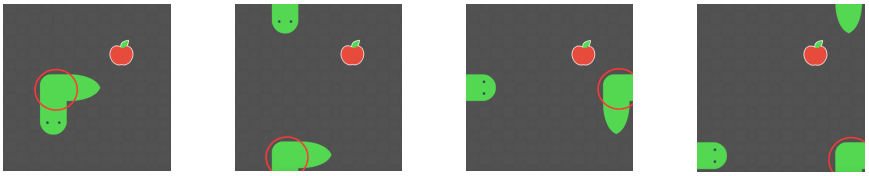
I Simple Snake kan spilleren ud fra slangens krop, se hvor han har været, men ikke hvilken vej han har bevæget sig, idet de udfyldte felter er fyldt helt ud til kanten. Slangens udseende forbedres derfor ved at vise slangens bevægelsesretning og retningsskift i hver enkel del af dens krop og generelt erstatte alle firkanterne, med mere beskrivende billeder. Dette giver seks mulige udseender for hver enkel del af slangens krop. Hovedet og halen, findes hver i fire versioner afhængigt af retningen, som spilleren vælger for hovedet, eller retningen af kropsdelen lige før halen. Kropsstykkerne imellem er dog ikke kun afhængig af retningen af stykket lige før eller lige efter, men begge dele. Den vandrette del og den lodrette del af slangen bestemmes let ved at undersøge om stykket, der skal tegnes ligger i samme række eller kolonne som stykkerne før og efter. Slangens hjørnestykker bestemmes på en mere indviklet måde, da stykket og dens tilgrænsende stykker aldrig ligger i samme række eller kolonne, men derimod kan ligge i fire forskellige forhold til hinanden. I figur 3.3a ligger de tilgrænsende felter lige under og til højre for hjørnet, men dette gælder f.eks. ikke for figur 3.3c hvor det ene stykke ligger lige under, mens det andet stykke ligger til venstre for hjørnet uden at grænse op til dens venstre side, der ellers ville give hjørnestykket spejlvendt i y-aksen. Da felterne for kroppen ligger i en ArrayList sorteret efter slangens opnåede dele, sammenlignes et felt let med feltet før og efter det i listen. Da stykket foran og bagved uden påvirkning på hjørnestykket kan bytte plads, findes der altså otte situationer for et enkelt hjørnestykke. I alt tjekkes der derfor - for kroppen alene - 34 mulige forhold mellem et stykke og dens stykke foran og bagved.

I *BoardBasePanel*-klassen blev alle de mulige situationer hvorpå slangen kan se ud undersøgt. Passede den valgte kropsdels forhold til delen foran og bagved med et if-statement, blev kropsdelens billede skiftet ud med det passende billede og derefter tegnet for den del. Det har dog været meget langt, da et if-statement for ét billede af et hjørne alene tog flere linjer, men koden kunne simplificeres ved at finde et mønster i de fire if-statements. 'isSnakeCorner'-metoden returnerer en boolean 'isCorner', som fortæller om tre *Field*-objekter udgør et specifikt hjørne. Et if-statement før indeholdte alle situationer opdelt med ||-symbolet, men er nu erstattet af fire korte if/else statements med 12 forskellige variable, der gør det muligt kun at kalde én boolean. Programmet kører ikke alle if-else-statements igennem hver gang den skal lave et hjørne, men kun det hjørne som bliver kaldt. Variablerne kan enten have værdien 0, 1, -1, 'lastColumn' eller 'lastRow'.

