Indholdsfortegnelse

Indholdsfortegnelse ... 1
Indledning ... 3
Opgaveformulering ... 3
Metode ... 4
Rockspillet ... 5
Præsentation af UP artefakter ... 6
Domainmodels ... 6
Use-cases ... 6
SSD/SD ... 7
Design Models ... 7
Alternative Artefakter ... 7
Processen ... 7
Inception ... 7
Business-case ... 8
Vision ... 8
Use cases ... 8
Risk managment ... 8
Afslutning af inception ... 9
Elaboration ... 9
1. iteration ... 9
Domæne model og klasse diagram ... 9
Workshop ... 11
Afslutning af 1. Iteration ... 12
2. iteration ... 13
Ansv,arsfordeling i systemet under 2. iteration ... 16
Dog usecase – et kort, men givende bekendtskab ... 17
Controllerstrukturen ... 17
Implementering af controllere ... 18
Udvidelse af Modellen ... 19
Afslutning på 2.iteration ... 19
3.Iteration ... 20
Shot ... 20
Enemy AI ... 20
Enemy shot ... 22

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Winning Conditions 22
Items 23
Boss AI 23
Afslutning på 3. iteration 25
Implementering af koden 26
Indledning 26
XP: Test-driven development & Pair-programming 26
Lav kobling og høj samhørighed 28
MVC 28
Model 28
X-Y-styring 28
Level 29
Nedarving 30
Hero 30
Enemy 31
enemyAI() og enemyShot() 31
Boss 32
Controller 34
LevelController 34
HeroMovementController 34
Kollision 35
Timer-Runnable-Thread 36
View 37
Kode-Kritik 37
Refleksion over brug artefaktbrug i forhold til kode 38
Grafik 38
Refleksion over processen 40
Konklusion 42
Litteraturliste 43

Ansvarsfordeling
Tobias Langkjær: s.7-12
Nis Sørensens: s.13-19
Troels Ebbesen: s.20-25
Simon Kunddal: s. 26-32

Antal Tegn
99843

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Indledning

Gennem en iterativ proces har vi i denne opgave udviklet et system, som forholder sig den opgaveformulering vi fik stillet. Vi har gennem de sidste par måneder både været i programmøren og systemudviklerens rolle, idet vi ved brug af Javas objekt orienterede programmeringssprog, har fulgt teorien og processen bag Unified Process og skabt et klassisk platformspil.

Opgaveformulering

Til denne opgave fik vi stillet en opgaveformulering, som vi har valgt at forholde os helt konkret til. Denne gik ud på at konstruere et lille computerspil, hvor det er muligt at styre en figur i en eller anden verden. Fokus i denne opgave skal være på, at den underliggende model er korrekt og kan rumme de krav, der er stillet. Kravene til spillet lyder som følger:

- *En helt*: Denne helt skal være styret at spilleren
- *Minimum 3 slags fjender*: Disse fjender er styret af computeren og helten skal enten slå dem ihjel eller bare gå uden om dem.
- *Minimum 3 slags genstande:* Disse genstande kan helten samle op og enten få en bonus eller point på en eller anden måde.
- *Spillet skal indeholde 3 forskellige baner med stigende sværhedsgrad*. Denne sværhedsgrad kan være flere fjender, større areal eller flere opgaver.
- *Hver bane skal have et mål.* For at komme fra bane til bane skal der være et mål som er klart for spilleren. Det kan være at samle alle genstande, besejre alle fjender eller bare nå til et bestemt felt.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Metode

Vi vil starte opgaven med at præsentere vores endelige spil, som det ser ud og fungerer efter inception og tre iterationer i Elaboration. Herefter vil vi i korte træk beskrive de artefakter, vi har gjort brug af igennem vores proces.

Efter præsentationen vil vi mere dybdegående forklare vores fire faser med fokus på de mest interessante og afgørende af vores valgt, samt på hvilken baggrund de er blevet foretaget. Vores brug af artefakter vil løbende blive beskrevet sammen med vores implementering ud fra forskellige designmønstre. Heri vil vores risikovurderinger og overvejelser i de forskellige iterationer løbende fremgå.

Derefter vil vi forklare nogle af de mest komplekse og dominerende dele i vores kode. Vi vil her beskrive de problemstillinger der lægger til grund for vores valg af klasser og metoder.

Afslutningsvis vil vi reflektere over den iterative proces og de begreber og teorier den bygger på. Vi vil her beskrive vores oplevelse af processen som helhed og reflekterer over hvilke valg vi kunne have gjort anderledes.

I vores konklusion vil vi samle op på opgavens problemformulering og de ting vi har lært af processen.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Troels Ebbesen
Nis Sørensen
Tobias Langkjær
Simon Kunddal

Rockspillet

Vores produkt er blevet til et humoristisk lille computerspil, hvor brugeren kan styre helten gennem skumle miljøer med masser af fjender. Helten er en rockstar i bedste 80'er stil, som fra enden af sin guitar skyder rødglødende ildkugler, mens han forsøger at bane sig vej gennem en beskidt storby, fyldt med rabieshunde og skydegale hiphoppere. Målet for vores rockstar er hans onde managers kontor, hvor manageren må stå til regnskab for mange års afpresning og udnyttelse af et stort talent. På vej fra hans hjem, i den hårde del af byen, må vores rockstar kæmpe sig vej gennem byens gader, over hustage og neonskilte, og gennem beskidte kloaker, baggårde og trænge elevatorskakte. På sin vej opsamler han whiskyflasker, smøgpakker og grønne dollars. Gennem mange års rockslid har vores rockstar opbygget et specielt forhold til netop whisky og smøger, som nu har en helbredende effekt på hans seje legeme. Gennem skarpskåret baggrundsrock og lækker layoutgrafik lokkes brugeren ind i et faretruende univers, hvor karakteren, der er hurtigst på aftrækkeren, har magten.

Når spillet åbnes fremkommer en 15 sekunders introskærm og spillet startes herefter automatisk, lige på og hårdt. Rockstaren i spillet styres af keyboardets piltaster og hans skud affyres med space. Han bevæger sig i en fast hastighed til højre og venstre, mens hans hop kan reguleres i højde ved at holde pilop-tasten i

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

bund. Brugerens mål er at gennemføre fem baner og til sidst dræbe bossen, altså manageren i spillet, og samtidig samle så mange point som muligt. Der er samtidigt en bonus-score, der tæller med i det samlede i regnskab, afhængigt af hvor hurtigt spillet gennemføres.

Præsentation af UP artefakter

Vi har løbende inddraget forskellige artefakter i vores proces. Vi vil i dette afsnit prøve at danne et overblik over hvordan de forskellige artefakter er blevet brugt. Som tidligere nævnt har vi opdelt processen i inception og den efterfølgende Elaboration som beskrevet af Larman. Han præsenterer brugen af artefakter i forhold til den del af forløbet, man befinder sig i, og denne har vi i store træk fulgt, for at skabe sammenhæng mellem vores proces og den, der fremstilles i bogen. Som Larman selv nævner, kan artefakter have mange forskellige former, og man er frit stillet til at bruge dem efter eget ønske. Det vigtigste er, at de tilføjer processen en praktisk værdi. De skal kunne sige noget om systemet og tilføre viden til projektet. Meget af vores arbejde har foregået i grupperum hvor vi havde adgang til tavler og whiteboards, hvor vi i fællesskab har kunnet tegne og diskutere, ændre designet og diskutere lidt mere, i ægte agile stil. Vi har derfor oprettet en del forskellige artefakter af varieret kvalitet under forløbet. Vi har i opgaven valgt at beskrive de artefakter, som vi mener, har haft indflydelse på vores valg og vores systems endelige form. Dem vi nævner igennem opgaven er vedlagt som bilag.

Domainmodels

Noget af det første vi gjorde efter inception var at skitsere en domænemodel ud fra hvilke grundlæggende fænomener vores system skulle repræsentere. Den beskrev vores problemområde ud fra konceptuelle klasser, og grundstrukturen i vores system stammer herfra. Artefakten har været brugt og opdateret igennem processen, i visse tilfælde har vi taget os friheden til at tegne systemklasser ind i modellen, og derved bøjet reglerne en smule, da disse er et produkt af systemet og derfor ikke en repræsentation af omverdenen. Vi mener dog, at den stadig har haft en funktion for en overordnet forståelse af systemet. Yderligere har vi ikke sigtet efter at skabe fuldstændigt korrekte modeller, som ifølge Larman kan lede til "*analysis paralysis*"[1], hvilket vi gerne ville undgå i processen.

Use-cases

Use-cases har været vores mest brugte artefakt, og dem vi føler har fået mest udbytte af. Vores tilgang til disciplinen har været at simplificere systemet, samt at abstrahere et udsnit af funktionaliteten ned på et niveau, hvor vi alle kunne diskutere og forstå det. Vi har primært benyttet os af sub-function use-cases til at beskrive systemoperationer og i mindre grad user goals, da vores system har meget enkle mål og vi mener,

[1] Larman (2005): s. 133

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

at dem vi lavede under inception dækkede de krav en bruger ville stille til systemet. Vi har oprettet dem på tidspunkter, hvor vi var i tvivl om, hvordan forskellige elementer skulle implementeres. Men også på områder, hvor vi ville undersøge, hvad nye elementer ville kræve af systemet.

SSD/SD

På baggrund af de use-cases vi syntes var interessante, oprettede vi sekvensdiagrammer (SD). Vi oplevede at denne visualisering af systemelementer og deres interaktion med hinanden var et anvendeligt værktøj til at analysere systemet i forhold til GRASP. Vi har også brugt systemsekvensdiagrammer (SSD) under 1. iteration til at vurdere systemet ud fra et brugerperspektiv.