De første to rækker kode af 'isCorner'-metoden, bruges når slangen er placeret midt på banen (se figur 3.3a), og derved grænser op til sin foran- og bagvedliggende del (front og back). Disse dele findes enten ved getColumn()+1 eller getColumn()-1 og på samme måde getRow()+1 eller getRow()-1. De to næste lijner kode i *isCorner* beskriver, slangens placering i bunden eller toppen af banen, hvor den går gennem torussen, så noget af den ender på den modsatte side af banen (se figur 3.3b). De to næste linjer kode er på samme måde slangens placering i en af siderne, når den går gennem torussen (se figur 3.3c), hvorefter noget af den ender på den modsatte side. De to sidste linjer kode er til, slangens bevægelse igennem torussen to steder i et af hjørnerne. Altså hvis slangen f.eks. er oppe i højre hjørne, går gennem torussen ved at gå opad, og straks til højre gennem torussen igen (se figur 3.3d). Derved er alle hjørne-situationer gennemgået. Man kan se et mere detaljeret billede af kroppens dele og de fire situationer i Appendix B, *Slangens Opbygning*.

Figure 3.3

(a) Corner 1 (b) Corner 2 (c) Corner 3 (d) Corner 4



Når spilleren færdiggør spillet enten ved at tabe eller vinde, tegnes Game-Over-skærmen ved en gennemsigtig rektangel, tekst og JButtons, der giver mulighed for at gå tilbage til menuen eller spille igen.

3.3.3 Control

Ændret siden simpel: Mute. Pause. Return. Return to Menu. Start/Play Again. JButtons.

For at give spilleren valgfrihed er der til mange funktioner både implementeret en JButton og en tilhørende genvej gennem tastaturet. Fra tastaturet er der blevet implementeret genvejene: (M) - mute, (P) - pause, (Esc) - return to menu, (Backspace) - return og (Enter/Space) - start / play again. Control-klasserne nedarver fra KeyAdapter-klassen, der registrerer tastatur-input. Samtidig implementerer de ActionListener-klassen, der registrerer tryk på JButtons. Disse control-objekter oprettes i deres tilsvarende view-klasse og tilføjer KeyListeners dertil og ActionListeners

til hver enkel knap, som de har funktioner for. Individuel kontrol over de forskellige knapper fås ved at sætte en unik ActionCommand til hver enkel knap, der derefter kan tjekkes for i ActionListeneren abstrakte metode 'actionPerformed'.

ViewFrameListener-klassen er som nævnt en pseudo global KeyListener, der står for kontrollen overalt i spillet. Dvs. muligheden for at slå lyden til eller fra eller vende tilbage til menuen. Når enten (M) eller lydknappen trykkes på, sættes *Audio*-klassens boolean 'mute' til det modsatte af det den i forvejen var, mens *HeaderBasePanel* (eller dens underklasser) notificeres for at opdatere lyd-ikonet.

Figure 3.4: *Lydikon, on/off.*

Control-klasserne *BoardSingleplayerListener* og *BoardMultiplayerListener* indeholder metoden 'actionPerformed', der genstarter eller går ud af spillet når spilleren trykker på 'Play Again'-knappen eller 'Menu'-knappen. Klasserne indeholder også KeyEvents, som kalder på en 'move'-metode i *GameSingleplayer* eller *GameMultiplayer*, der får slangen til at bevæge sig, når spilleren trykker på tasterne, som styrer slangen. Klasserne bruger P, der stopper eller starter spillet ved at stoppe timeren i game-klasserne, og ENTER/SPACE, der fungerer som 'Play Again' knappen.

Spil-indstillinger

OptionsListener, der er en abstrakt klasse implementeret af *OptionsSingleplayerListener* og *OptionsMultiplayerListener*, styrer knapperne i indstillings-scenerne. Før spillets start kan spilleren som tidligere nævnt - udover at vælge farve og sværhedsgrad - vælge banestørrelsen defineret med 'width' og 'height', der beskriver helholdsvis antal felter pr. række og antal felter pr. kolonne. Denne funktion er indbygget vha. to JFormattedTextFields, der har fået et Formatter-objekt, som begrænser inputtet til højst et tre-cifret tal. Dette begrænser spillerens mulighed for at indtaste en ugyldig størrelse. Da tekstfelterenes Formatter får deres caret til at sætte sig i starten af tekstfeltet, selvom spilleren trykker et andet sted i feltet, implementerer *OptionsBasePanel*-klassen en FocusListener, der har de abstrakte metoder 'focusGained' og 'focusLost'. I 'focusGained' bruges SwingUtilities 'invokeLater'-metode, der sørger for at opgaver ikke udføres samtidig så brugergrænsefladen kan opdateres korrekt, og tekstfelternes caret kan placeres det rigtige sted når felterne får fokus.

Trykkes på en af farveknapperne eller sværhedsgradsknapperne, optegnes den vha. JButton-metoden 'setBorder' med en tyk kant, for at vise at den er aktiv, mens kanterne på de andre knapper fjernes med 'setBorderPainted(false)'. Trykkes på en farveknap, sendes farven til *BoardSingleplayerPanels* eller *BoardMultiplayerPanels* 'setSnakeColor'-metode, der derefter bruger farven som beskrevet tidligere. Trykkes på en sværhedsgradsknap, sættes en 'Difficulty' enum i *OptionsListener*-klassen til den valgte sværhedsgrad, hvorefter denne bruges når 'Play'-knappen trykkes på igennem dens underklasser, eller når der trykkes (ENTER). 'Play'-knappen kalder

på den implementerede abstrakte metode 'startGame', som først kalder på 'getChosenGameSize' fra *OptionsBasePanel*-klassen, der tjekker input-felterne med banestørrelsens dimensioner. For ikke at gøre det for svært for spilleren at indtaste en gyldig værdi, ses der bort fra eventuelle mellemrum før, i eller efter tallene. Dette gøres ved at metoden fjerner alle mellemrum ved at erstatte strengen " " med "". Derefter tjekkes der om felterne er helt tomme, da der så printes en fejlmelding. Er de ikke tomme, undersøges der, om tallene ligger mellem 5 og 100. Hvis dette er tilfældet, sættes banens størrelse vha. game-klassernes 'setBoardSize'-metode, sværhedsgraden sættes med 'setTimedMovementSpeed'-metoden med et heltals-parameter afhængig af Difficulty enum-værdien. Til sidsts startes spillet med metoden 'start' og *ViewFrame*-klassens 'showGame' metode kaldes for at skifte options-panelet ud med spil-panelet.

3.4 Evaluering

Forskellige Operativsystemer

For at sikre stabilitet af brugergrænsefladen er spillet afprøvet i forskellige størrelser og på forskellige operativsystemer. Placeringen af komponenterne er ens i Micrsoft Windows 8.1 og dens ældre versioner, iOS og Linux. Derimod har JButtons et lidt andet udseende på iOS, idet deres baggrundsfarve i dette operativsystem ikke er synlig medmindre deres kant skjules. Derudover virker de heller ikke for iOS, hvis de tilføjes til et panel i 'paintComponent'-metoden. Det første problem løses ved at skjule kanten eller i stedet at bruge et ImageIcon til knappen i stedet for at give den en baggrundsfarve. Knappens funktionalitet opnås ved at tilføje knapperne i konstuktøren, men 'setLocation'-metoden kan stadig ligge i 'paintComponent'-metoden for at give den samme effekt som tidligere. Vi havde i gruppen to personer med Windows, en med Linux og en med MAC.

Vi havde nogle problemer med lyden på Linux, idet spillet ikke spillede hele sekvensen fra lydfilerne, når spillet blev startet eller når spillet endte. Denne fejl har vi dog ikke kigget nærmere på, da der kan være rigtig mange fejlkilder. Det kan enten være Java implementeringen, at Linux er installeret forkert, lyd driveren eller andre hardware problemer.

Køretid

Efter tilføjelsen af en masse grafik og udvidede funktioner til Simple Snake, kunne man på nogle computere se at spillet ikke reagerede lige så hurtigt som før, og køretiden var for lang. Dette ses især på en af vores sidste versioner af Advanced Snake, hvis man sætter størrelsen på banen til 100×100. Man kunne ikke holde piletasterne nede, og få slangen til at køre hurtigere. Dette blev løst ved først at lave vores slange om fra billeder til filledRectangles igen, der fik spillet til at køre ved normal hastighed som før. For at få billederne til også at fungere ved normal hastighed, ændrede vi i stedet strukturen af hvornår billederne af slangens krop kalder 'getScaledInstance'-metoden. Tidligere blev hver del af slangen skaleret efter at være blevet udvalgt som den rigtige del igennem en for-løkke, mens alle billederne nu skaleres i forvejen, så billeder der bliver brugt flere gange, ikke behøver at skaleres hver gang. Dette løste problemet så køretiden igen var normal.