Design Models

Vi har i vores proces brugt klasse diagrammer til at illustrere vores klasser og deres forbindelser til resten af systemet. Vi har brugt dem i en mere simplificeret form, hvor fokus har været på at skabe et udgangspunkt, hvorpå vi kunne basere implementeringen af systemet. Vi baserede vores diagrammer på vores domænemodel, som videreudvikledes ved at tilføje metoder og attributter til klasserne. Igennem processen har vi yderligere brugt forskellige modelleringsværktøjer til at visualisere vores implementerede kode hvilket igen har bidraget til en øget forståelse for systemet.[2]

Alternative Artefakter

Udover de anerkendte artefakter som nævnt ovenfor, har vi udviklet mange forskellige former for arbejdspapirer, der har hjulpet os på mange måder. Vi har tegnet skitser over hvordan vores spil kunne komme til at se ud i en GUI, de forskellige baners design, samt styring via X-Y koordinater. Vi har lavet mockups af grafikken ved at klippe en masse billeder sammen og sætte en karakter ind for at se hvilke muligheder vi havde for at gøre designet mere spændende. Udover disse har vi produceret en stor mængde af små tegninger og kruseduller som vi mener, har tilført processen en værdi.

Processen

Inception

Inceptionfasen bruges til at beskrive visionen for projektet og til at vurdere holdbarheden af det endelige produkt. Med det menes om visionen kan opfyldes tilfredsstillende indenfor det givne tidsrum. Desuden begyndes på udviklingen af forskellige artefakter, men ikke noget dybdegående. Det er stadig en

[2] Eksempelvis JUDE Community og Poseidon.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

indledningsfase, som skal anskueliggøre om projektet er værd at arbejde videre med. Det gælder så at sige om at skabe en holdbar vision, ud fra tidshorisonten på sit projekt.

I starten af april begyndte vi med inceptionsfasen, hvor vi skulle vælge hvilket projekt vi ville arbejde med. I denne periode lavede vi en vision, et glossary, en business model og det første udkast til en domænemodel. Vores første vision var, som beskrevet tidligere at lave et platformspil. Derudover var der angivet nogle minimumskrav som skulle opfyldes, og disse blev grundlaget for vores vision.

Business-case

Vi forestillede os et scenarie, hvor en musik portal havde bestilt et spil der kunne lokke nye kunder til, og holde de personer der i forvejen brugte sitet længere derpå. Det eneste krav var, at det skulle være et "musik-spil", men stadig action-præget. Det vil sige, det skulle have et eller andet forhold til musik, og være interessant at spille flere gange. Derudover skulle spillet have en high-score, så det var muligt at bruge spillet i forbindelse med konkurrencer og lignende.

Vision

Vores vision var at lave et platform spil, hvor man styrer en rockstar rundt i forskellige baner. Denne rockstar skulle kæmpe sig forbi nogle fjender, som var af forskellig styrke, for at komme til næste bane. I de forskellige baner skulle der være health-items (smøger og whisky), som helten kunne tage, og derved øge sin health. Helten skulle have en magisk guitar, som kunne skyde med guitar riffs, og han ville derved kunne dræbe sine fjender. Til gengæld skulle han selv miste health, hvis han blev ramt af fjendernes skud, eller hvis han fysisk ramte fjenderne. Målet for rockstaren skulle være at dræbe sin onde manager, der skulle fungere som en boss i spillet.

Use cases

I inceptionfasen blev der produceret en del use-cases, der hjalp os i gang med opgaven ved at sætte ord og fokus på de risikoområder, der kunne opstå. Vi havde en formodning om, hvilke dele af koden der kunne volde problemer, og vores use-cases hjalp med at få bedre styr på disse områder. Vi har vedlagt et par, af de use-cases vi benyttede os mest af, som bilag. USECASE?

Risk managment

Vi definerede nogle højrisiko-områder, som vi ville lægge fokus på tidligt i processen. Disse højrisikoområder var en form for styring af XY-koordinater samt en gravity-effekt, som vores helt skulle påvirkes af. Derudover vil vi gerne, for vores egen skyld, have en simpel GUI, som skulle give os mulighed for at finde fejl og følge med i implementeringen.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Afslutning af inception

Udover disse overvejelser og oprettelse af artefakterne, lavede vi en plan for, hvad vi skulle nå i Elaborations første iteration. Vi planlagde at have tre iterationer af perioder af tre uger. Derudover ville vi lave en afslutning på hver iteration, hvor vi så tilbage på hvad vi havde nået, i forhold til de mål vi havde opstillet. Dette skulle sikre, at vi ikke kom til at hænge for lang tid i en iteration, selvom der opstod problemer som vi ikke umiddelbart kunne løse.

Vores mål til 1. iteration var at udvikle en domæne model, som så senere skulle lægge bunden for den design model vi skulle bruge. Vi ville også gerne undersøge hvordan man kunne kode keyevents, som blev afspejlet i en GUI. Vi ville derudover forsøge at kode et objekt, der var tynget af en gravity-effekt. Vi vurderede nemlig dette som et højrisiko-område. Dette var hovedpunkterne for den første iteration, hvor vi også ville producere nogle flere use-cases, SD'er og SSD'er, som kunne understøtte senere implementering.

Elaboration

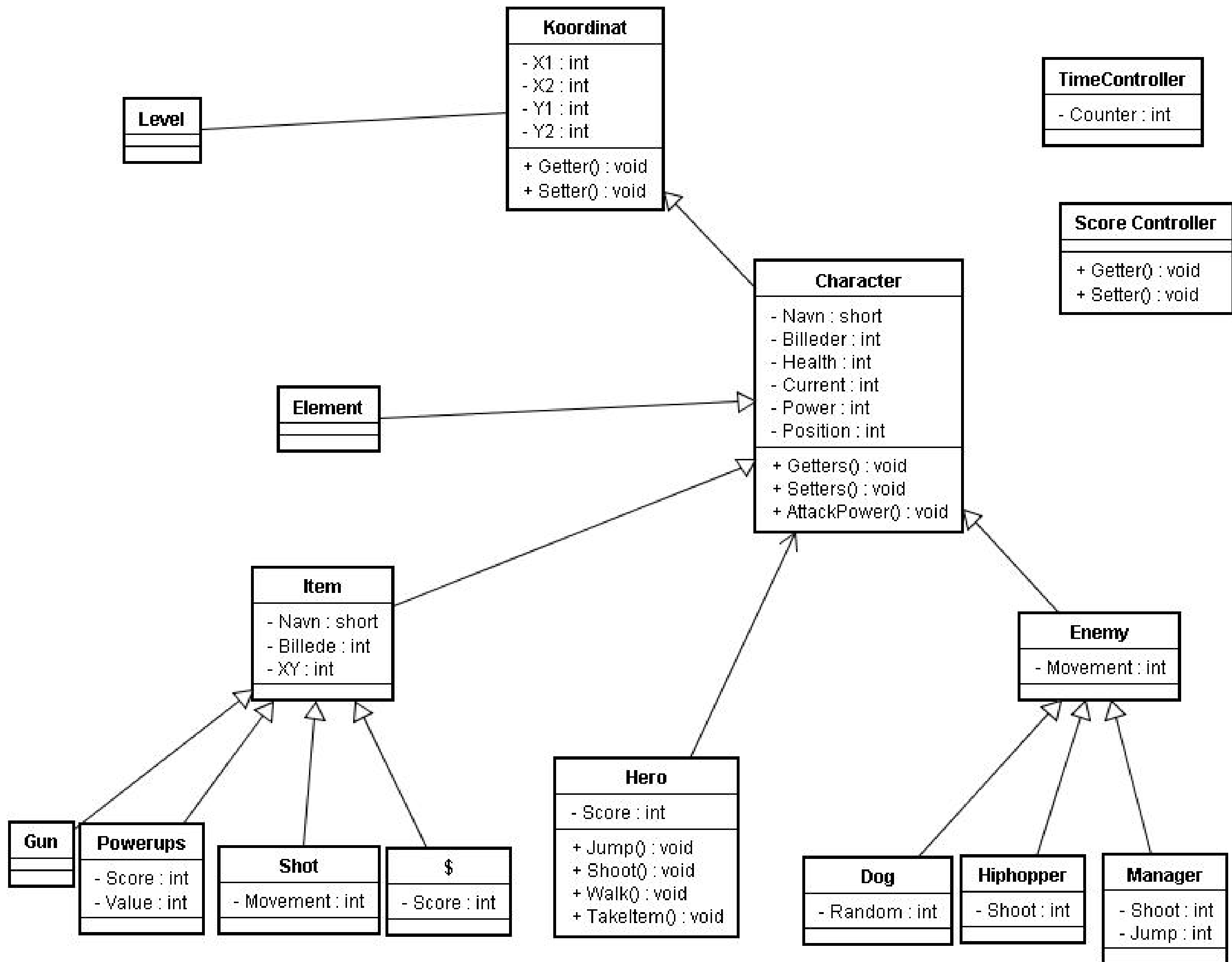
1. iteration

Domæne model og klasse diagram

I denne periode lavede vi vores første udkast til en domænemodel og et klassediagram. Disse modeller er tæt forbundet, og giver derfor mening at lave dem i samme iteration. Vi begyndte med at lave en domæne model, som vi derefter ville "oversætte" til en design model. En domænemodel, er den model der bliver udviklet sammen med kunden og hvor man benytter nogle af de termer som giver mening for kunden. I vores tilfælde var kunden os selv, og vi tog derfor udgangspunkt i den vision vi havde defineret i inception. Herved fandt vi frem til de væsentligste klasser, hvorudfra vi efterfølgende producerede en glossary[3]. Denne glossary forklarede i runde vendinger, hvad objekterne i hver klasse skulle kunne eller opfylde i spillet. Vi begyndte herefter at konstruere vores domænemodel. Vi havde endnu ikke den grundlæggende opfattelse af, hvordan alle klasser skulle være forbundet, men vi var enige om mange af de metoder og attributter, som klasserne skulle indeholde. Derfor forsøgte vi, ud fra vores domænemodel, at skabe en design model, hvor attributter og metoder blev sat ind. Designmodellen var på dette tidlige tidspunkt uden tanke på hverken controller eller GUI.