Slangens farver

Nederst i *BoardBasePanel*-klassen ligger metoden *colorSnakeImage*, som farver hver pixel i slange-billederne, så de passer med valgene fra menuen. Inden denne metode blev lavet, overvejede vi at importere fire gange så mange snake-billeder, som der oprindeligt var - til fire forskellige farver. Da det i forvejen er mange billeder, der bruges til slange-kroppen, ville dette være for mange billeder at importere, hvis man i stedet kunne finde en metode til farvningen. På denne måde er farve-valg i menu'en blevet optimeret en del ved at bruge WritableRaster til farvningen.

Knapper

Da vi i starten ikke ville bruge JButtons på grund af deres design, var knapper i starten oprettet som tegnede rektangler med lige så store områder hvorpå man kunne klikke, hvilket ville give en illusion af en knap. Senere fandt vi ud af at JButtons kunne modificeres både med hensyn til

udseende og størrelse. Dette gav dog problemer med vores oprindelige GridLayout i panelerne, idet knappen ville fylde hele dens felt i gitteret ud, og derved blive for stor. Derfor endte vi med at bruge standard-layoutet FlowLayout og setLocation, der gjorde det let at manipulere med knappens placering i vinduet.

Figure 3.5: *Rektangler og JButtons*

Implementering af multiplayer

Ved implementeringen af multiplayer-spillet overvejede vi først at bruge en kombineret single- og multiplayer-klasse, som kunne holde et array af slanger. Hvis der var to slanger, ville vi bruge multiplayer-gameplay og singleplayer-gameplay ved én slange. Fordelen ved dette er, at view-delen i MVC-designet, bare kan loop over nogle arrays lige meget om det er single- eller multiplayer-spillet. Det vil være let at tjekke om spilleren er i singleplayer- eller multiplayer-spillet, da skal bare sættes et if-statement der tjekker for antallet af slanger. Det ville være nemt at tilføje flere slanger. Ulempen ved den fælles klasse er, at der vil være mange if-statements. At holde spillene adskilt gør det også lettere at lave justeringer i det ene spil uden at påvirke det andet. Der er ikke er klart overlegent valg, begge implementeringer har sine ulemper og fordele.

Chapter 4

Konklusion

Formålet med projektet var at starte med at lave en simpel udgave af snake, der ville være let at udvide til en avanceret version ved at tilføje flere funktioner. Dette gør det ideelt at tilføje én funktion ad gangen, frem for at planlægge alle funktioner på en gang og tilføje dem samtidig. Resultatet bliver et, til at starte med, simpelt men fungerende program, hvorefter yderligere funktioner kan tilføjes. Programmet er altså udviklet iterativt, hvorved der opstår flere fungerende versioner af spillet, men med forskellige funktioner. Dette gør det muligt at tilpasse programmet, hvis der opstår nye idéer eller krav undervejs, og altså lettere at opgradere Simple Snake til Advanced Snake.

I vores version af Snake, valgte vi at fokusere på en ren og fleksibel brugergrænseflade og en struktureret opbygning af programmet, der ville gøre det let at foretage ændringer og implementere nye funktioner. Dette blev gjort med *Model-View-Controller*-designet, som holdte forskellige klasser adskilt. Derudover gjorde designet det også let at finde rundt i klasserne, når fejl skulle rettes, eller algoritmer skulle ændres.

På trods af meget importeret materiale og mange klasser, forsøgte vi dog også at mindske programmets størrelse, så meget som muligt, for at kunne overskue det. Dette gjorde vi ved at lade nogle klasser bruge metoder fra andre klasser, f.eks. ved tegning af baggrunden. Forskellige klasser, der har meget tilfælles, har vi også ladet nedarve fra en fælles superklasse, der gør genbrug af kode simpel, og underklasserne kortere. Dette ses f.eks. ved *HeaderBasePanel*-klassen der er top-panelet, når man ikke er inde i spil-scenerne. Denne bliver nedarvet af *HeaderSinglepalyerPanel* og *HeaderMultiplayerPanel*, som ses inde i spil-scenerne. Disse genbruger superklassens udseende og knapper, samtidig med at tilføje deres score-tekstfelter. Også *OptionsSingleplayerListener* og *OptionsMultiplayerListener* nedarver fra superklassen *OptionsListener*, der dog er en abstrakt klasse, idet den aldrig skal kunne oprettes for sig selv - ligesom den abstrakte klasse *Game*.

Evalueringen af Simple Snake kunne bruges til videre udvikling af spillet til Advanced Snake, hvor f.eks. skalering af spillet spillede en stor rolle i begge versioner. I den første version bevarede felterne ikke deres størrelsesforhold, mens dette var bevaret i den nye version, hvor noget af vinduet dog derimod ikke blev fyldt helt ud af banen. Nogle af funktionerne i Simple Snake er også bevaret i Advanced Snake, f.eks. brugen af ArrayList, oprettelsen af æblet og 'move'-

metoden, der dog er ændret en smule, men har beholdt den oprindelige metodes fremgangsmåde, da den viste sig at være effektiv.

Med implementeringen af flere scener, paneler og ActionListeners, der gjorde musseklik muligt, var det også vigtigt at have kontrol over hvilket panel der fik fokus og hvornår panelet mistede fokus, idet selve spil-scenen kunne miste fokus, når spillet blev startet og derved gøre det umuligt for spilleren at styre slangen. Med nye implementeringer er det altså generelt vigtigt straks at løse de problemer der følger, da det ellers kan give fejl, der er svære at finde og gennemskue, i det ellers tidligere fungerende spil.