[3] Se bilag 5.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image



Figur 1

I forbindelse med opbygningen af domæne og design modellerne besluttede vi at bruge nedarving som en rød tråd gennem hele modellen. Vi havde en idé om at nedarving kunne spare os for en del unødvendig og ens implementering i de forskellige klasser i modellen. Eksempelvis ville subklasser kunne overskrive supermetoder ved hjælp af polymorfisme, og herved kunne elementer som items og enemies gøres specifikke. Som det fremgår af modellen valgte vi at oprette klasserne Character og Item, som de mere specifikke klasser, eksempelvis Hero og Enemy, samt powerups og shot, kunne nedarve fra. Samtidig kunne der foretages en yderligere specialisering af Enemyklassen med dens tre subklasser, Dog, Hiphopper og Manager (Boss).

Allerede senere i iterationen overvejede vi navnet for den klasse vi havde kaldt Koordinat, og fandt frem til, at vi hellere ville kalde den Element. Alle dens subklasser kunne nemlig defineres som elementer i en level. Samtidig omdøbte vi den tidligere Elementklasse til Structure. Derudover ændrede vi i de parametre Elementklassen havde i sin constructor. De kom i stedet for x1, x2, y1 og y2, til at hedde x, y, xLen og yLen. X og Y er koordinatet i vores grid, mens xLen og yLen er henholdsvis bredden og højden på

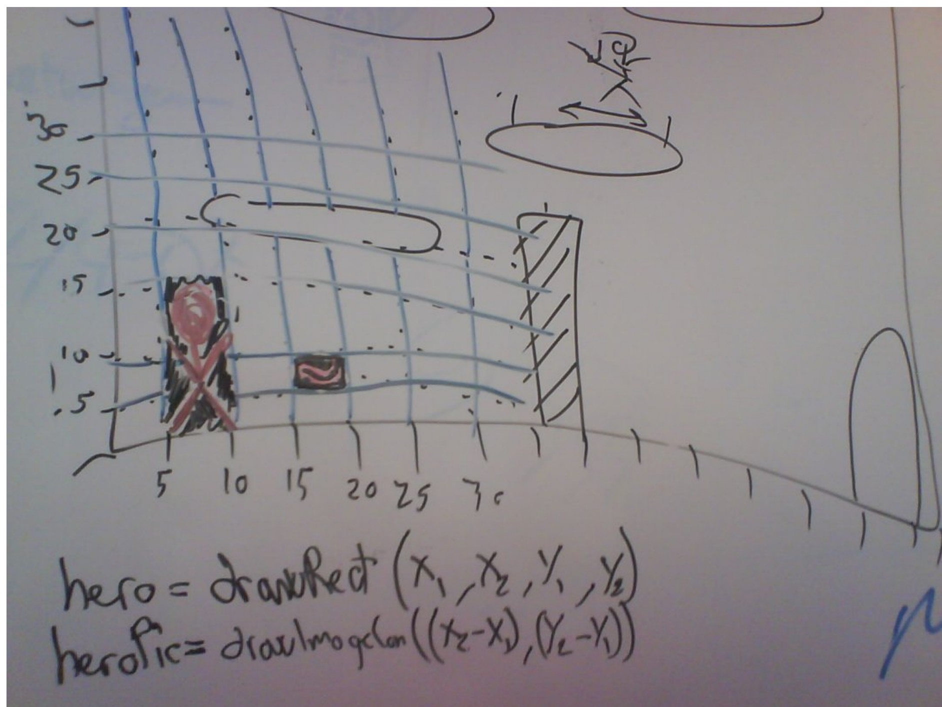
created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

selve elementet. På grund af nedarving ville en klasse som Element kunne indeholde generelle metoder, der kunne bevæge elementer i et koordinatsystem. Disse metoder var nemmere at implementere i systemet, når der ikke skulle flyttes to koordinatsæt, men i stedet kun et. Denne opfattelse vil blive forklaret senere i opgaven, sammen med forklaring af attributter og metoder for denne superklasse.

Workshop

I slutningen af 1. iteration havde vi arrangeret at lave en workshop, hvor vi ville sætte kodningen i fokus. Vores mål var, at få kodet de højrisiko områder vi havde opstillet, og gerne få en mini version af spillet, så vi havde noget konkret at kigge på. Selve workshoppen fungerede ved, at vi mødtes alle fire og aftalte hvem der skulle lave hvad. Derefter satte vi os sammen to og to og pair programmerede, indtil den opgave vi havde stillet os var løst. Vi snakkede løbende sammen, når der var noget vi var i tvivl om, eller som lå udenfor det ene pars begrænsninger. Derudover benyttede vi et whiteboard til at optegne SD, SSD og andre artefakter som vi fandt behjælpelige, når vi ramte en blindgyde.

Vi fik inspiration til XY-styringen ud fra et tidligere eksempel fra vores Java lærebog – ”Java Software Solutions”, hvor vi samtidigt ud fra vores use-case om XY-styring, havde et godt grundlag for implementering af denne risiko-vurdering fra inception. Vi diskuterede use-casen, samt det første klasse-diagram og om hvordan det kunne implementeres. Dette førte til en skitse på et whiteboard, som vi alle blev enige om kunne bruges. Her fik vi lavet skitser til en mulig kollisions-beregner (distance-to-object). Samtidigt fandt vi også ud af, at alle vores objekter i spillet skulle fungere som firkanter.



Figur 2

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Ved afslutningen af workshoppen, havde vi opnået vores mål. Vi kunne nu bevæge en grøn prik rundt i et frame, hvor man så kunne ramme nogle røde prikker, som skulle gøre det ud for items. Det var ikke meget, men bare det, at vi havde fået noget visuelt, altså lidt GUI, betød meget for os, da det gjorde det nemmere at forholde sig til spillet og hvordan det skulle se ud. Samtidig lagde det en god grund for resten af koden, da vi allerede på daværende tidspunkt, havde en gravity der nogenlunde virkede. Vi kunne samtidig flytte helten rundt som vi ville i banen, og dermed fungerede vores XY-styring også.

Vi havde derudover et ønske om at få implementeret en gravity-effekt, der var virkelighedstro. Derfor fandt vi et eksempel på Internettet, hvor en tidligere IBM medarbejder havde lavet en meget avanceret gravity-effekt, som fulgte de matematiske love om tyngdekræften[4]. Dette eksempel virkede som en inspiration da vi skulle til at kode vores egen gravity. Vi fik også lavet en gravity, der virkede, men desværre kunne vores tidlige kollisions-kode ikke nå at stoppe helten, når han bliver trukket ned af tyngdekræften. Dette skyldes, at hans fart øgedes eksponentielt, og derved kunne kollisions-koden ikke nå at stoppe ham, på et enkelt pixel. Resultatet var at det blev tilfældigt om helten stoppede på en struktur, eller om han fortsatte igennem. Dette var selvfølgelig ikke tilfredsstillende, og det blev derfor opsat som et risikoområde til 2.iteration, at få gravity til at fungere. Men alt i alt var det en effektiv workshop, som gjorde at vi kom godt i gang med koden, og gav os et godt afsæt til 2. Iteration.

Et af problemerne med vores kode på daværende tidspunkt, var dog at vi ikke havde implementeret med tanke på Generel Responsibility Assignment Software Patterns (GRASP) og Model-View-Controller (MVC). Dette kunne blandt andet ses i vores GUI, hvor vi havde information om kollision mellem helten og andre elementer til at ligge. Vi var udmærket klar over, at kollisions-koden ikke skulle ligge i GUI'en, men for at have noget at fremvise i afslutningen af 1. Iteration, blev den midlertidigt oprettet her, og vi ville så senere placere den mere hensigtsmæssigt. GRASP- og MVC-mønster var et område som vi valgte at udskyde lidt, idet vi prioriterede højere at få noget specifik kode der virkede.

Afslutning af 1. Iteration

I 1.iteration begyndte vi på at udfærdige de artefakter vi skal bruge til projektet. Nogle af de mål vi havde sat for denne iteration, var at få lavet en domænemodel og klassediagram. Dette mål blev opnået, og det gjorde samtidig, at vi begyndte at overveje områder, som udsprang af processen. Blandt andet fandt vi ud af, at nedarving skulle være en central del af systemet, og benyttes hvor det kunne lade sig gøre. Efter vores introduktion til MVC, besluttede vi at opbygge vores model efter dette mønster, da det kunne hjælpe os med at få adskilt model, controller og GUI. Dette ville medføre en god og overskuelige struktur, som ville give et større overblik ved implementering. I vores planlægning af næste iteration ville vi også lægge vægt på at implementere i forhold til GRASP-mønstre. Spillet ville godt kunne fungere uden disse mønstre, men

[4] http://www.geocities.com/Paris/6502/impact.html

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

det er god programmørskik at overholde dem. Samtidigt vil det gøre det nemmere at foretage ændringer i dele af systemet uden det får konsekvenser for helheden.

Udover implementeringen af GRASP og MVC var vores plan at videreudvikle spillet, på det grundlag vi havde skabt i 1.iteration. Vores højrisiko-områder i 2.iteration vurderede vi til at være kollision/interaktion imellem objekter, da vores midlertidige implementering fra 1.iteration ikke var tilfredsstillende.

2. iteration

Vi har valgt at gå i dybden med de område vi mener, har været afgørende for den endelige struktur vores system har fået, samt de tanker og overvejelser vi havde igennem 2.iteration. I forrige afsnit beskrev vi hvordan vi skabte den grundlæggende arkitektur og testede at vi kunne udvikle den nødvendige funktionalitet, vores endelige system ville kræve. Vi havde haft forskellige designpatterns i baghovedet, men da vi ikke var sikre på hvilke muligheder vi i realiteten havde, var meget af koden ikke i overensstemmelse med de forskellige principper. Derfor var planen for denne iteration, en korrekt implementering af koden i forhold til GRASP og MVC. Styringen via X-Y koordinater var fastlagt og nu var der fokus på, hvordan vi kunne få de forskellige objekter til at interagere med hinanden. Interaktionen imellem objekterne i spillet er baseret på kollisioner. Derfor skulle der udvikles en klasse der kunne holde styr på hvor og hvem der kolliderede med hvad. Dette førte til en væsentlig udvidelse med 4 nye controllere, som var afgørende for den endelige form på vores system.

MVC

Under 1. iteration blev vi enige om at vores spil skulle overholde Model-View-Controller princippet og vi gik derfor i gang med at implementere dette ved starten af denne iteration. Princippet omhandler kommunikationen mellem domæne laget og GUI´en. Ifølge Larman er der mindst to dele der skal overholdes:

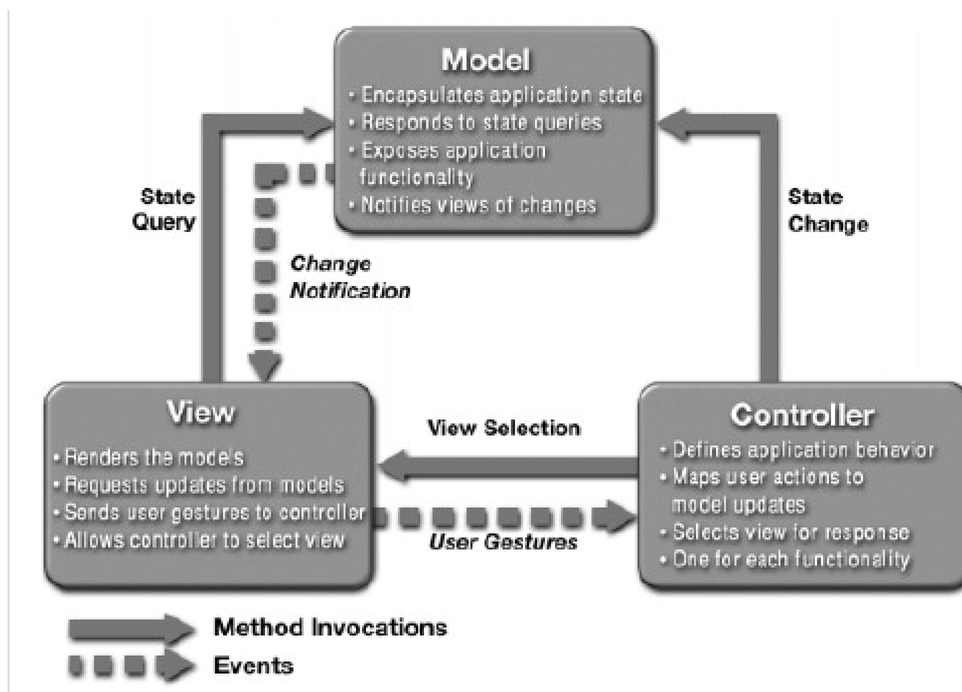
1. Der må ikke være en direkte kobling mellem GUI og ikke-GUI objekter, de skal operere uafhængigt af hinanden.
2. GUI´en må ikke have logiske applikationer og altså ikke foretage beregninger. GUI´en skal kun initialisere grafiske elementer og modtage brugerinputs.[5]

En korrekt implementering af MVC ville give vores system en større fleksibilitet, hvor modellen fungerede uafhængigt af en GUI. I modellen lå de centrale funktioner, altså selve logikken i spillet. View initialiserer og

[5] Larman (2005): s. 209

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

opdaterer alle grafiske elementer baseret på modellens beregninger. Controllerlaget modtager brugerinput og uddelegerer det ud i modellen, samt koordinerer systemoperationer.



Figur 3

Inddragelsen af MVC princippet tidligt i udviklingen gav flere fordele. Opdelingen i de adskilte lag og hvert lags ansvarsområde bidrog til en bedre forståelse for systemets opbygning. Vi kunne i første omgang koncentrere os om at udvikle et fungerende model- og controller-lag, uden at skulle bekymre os om det grafiske. Spillet skulle kunne køre udelukkende i modellen, og den grafiske side skulle kobles på efterhånden som behovet opstod. Under den tidlige udviklingen havde vi, som nævnt, haft stor nytte af at kunne visualiserer funktionaliteten i vores test-GUI. I et system af vores omfang var det forholdsvis nemt at lokalisere fejl og justere koden ud fra det vi kunne se på skærmen.

Vi anså MVC princippet som et af de betydelige kriterier vores spil skulle overholde. Grunden til dette er den stabilitet og fleksibilitet det tilfører et system, hvis det implementeres korrekt. En model, der opererer uafhængigt kan kobles til flere forskellige GUI´er og kan derfor bruges i flere steder og sammenhænge. Vores spil, der illustrerer en afdanket rockstjerne, der skyder med flammende knytnæver, kunne via en anden GUI ligeså godt være en prinsesse, der skød med vand for at redde blomster. Ud fra modellen kunne vi udvikle mange forskellige spil, der henvender sig til forskellige målgrupper. Dette gør, efter vores mening, MVC mønsteret relevant. En god kode, der opfylder de grundlæggende GRASP principper gør det muligt at genbruge klasser og klassestrukturer andre steder. De fremstår på denne måde som en slags legoklodser, hvorpå man kan opbygge mange forskellige systemer ved at genbruge generelle

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

klasser. Da MVC afskiller modellen fra GUI´en kan man bygge modellen op ud fra ens ”klodser” og derefter udvikle en ny GUI eller justere en allerede eksisterende, så den kan dække det problem, det nye system skal løse. Denne mulighed er noget vi diskuterede en del i gruppen, og vi mener at det skaber et incitament til at producere en god og stabil kode, fordelt ud i generelle klasser og subsystemer, der kan genbruges mange steder.

Til implementeringen af MVC princippet brugte vi Gang of Four (GoF) mønsteret Observer. Mønstret bliver også kaldet publisher/subscriber (observer/observable), hvilket beskriver konceptet godt. En publisher udgiver informationer, uden at vide hvem, der modtager dem, og en subscriber abonnere så at sige hos en publisher, og sørger for at få de informationer en publisher udgiver. Larman foreslår, at man skaber den lave kobling mellem GUI´en og modellen ved at definere et interface, der kan informere subscribers, når der sker forandringer i modellen. Publisher kender ikke sine subscribers, men kun de klasser der implementerer interfacet. Kobling går derved kun til interfacet og ikke til en klasse.[6] Interfacet er en del af Java biblioteket og importeres på følgende måde:

```
import java.util.Observer;
import java.util.Observable;
```

Det lykkedes os, med stort besvær, at implementere Observer mønsteret. GameController implementerede interfacet Observer og Element klassen extendede observabel, som vist i eksemplet her.

```
public class GameController implements Observer, ActionListener {

        public void update(Observable arg0, Object arg1) {
                this.frame.repaint();
                (....)

public class Elements extends Observable{

        public void moveXY(int x, int y){
                this.x+=x;
                this.y+=y;

                this.CoordinateSet.set(0, (this.x));
                this.CoordinateSet.set(1, (this.y));

                setChanged();
```

[6] Larman (2005): s. 463-472

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

```
notifyObservers(this);
(….)
```

Vi valgte at lade Element klassen nedarve fra observabel. Dette faldt os mest logisk da denne i forvejen havde ansvar for at de underliggende klasser fik X-Y koordinater. Vi kunne have valgt at oprette et interface i stedet. De klasser, der skulle bruge funktionaliteten kunne have implementeret dette, og på denne måde kunne vi have undgået, at f.eks. vores Structure-klasse også fik funktionaliteten, da dens objekter ikke skulle opdateres i GUI´en. Vi valgte dog at lægge ansvaret hos Element. Alle de klasser der nedarver fra Elements fik metoden notifyObserverves, hvilket satte dem i stand til at sende en meddelelse når de havde foretaget en forandring. GameController havde Observer funktionen og holdte øje med de klasser der var observabel. På denne måde forblev modellen adskilt. Der forekom derfor ikke direkte metodekald fra modellen og op til controller laget. Controlleren foretog kaldet ned i modellen via update metoden, og hentede de informationer den skulle bruge fra de forskellige klasser i modellen, hvorefter MainPanel fik besked på at opdatere sig igennem repaint metoden.

Efter vi havde fået implementeret Observer mønsteret oplevede vi at vores system begyndte at lagge. Specielt havde vi problemer med at nogle keyevents ikke blev opdateret. Dette førte til irriterende fejl, hvor man sad og hamrede løs på knapperne for at få helten til at reagere. En anden effekt af vores implementering var en fejl i opdateringen af GUI´en. Da de observerede klasser kun sendte besked til Observer ved tilstandsforandring, opstod der et opdaterings fejl. Det virkede som om vores GUI kun blev repaintet sporadisk og ikke løbende, hvilket resulterede i objekterne på skærmen forsvandt og øjeblikket senere dukkede op et andet sted. Vi kunne ikke identificere, hvor fejlen opstod, og valgte til sidst i iterationen at implementere MVC på en lidt anden facon end beskrevet af Larman. Vi mener stadig at vi overholder grundideen af MVC og dette vil blive beskrevet under implementeringen af MVC, hvor vi går i dybden med vores nuværende systems opbygning.