Appendix A

Simple Snake - Klassediagrammer

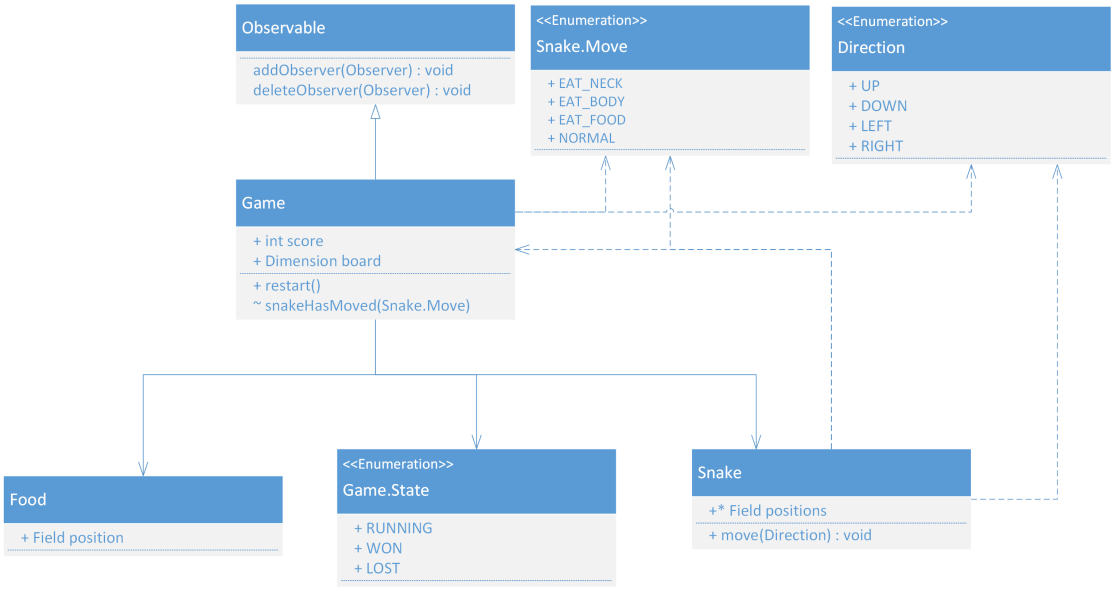


Figure A.1: *UML klassediagram over Model-pakken.*