Ansv,arsfordeling i systemet under 2. iteration

Igennem processen har vi haft fokus på hvordan vi bedst kunne fordele ansvar i vores system. Sammen har vi diskuteret designet af systemet i forhold til teorierne bag de forskellige GRASP mønstre. I denne iteration gik vi derfor i gang med at analysere vores system, for at sikre os at mønstrene blev overholdt i så stor grad som muligt. En spændende del under denne iteration var udvidelsen af controller-laget, hvor vi fik koblet teorierne med det praktiske og, efter vores mening, endte med en rigtig god løsning, der har medvirket til at give vores system en lav kobling og høj samhørighed.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Dog usecase – et kort, men givende bekendtskab

Som nævnt var kollisioner et af vores riskiko-områder for denne iteration. Vi forestillede os en controller klasse, som vi i første omgang kaldte *DistanceTo*, der skulle holde styr på alle objekter under spillets eksekvering. Controlleren skulle kende Level klassen, der indeholdt ArrayLister over statiske og dynamiske elementer. Igennem DistanceTo´s kendskab til Level kunne den sammenligne objekternes koordinater og reagere og give besked, når der opstod kollisioner. For at undersøge, hvordan dette kunne fungere i praksis skrev vi en use-case over, hvordan en Dog skulle bevæge sig, mens dens blev overvåget af DistanceTo.[7] Ideen var, at hvis vi kunne finde en fornuftig løsning på hvordan Dog, den simpleste af vores fjender, kunne styres, ville det være nemt at overføre dennes funktionalitet til de andre enemy klasser. Use-casen var ikke særlig korrekt, men viste flere svagheder i vores planlagte design. Den ene var måden hvorpå vores fjender bevægede sig. Hver Dog skulle tildeles to koordinatsæt, og når den ramte en af disse skulle den vende og gå i den anden retning. Dette var ikke en holdbar løsning, da hver Dog skulle afpasses til den Structure den startede på, hvilket ville kræve, at hvert Dog objekt skulle oprettes med et individuelt bevægelses mønster, og dermed ikke være generel. Det ville være bedre hvis Dog klassen havde metoder til selv at styre, hvornår den skulle vende. Men hvordan kunne den vide hvor den var, i forhold til andre objekter, uden at dette medførte en høj kobling i systemet? En løsning ville være at lade DistanceTo fortælle Dog, hvornår den skulle vende. Den fremstår altså som *information expert*, da den ville få den nødvendige viden til at påtage sig dette ansvar. På den måde kunne vi undgå at koble Dog til Structure-klassen. Derfor fik DistanceTo ansvar for at sætte de informationer, der fik Dog til at skifte retning, samt tjekke alle kollisioner i mellem Dog og Structure[8]. Vi indså at DistanceTo-controlleren ville blive meget omfattende og få et stort ansvar når vi senere skulle implementere resten af vores fjender, items og skud, da de alle skulle styres fra denne controller. Vi begyndte derfor se efter andre muligheder, for at løse disse problemer og på den måde sikre at vores controllerklasser opretholdt høj samhørighed.

Controllerstrukturen

Vi vurderede at de to oprindelige controllere fra 1. iteration og den planlagte movement/collision controller ikke var tilstrækkelig. GameControlleren modtog bruger intputs og var samtidig creator for LeveController som i sin tur fik ansvar for at oprette alle levels med tilhørende objekter. DistanceTo skulle derefter koordinerede alle objekterne ud fra kollisioner. Overordnet set var der altså 3 forskellige områder vores controllere skulle dække; koordineringen af brugerinputs, oprettelse af objekter og levels, samt kontrol og styring af dynamisk indhold. Vi tog udgangspunkt i disse områder og begyndte at undersøge, hvordan ansvaret kunne fordeles bedst muligt. Vi blev enige om at DistanceTo´s funktion skulle spredes ud på flere kontrollere, da vi gerne ville undgå en *bloated controller*. En bloated controller er en controller, der

[7] Se bilag 1
[8] Dette uddybes under afsnittet Implementering af koden.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

varetager for mange urelaterede områder uden at uddelegere ansvaret.[9] Løsningen blev to typer af controllere, én til styring af kollision, og én til bevægelse. For at opnå en højere samhørighed valgte vi at adskille de to typer og fordelte ansvaret ud på fire forskellige controllere. Dette resulterede i en enemyMovementController og enemyCollisionController, altså de systeminterne operationer, der styrer enemyklassernes handlinger, samt overvåger, hvor i koordinatsystemet de befinder sig. Til den interaktive del oprettede vi heroCollisionController, der overvåger heltens interaktion med andre elementer og en heroMovementController, der håndterer heltens bevægelser ved at foretage kald ned i modellen. Vi valgte at flytte keyevents fra GameControlleren ned i heroMovementControlleren, da det ville være mere samhørigt at lade denne varetage styringen af vores helt. Yderligere ville GameControlleren også få ansvaret for at oprette alle de nye controllere, og vi mente dette var en bedre løsning.

Implementering af controllere

Den første controller vi implementerede var heroCollisionController. Denne klasse skulle som nævnt sørge for at vores helt kunne interagere med andre objekter, hvilket var en af de helt grundlæggende funktioner i vores spil. Controlleren skulle hente Heros koordinater og sammenligne dem med de koordninater Level indeholdte. Level havde den nødvendige information, da den havde fået tildelt alle de Structure- og Enemyobjekter som den specifikke Level skulle have, fra levelController. Igennem metoden calculateDistance kunne HeroCollisionControlleren, ved at foretage metodekald til Level, sammenligne Hero objektet med alle de andre objekter i den specifikke Level. Hvordan dette fungerer i praksis vil blive uddybet i afsnittet ”Implementering af koden”.

Funktionalitet fra vores heroCollisionController overførte vi til enemyCollisionController så Enemy objekterne også kunne gå på platforme og stoppe ved vægge. I princippet var disse CollisionControllere meget ens, idet koden havde den samme logik. Opdelingen gav os dog mulighed for, senere i processen, at tilføje de specielle objekter (items) som kun Hero skulle kollidere med. Denne opdeling af specifikke controllere bidrog til en høj samhørighed i controller-laget og sikrede også en lav kobling i modellen, der via strukturen flytter den nødvendige kobling imellem klasserne ud af modellen og op i controllere-laget. Her repræsenterede hver controller et afgrænset og samhørigt område, hvilket gjorde det meget nemt at foretage ændringer i en enkel klasse eller controller, uden at det havde indflydelse på resten af systemet.

Det skal her nævnes, at ud fra GRASP principperne, er en controller det første lag, efter GUI´en der modtager og koordinerer brugerinputs. Det ville derfor være mere korrekt at have kaldt nogle af vores controllers for Handlers eller Coordinators, da de kan opfattes som *use-case controlleres*, idet de

[9] Larman (2005): s. 311

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

behandler systeminterne operationer frem for bruger inputs.[10] I vores forsøg på at undgå en bloated controller videreførtes dette begreb en anelse for langt og vi overså denne lille begrebsmæssige detalje.

Udvidelse af Modellen

Som beskrevet i 1. Iteration havde vi på dette tidspunkt implementeret en gravity-effekt, som alle Characters skulle være påvirket af. Denne kode var placeret forkert i forhold til MVC og den fungerede heller ikke i overensstemmelse med collisionsControllerne. Der måtte derfor foretages en mere simpel implementering. Dette resulterede i at gravity-funktionen, nu blot trak en Character en pixel nedad af gangen, i stedet for en eksponentiel gravity. Samtidigt blev gravity-effekten placeret som en metode i Character, hvilket muliggjorde at subklasser ligeledes fik denne påvirkning.

I og med controllerne fik mere funktionalitet kunne vi begynde at koble dem med modellen. Vi begyndte med at udvide vores Character klasse med den funktionalitet, der nu kunne varetages af collision- og movementControlleren. Booleans som cantGoRight, cantGoLeft sørgede for at fjenderne skifter retning når de enten rammer en væg eller enden af en structure, og sørger samtidigt for at Hero ikke kan penetrere vægge og platforme. På samme måde fungerer isInAir, der aktiverer vores gravity og sørger for at helten og fjender falder ned, hvis de ikke står på en platform.

I vores test-GUI kunne vi nu hoppe rundt med vores Hero, samt lande og blive stående på Structures, der fungerede både som platforme og en slags vægge, der sørgede for at afgrænse det område han kunne bevæge sig i. Da dette var implementeret kunne vi begynde at lege med gameplayet i forhold til hvordan vores forskellige baner skulle bygges.

Afslutning på 2.iteration

Ved afslutningen af 2. iteration havde vi opfyldt mange af de ambitioner vi havde haft for denne fase. Det var lykkedes at implementere en form for MVC, der ikke var helt i tråd med teorien, men som stadig sørgede for en adskillelse imellem lagene. Vi havde med vores controllere skabt et grundlag, hvorpå vi kunne øge funktionaliteten i hele systemet og samtidigt sikre en god fordeling af ansvar ud fra GRASP principperne. Vi så derfor frem til at komme i gang med 3. Iteration, hvor vi blandt andet planlagde at fokusere på implementering af skud, forskellige slags fjender og items. Disse blev også vurderet som risiko-områder. Vi vidste, at vi kunne gøre brug af mange af de valg vi havde foretaget tidligere, og på den måde genbruge både ansvarsfordeling og specifik kode.

[10] Larman (2005): s. 302

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

3.Iteration

Den 13. maj startede vi 3. Iteration. Vores risiko-område til denne del af processen var som tidligere nævnt implementering af Shot, flere forskellige slags fjender, samt items. Vores spil manglede også flere baner og winning conditions. Det var disse dele vi tog fat på med fokus på shot og enemyAI, som vi mente var de to sværeste opgaver, både med hensyn til kode og ansvarsfordeling.

Shot

Vi diskuterede i lang tid hvem der skulle have ansvaret for at oprette et Shot. Vi fokuserede i denne sammenhæng på, hvordan heltens skud skulle opstå. Vi havde fra starten besluttet at hero skulle skyde når brugeren trykkede på space. Dette illustreres i en SD.[11] Vi ville dog også gerne have fjender, der kunne skyde, og disse skud skulle ikke være styret af brugeren.

Ud fra denne opfattelse blev vi enige om at give Character-klassen en makeShot metode. På den måde kunne Enemy også få mulighed for at skyde, idet den nedarvede fra Character. Metoden makeshot kunne oprette et Shot-objekt, som skulle starte ved siden af en Character og bevæge sig enten til højre eller venstre i banen, afhængig af hvilken vej Characteren vendte.

Vi tilføjede knappen space til keyevents i HeroMovementControlleren, som styrede heltens andre keyevents. Når der blev trykket på space kørtes metoden Makeshot(), som Hero nedarvede fra Character. Vi bestemte at hver level skulle have en liste af heroshots hvortil HeroMovementController tilføjer et shot, når det blev oprettet af en Hero. Dette var i tråd med den liste af Structures vi allerede havde til hver level, og de andre lister af forskellige elementer vi havde tænkt level skulle indeholde.

Til at starte og stoppe bevægelsen for shots oprettede vi en ShotMovementController. Til at checke på kollision mellem Shot og Structure oprettede vi en Shotcontroller med metoden calculateDistance. De to nye controllere var i tråd med oprettelsen af flere og mere specifikke controllere i 2.Iteration, hvor ansvaret blev delt ud for at opnå høj samhørighed indenfor controller-klasserne. Herved undgik vi, som tidligere nævnt, det Larman beskriver som en bloated controller.

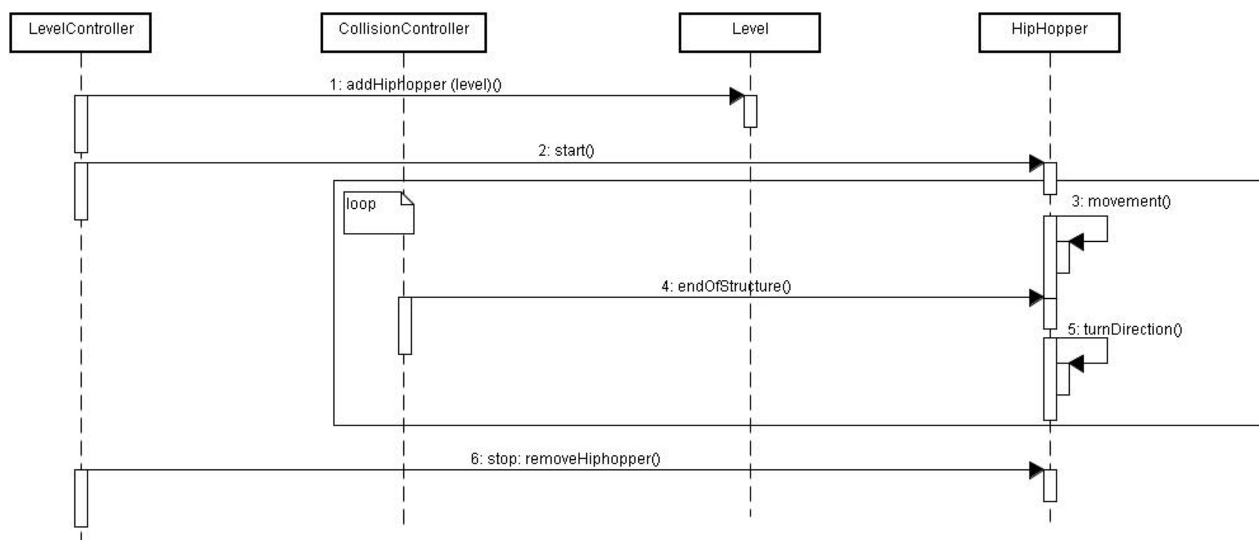
Enemy AI

Vi lagde efterfølgende fokus på den kunstige intelligens som fjender skulle have i spillet. Vores ønske var at fjender i spillet skulle bevæge sig frem og tilbage på en Structure. De skulle altså vende om når de ramte en væg eller enden af en Structure. Vi havde tidligere implementeret nogle fjender, der bevægede sig mellem to definerede xy-koordinater. Denne funktion var dog ikke særlig generel, og vi besluttede derfor at lave en ny use-case over en fjendes bevægelser. Use-casen blev lavet over en hiphopper, men kom senere til at

[11] Se bilag 4

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

være mere generelt gældende for klassen Enemy. Efterfølgende producerede vi denne SD, som illustrerer use-casen.



Figur 4

Vi besluttede at fjender selv skulle vide hvordan de skulle bevæge sig, og vi tilføjede derfor metoden enemyAI til klassen Enemy. De skulle dog ikke selv starte deres bevægelser eller holde styr på deres kollisioner. Da vi producerede den overstående SD, mente vi at levelControlleren kunne starte fjenders bevægelser, mens en CollisionControlleren kunne checke, hvor fjender befandt sig i level, og give besked til dem, når de ramte kanten af en Structure. På den måde opretholdte vi adskillelsen af controllere og model, og bibeholdt høj samhørighed.

EnemyAI metoden får en Enemy til at gå, enten til højre eller til venstre, på den Structure den lander eller befinder sig på. Denne metode kunne vi senere overskrive og gøre mere specifikke i Enemys tre subklasser Dog, Hiphopper og Boss. Som beskrevet havde vi i første omgang tænkt at levelcontrolleren kunne sætte fjenders bevægelser i gang. Men for at opnå høj samhørighed i controllerklasserne implementerede vi den tidligere omtalte EnemyMovementController, som skulle starte EnemyAI for alle tilføjede fjender i en level. Dette ansvar mente vi ikke at fjender selv skulle have, idet de ikke bør vide hvilken level de tilhørte. I EnemyCollisionControlleren tilføjede vi en ny funktion i dens calculateDistance metode. Funktionen skulle holde styr på hvornår hver Enemy var for enden af en Structure og ændrede den boolean, der bestemte hvilken retning de skulle gå[12]. Ved at fordele dette ansvar ud til en controller undgik vi, at der opstod for høj kobling mellem elementerne i modellen.

[12] Funktionaliteten uddybes i afsnittet enemyAI() under Implementeringen af Koden.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Som tidligere beskrevet kunne vi ved at bruge polymorphisme lave specifikke AI metoder for hver enemy. Vi besluttede at Dog skulle bevæge sig som defineret i Enemy-klassen, altså blot frem og tilbage på en Structure. Hiphopperens AI ville vi dog gerne udvide så han også kunne hoppe og skyde. Disse funktioner kunne vi nu tildele i hiphopperklassens enemyAI metode.

Enemy shot

Fordi at Enemy, ligesom Hero, nedarvede fra Character, havde de også mulighed for at bruge metoden makeshot. Vi kunne derfor implementere en metode i klassen Enemy, som hedder EnemyShot(). Denne metode overskrev vi ved hjælp af polymorphism i Enemys tre subklasser Dog, Hiphopper og Boss. I Enemyshot() brugte vi metoden makeshot fra Character til at oprette et Shot. Idet vi ikke ønskede at Dog skulle skyde, lod vi dens enemyshot returnere null. Ansvaret for, hvornår fjender skulle bruge metoden, og derved skyde, gav vi til EnemyMovementController. Det skyldes at netop denne controller i forvejen startede enemies AI-metoder og derved var information expert på, hvornår en Enemy skulle agere. Character fik samtidig implementeret parameteret attackPower i sin konstruktør, så Hero, Hiphopper og Boss kunne give varierende skade med hver deres makeshot-metode.

HeroCollisionController fik nu ansvar for at checke på kollision mellem Hero og Enemyshots og EnemyCollisionController fik ansvar for at checke om en Enemy kolliderede med det tidligere omtalte heroshot. På den måde blev ansvaret fordelt med den højeste samhørighed.

Winning Conditions

Vi havde gennem længere tid diskuteret, hvordan en bane skulle gennemføres. I 3. Iteration blev vi dog enige om at hero skulle styres til enden af Level, altså hen på et bestemt xy-koordinat. Vi tilføjede derfor en speciel Structure til hver Level, som sender hero videre til næste Level i det øjeblik han kolliderer med den. Informationen om denne Structure eller "winning condition" gives med i Levels konstruktør og er derfor specifik for hver Level. Vi tilføjede samtidig et specifikt xy-koordinatsæt til Levels konstruktør, som angiver hvor hero starter i hver Level.

CalculateDistance metoden i HeroCollisionControlleren tog på dette tidspunkt højde for alle de mulige kollisioner Hero kunne have. Den fik derfor også ansvaret for at tjekke om hero kolliderede med winning condition. LevelControlleren fik på samme tidspunkt en ArrayList af Levels. Vi oprettede også en ny metode i levelController, changeLevel(), som sætter attributten currentLevel til den næste i listen, i tilfælde af at HeroCollisionControlleren giver besked om kollision mellem hero og winning condition. Vi mener at ansvaret på denne måde, blev fordelt så vi opnåede den højst mulige samhørighed indenfor hver controllerklasser.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Items

Vi havde hele tiden haft en idé om en klasse, der hed Item som nedarvede fra Element. Vores powerups, point og shot skulle så nedarve fra Item. Denne idé lavede vi imidlertid om, da vi ikke fandt det nødvendigt at have klassen Item. I stedet satte vi Whisky, Cigarrettes og Dollars til at nedarve direkte fra Element ligesom Character og Shot gjorde det i forvejen. Whisky og Cigarrettes fik begge en healthværdi og en pointværdi. Klassen Dollars fik kun en pointværdi, som skulle bruges i forbindelse med HighScore. Objekterne fra de tre klasser skulle oprettes af LevelControlleren, som også bestemte deres placering i hver Level. Fordi at det kun skulle være Hero, der kunne tage disse powerups og point, placerede vi ansvaret for at checke kollision mellem Hero og disse i HeroCollisionControlleren. Derved forblev HeroCollisionControlleren information expert på alle de kollisioner Hero havde mulighed for.

På dette tidspunkt gav vi Characterklassen en healthværdi som nyt parameter i dens konstruktør, og derved fik både Hero og Enemyklasserne denne værdi. Healthværdien kunne vise styrkeforholdet mellem de forskellige Characters og samtidig bruges til at regulere, hvor meget skade et Shot skulle give. Heroklassen fik implementeret metoder, der kunne øge en heros health og pointværdi når HeroCollisionControlleren gav om besked om kollision.