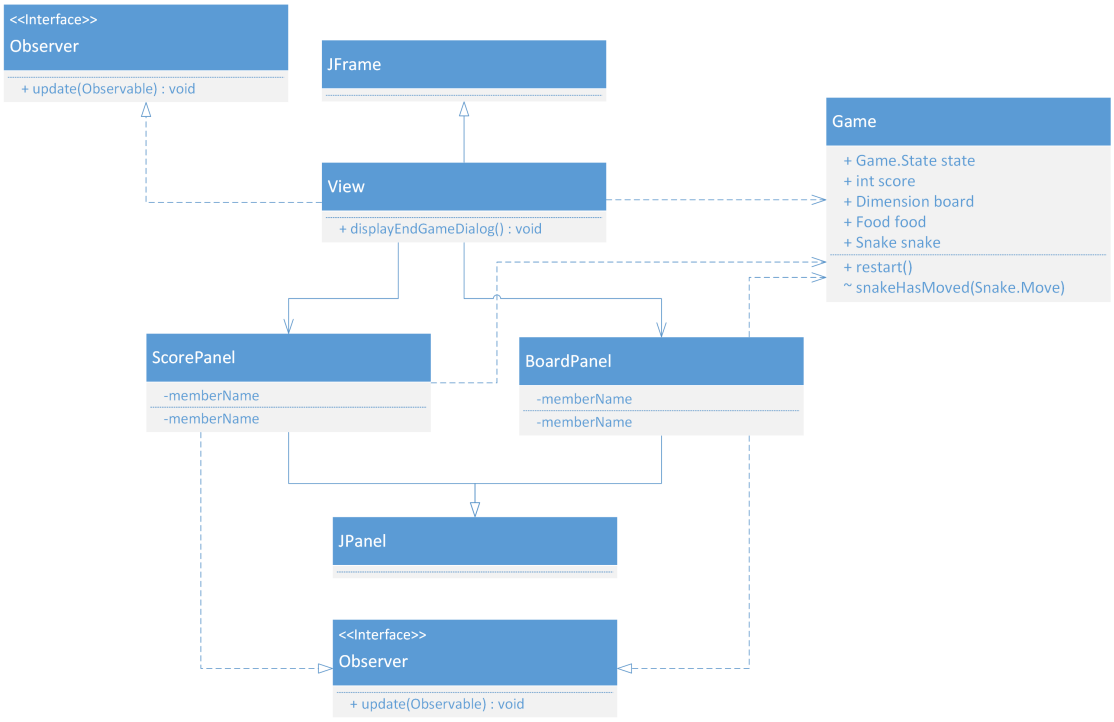


Figure A.2: *UML klassediagram over View-pakken.*

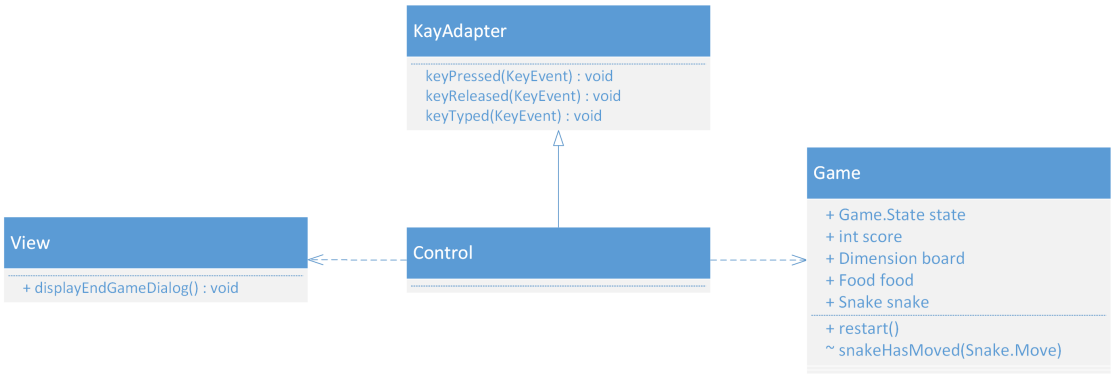
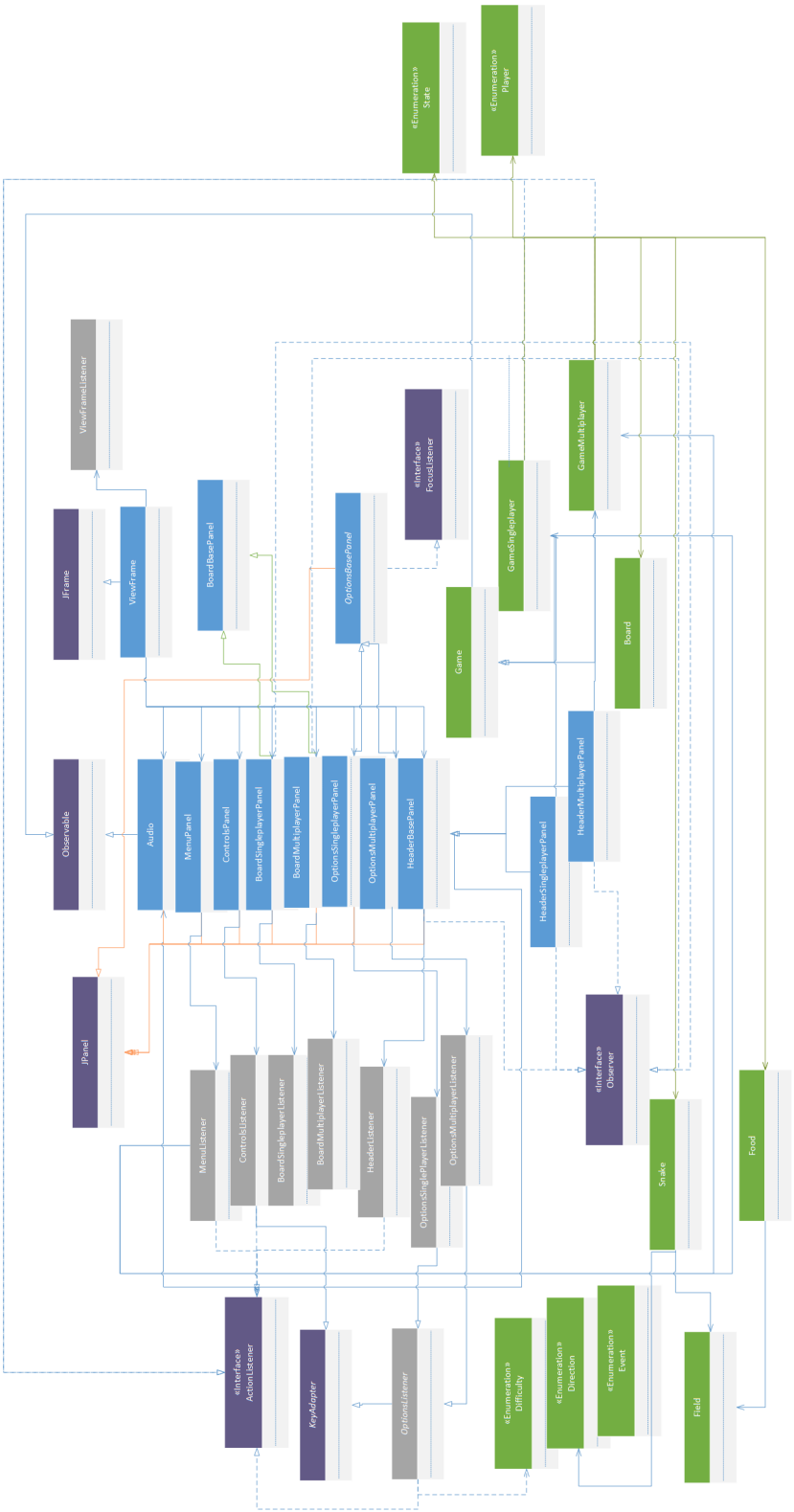


Figure A.3: *UML klassediagram over Control-pakken.*

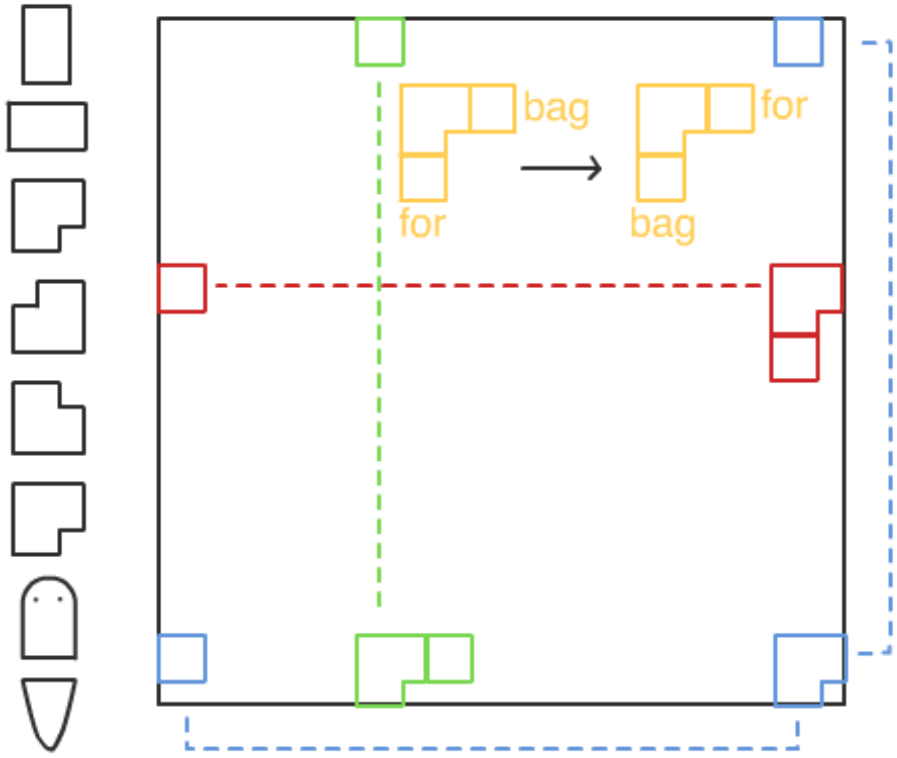
Appendix B

Advanced Snake - Relationsklasser


«Enumeration»
State
«Enumeration»
Player
ViewFrameListener
«interface»
FocusListener
GameMultiplayer
GameSingleplayer
JFrame
ViewFrame
BoardBasePanel
OptionsBasePanel
Game
Board
Observable
Audio
MenuPanel
ControlsPanel
BoardSingleplayerPanel
BoardMultiplayerPanel
OptionsSingleplayerPanel
OptionsMultiplayerPanel
HeaderBasePanel
HeaderMultiplayerPanel
HeaderSingleplayerPanel
JPanel
«interface»
Observer
MenuListener
ControlsListener
BoardSingleplayerListener
BoardMultiplayerListener
HeaderListener
OptionsSingleplayerListener
OptionsMultiplayerListener
Snake
Food
«interface»
ActionListener
KeyAdapter
OptionsListener
«Enumeration»
Difficulty
«Enumeration»
Direction
«Enumeration»
Event
Field

Appendix C

Slangens opbygning



References

[1] Wikipedia. Snake (video game), 2014. URL https://en.wikipedia.org/wiki/Snake_%28video_game%29. [Online; Tilgået 8-Januar-2015].