Boss AI

Da vi efterhånden havde implementeret mange af de krævede elementer i spillet kom vi til implementeringen af Bossen. Vi havde tidligere brugt lang tid på at diskutere, hvordan han skulle bevæge sig og hvor kompliceret hans AI skulle være. Disse tidligere beskrivelser og idéer samlede vi i en use-case, der helt konkret forklarede EnemyAI metoden i klassen Boss, samt hvordan denne arbejder sammen med EnemyCollisionController og EnemyMovementController. Use-casen gav et rigtig godt billede af hvad bossen skulle og hvordan ansvaret skulle fordeles. Derfor var den også til stor hjælp, da det kom til implementeringen. Vi har valgt at vise nogle dele fra use-casen, som forklarer vores overvejelser. Koden for bossklassens enemyAI og enemyShot vil blive uddybet senere i opgaven.

Stakeholder and interests:

- Levelcontrolleren opretter bossen og tildeler ham til en bestem level.
- EnemyCollisionControlleren skal kende bossen, og beregne hans kollisioner med structures og heltens skud.
- EnemyMovementController starter enemyAI() og enemyShot() for alle enemies og derfor bossens.

Preconditions:

Bossen er oprettet og placeret i en level(uden winning condition).

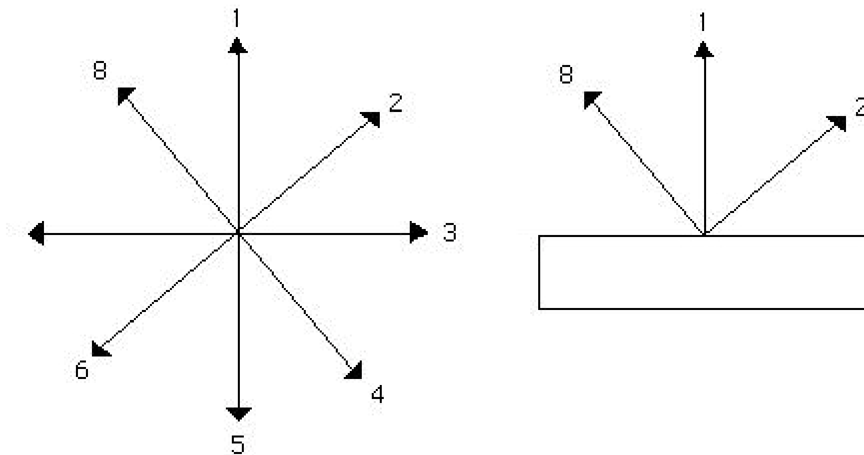
created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Succes Guarantee:

Bossen skal flyve random rundt, og skifte retning når han rammer en struktur, samtidig skal han skyde i begge retninger. Måden han skyder og flyver på styres af hans egen enemyAI og enemyshot metoder, samt EnemyMovementController og EnemyCollisionController.

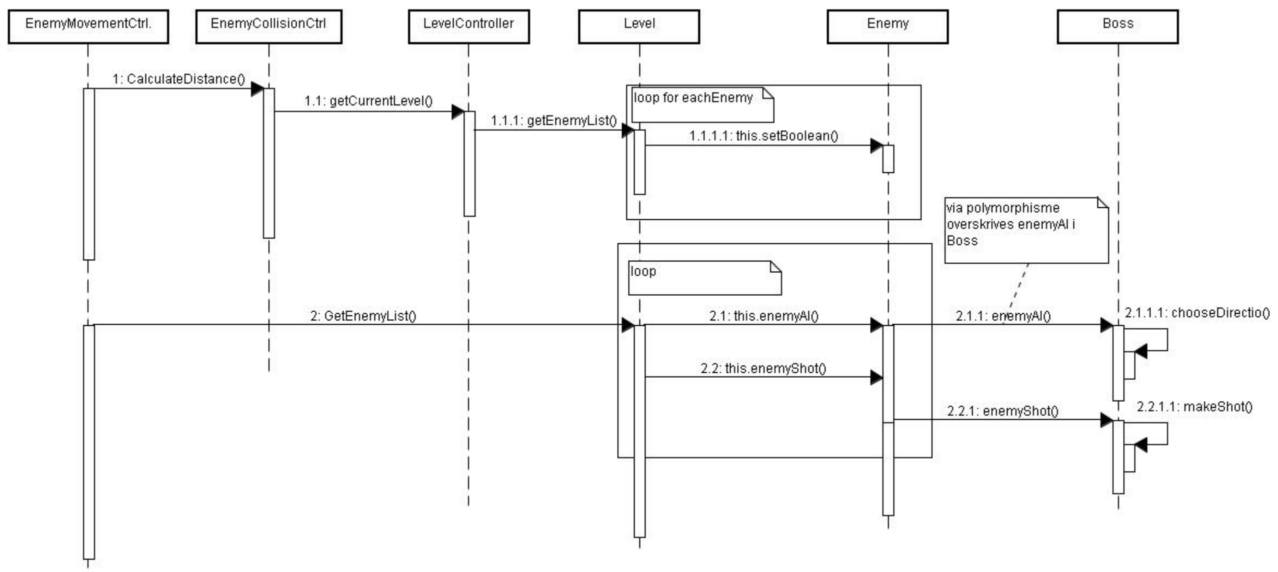
Main Succes Scenario:

1. Bossen oprettes af levelcontroller og bliver placeret i en level.
2. Bossen begynder at flyve når hero kommer til denne level. Hans metoder startes af enemyMovementControlleren, og han flyver i en tilfældig retning ud af 8 mulige
3. Han skyder til begge sider med et tilfældigt interval.
4. Når hero kommer ind i bossens synsfelt (horisontalt) skyder bossen ekstra hurtigt.
5. Når bossen rammer en struktur vil enemyCollisionController informere ham om at han skal skifte retning. Han tvinges til at vælge en af de 3 retninger der peger væk fra strukturen. (se illustration)
6. Bossen dør når han har mistet alt sit health og han bliver fjernet fra denne levels enemyliste. Bossen skal have health til at kunne klare omkring 20 skud fra hero.



Efter vi havde lavet use casen producerede vi en SD, som illustrerer de kald vi mente var nødvendige for at styre bossen.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image



Figur 5

Implementeringen af Boss-klassens metoder og fordelingen ansvaret for at kører dem, blev gjort en del nemmere med use-casen og SD'en. Boss skulle overskrive to metoder som den nedarvede fra Enemy, og ansvaret for at sætte dem i gang skulle placeres ved EnemyMovementController. EnemyCollisionController skulle tjekke på kollision mellem boss og Structure, og herefter sætte nogle booleans, der bestemte hvilken retning bossen kunne flyve. Boss-klassen blev gjort mere specifik end de andre fjender ved brug af polymorphism, og derved overskrivningen af enemyAI og enemyShot metoderne. Selvom den blev mere specifik mistede den ikke sin lave kobling til resten af modellen.

Vi implementerede samtidig en meget specifik boolean hos Bossen, bossIsAlive. Mens denne er lig true kører spillet og det stoppes når bossen dræbes, da booleanen dermed sættes lig false. Ansvaret for at sætte denne boolean lagde vi hos levelControlleren. Idet at spillet ender, når bossen dræbes valgte vi at den sidste bane, ikke skulle have nogen konkret winning condition, da målet er at dræbe Bossen og ikke at finde et bestemt sted i banen. Når først sidste bane var blevet til currentlevel, skulle det altså ikke være muligt at komme videre til en ny Level, hverken før eller efter man havde dræbt bossen.

Afslutning på 3. iteration

Da vi havde implementeret bossen opfyldte vi problemformuleringen for computerspillet, idet vi nu havde en helt, tre forskellige slags fjender (Dog, Hiphopper og Boss) og tre forskellige slags items (powerups, point og skud). Vi havde også implementeret flere levels, og winning conditions på disse. Vi havde arbejdet i en iterativ proces, hvor vi havde udviklet vores system gennem brug af UP artefakter og mange overvejelser angående mønstre og ansvarsfordeling.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Afslutningsvis gjorde vi spillet mere interessant, idet vi implementerede baggrundsmusik og guitarlyde når helten skyder. Samtidig implementerede vi start og slut paneler, samt paneler der under spillet viser heltens health, score og antal liv. Vi vil ikke gå i detaljer med disse implementeringer, da de ikke er sket ud fra nogen længerevarende overvejelser eller specifikke artefakter. Vi mener dog at disse implementeringer afrunder spillet og har gjort det mere fuldendt. Ved afslutning af 3. Iteration havde vi på denne måde opfyldt alle problemformulerings, og egne krav, til et fyldestgørende computerspil. På dette stadie havde vi samtidig lært en del om procesarbejde og implementering ved hjælp af UP artefakter.

Implementering af koden

Da det vil være alt for omfattende at beskrive hver enkelt linje kode, har vi i dette afsnit udvalgt dele af systemet, som uddyber tidligere emner i de andre afsnit, men også særlig speciel kode, som kræver en lidt mere dybdegående forklaring. For yderlig dokumentation og forklaring henvises til JavaDoc, som er vedlagt på CD.

Indledning

Som tidligere nævnt har vi fået inspiration til vores spil fra andre platformspil, såsom Supermario, Duke Nukem etc. Men at gå fra artefakter, design og UML-diagrammer forløber ikke altid uden fejl, og det bliver nødvendigt med nogle teknikker for at få produceret noget ordentlig kode. Derudover følte vi også det var vigtigt at have noget grafisk at forholde sig til i starten - en form for mock-up, som uden at opfylde nogle specifikke krav, kunne give os et indblik og inspiration til det rigtige spil. Prototypen var baseret på et simpelt stykke kode fra Lewis' Javabog, der skulle illustrere brugen af KeyListener. I stedet for billeder af piletaster, erstattede vi billederne med en rockstjerne. Udbyttet blev for os et indblik i keyEvents, og inspiration til styring i modellen. Men hvor skal man starte i koden? Vi vidste på forhånd at spillet ville komme til at indeholde mange klasser, og at det derfor var nødvendigt at finde ind til kernen i spillet, og få kodet dele af denne. Som vi også vurderede ved afslutningen af inception var det største risikoområde vores XY-styring, og da denne også var en del af kernen, var det naturligt at starte her[13]. Først vil vi kort præsenterer en af de teknikker vi anvendte i forbindelse med udarbejdelse af koden, nemlig elementer fra eXtreme Programming - testdriven development og PairProgramming.

XP: Test-driven development & Pair-programming

Som programmører er det nødvendigt at have nogle redskaber for at producere effektiv og præcis kode. eXtreme Programming (XP), er et af disse, og vi har brugt dele af de elementer det består af. Det ultimative mål med programmering er kode, der virker. En måde at opnå dette på er ved brug af test-driven

[13] Se figur 1.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

development[14] (TDD).

"Clean code that works is the mantra of test-driven development"[15]

Vi har tidligere haft dårlige erfaringer med kodeimplementering, som ikke var testet, og vi vidste på forhånd at det ville blive nødvendigt grundet størrelsen af vores projekt. Vi ønskede en pålidelig kode, der kunne arbejdes videre med gennem iterationerne. Ud fra vores manglende erfaring med værktøjet JUnit vurderede vi at det var for risikabelt at anvende det. Vi ville derimod bruge print-statments de steder, hvor der skulle testes. Vi har gennem alle iterationerne anvendt TDD. Eksempelvis anvendte vi TDD i forbindelse med implementeringen af vores Boss[16] i systemet. Ud fra en fyldestgørende use-case, samt SD, blev produktionskoden[17] hurtigt indtastet, og systemet var klar til første test. Vi oprettede en boss i levelController og tilføjede ham til dens enemyList, hvorefter vi startede programmet. Der skete en fejl, da Bossen kolliderede med en Structure. Han fløj i den forkerte retning og forsvandt dermed ud af skærmbilledet. Derfor startede vi en test-case, og vi ville ved hjælp af udelukkelsesmetoden, samt print-statements finde fejlen. Bossen var synlig, så hans nedarvning fra henholdsvis Enemy, Character og Elements var succesfuld. Han bevægede sig også, så brugen af polymorphism i form af enemyAI() måtte også være succesfuld. Metoden enemyAI() skulle tilfældig vælge mellem 3 retninger afhængig af, hvilken Structure bossen kolliderede med, og fejlen måtte muligvis findes her. Vi indsatte derfor print-statements i de fire muligheder han havde for at kollidere med Structure. Det viste sig, at der her ikke var nogen fejl. Den retning, der ville blive valgt efter kollisionen skulle sendes over i en metode, hvor bossen skulle flyttes ud fra. Her fandt vi fejlen, som bestod i, at der var blevet byttet om på + og - i forhold til y-aksen. Hvis Bossen skulle flyttes nord-vest, skulle han flyttes +1 af x-aksen, og -1 på y-aksen og ikke +1 på y-aksen, som vi havde gjort. Dette blev rettet, hvorefter bossen fungerede optimalt. I dette eksempel bestod testkoden blot af et par print-statements, men deres funktion var yderst brugbar.

En anden måde vi observerede fejl på, var ved brug af et andet element fra XP. Her brugte vi primært det såkaldte *Pair Programming*[18], som består i at koden bliver udført med fire øjne, altså to og to. De største fordele ved at sidde to og to, var at alle gruppens medlemmer fik indblik i koden, samt udviklingen heraf, men der blev også observeret væsentlige flere fejl hurtigere, som dermed kunne rettes førend systemet udsendte fejlbeskeder. Der blev heller ikke skrevet uforståelig kode, da Co-pilot hurtigt

[14] Teknikken er oprindeligt opfundet af Kent Beck og Ward Cunningham, men vi tager udgangspunkt Henrik Bærbak Christensens syn.
[15] Christensen (2006): s.9
[16] Koden vedrørende boss-klassen vil blive uddybet senere.
[17] Produktionskode: Den kode, der skal opfylde kravene til den enkelte opgave.
[18] Palmer & Steinberg (2004): s.35-47: Pair-programming består af en Driver, som sidder ved tastaturet, samt en Copilot, der sidder ved siden af og observerer.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

kunne se, hvis det ikke gav mening. Desuden hjalp det os til at holde fokus længere tid af gangen uden at blive distraheret af omgivelserne.

Lav kobling og høj samhørighed

At skulle kode ud fra disse to GRASP-mønstre, er mere eller mindre bare at kode ud fra ens sunde fornuft og logik. I forbindelse med objekt orienterede programmering, hvor begreber og fænomener fra den virkelige verden bliver anvendt mere præcist og bliver opdelt i klasser, skaber dette også krav om større samhørighed og lav kobling. I et system, som vores bestående af ca. 30 klasser, ville det være ufattelig svært at overskue, hvis der lå funktioner i en klasse, som havde lav samhørighed og samtidigt var højt koblede til andre klasser.

MVC

Som tidligere nævnt valgte vi allerede fra start af at overholde Model-View-Controller princippet i forhold til vores spil. Dette har givet et stort overblik i forhold til koden ved at den nu er struktureret i de såkaldte Java Packages. Ved at overholde MVC, kan vi også få spillet til at fungere uden en GUI. Der vil i de tre næste afsnit blive uddybet kode elementer fra MVCs enkelte dele.

Model

Modellen skal ud fra det objektorienterede perspektiv bestå af alle de begreber der kan findes ud fra beskrivelsen af opgaven. Med det menes alle de ting fra den ”virkelige” verden i forhold til spillets historie og elementer. Modellen endte i slutningen af 3.iteration med at bestå af 16 klasser, hvor de vigtigste elementer vil blive uddybet i følgende afsnit.

X-Y-styring

Implementeringen af XY-styringen fandt sted til en workshop, hvor dette, ud fra risikovurderingen, var fokusområdet. Der blev udarbejdet en artefakt i form af en mock-up på et whiteboard, som var med til at danne grundlaget for implementeringen af koden.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Figur 6

Vi startede ud med at kode klassen Elements, som i teorien ikke består af andet ind fire instans variabler, nemlig Integers og en tilhørende ArrayList - coordinateSet, bestående af typen Integers. Vi valgte en liste af typen ArrayList, da den er nemmere at arbejde med, og da der allerede er prædefinerede metoder til indsættelse af værdier. De fire Integers er X og Y koordinat samt en længe og bredde[19]. Metoderne i klassen Elements er ikke andet end getters og setters. Man kan vælge at skifte hele koordinatsættet, eller blot bruge moveXY(), hvis et Element skal flyttes uden at dens længde og bredde skal berøres. Herefter implementerede vi en midlertidig GUI og testede om klassen fungerede. Testresultatet var en firkant der kunne bevæge sig på skærmen, og der var nu lagt et fundament, som andre klasser kunne nedarve fra.

Level

Level blev implementeret som noget af det sidste i 1.iteration. Level skulle indeholde alt det man kunne se og ligesom danne rammen om spillet. En Level skulle altså kunne indeholde fjender, Structure, Hero, Shot og items. Ud fra klasse-digrammet blev dette hurtigt implementeret i form af ArrayLister med typen af de elementer, som skulle være i vores level. Fx en ArrayList<Enemy>, hvor man kan tilføje og fjerne fjender. Som led i 3.iteration, og et stykke nærmere det endelig mål, endte level med at få en liste både til hete-skud og en til fjende-skud. Derudover indsatte vi en winning-structure, som skulle være målet med den enkelte level. Samtidigt oprettede vi et heroStartX og Y koordinat til hver level, som skulle være det sted i banen hvor helten startede første gang han kom ind i banen, men også når han mistede et liv.

[19] Vi havde i starten valgt at bruge to X koordinater og to Y koordinater (x1,x2,y1,y2), men pga. metoden fillRect() i GUI'en valgte vi at bruge (x,y,xLen,yLen).

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Nedarving

Nedarving er et stærkt værktøj indenfor kode. Det er vigtigt kun at bruge nedarving, hvor det er relevant, men også logisk. Der er tale om generalisering og specialisering. Generalisering er når man kan samle nogle fællestræk fra nogle klasser til en meget overordnet og abstrakt klasse. Specialisering er når man går den anden vej, altså har nogle begreber, som opfører sig mere specifikt. Man kan altid tjekke om man bruger nedarving korrekt ved at sige, at specialiseringen er en slags af det generelle. Fx i vores spil, er en Hero en slags Character. Som det kan ses i vores første klasse-diagram, har vi brugt nedarving rigtig meget. Den største fordel er, at der er sikret ens metodekald på alle elementer i spillet i forbindelse med XY-styringen, men det åbner også en masse muligheder, nemlig at kunne flytte Structures og Items. Nedenfor ses et kode-eksempel, hvor klassen Whiskey nedarver fra Elements. Elements konstruktør bliver kørt, som metoden super(), som skal bruge de samme parametre som Elements selv.

```
public class Whiskey extends Elements{

        private ImageIcon currentImage;
        private int health,score;

        public Whiskey(int x, int y, int xLen, int yLen) {
                super(x, y, xLen, yLen);
                this.health=20;
                this.score=5;
                currentImage = new ImageIcon("gfx/whiskey.gif");
        }(…)
```

Nedarving ses igen fra Character og ned til Hero og Enemy. Dette blev valgt ud fra at helten og fjenderne skulle være udsat for samme miljø (Gravity, Health, etc). Der blev også defineret nogle metoder walkRight() og walktLeft(), som sikrede at alle Characters kan bevæges. Derudover har de en masse booleans, der beskriver i hvilken tilstand de er i. Fx boolean'en isInAir, som beskriver om en Character er i luften eller ej. Disse booleans bruges i forbindelse med vores CollisionsControllere, enemyAI(), og keyListener til styring af helten. Nedarving muliggør også polymorphism, som betyder at forskellige klasser kan opføre sig på en specifik måde i samme metode-kald.

Hero

Hero nedarver både fra Elements og Character. Hero-klassen var i starten ikke særlig speciel, og har først udviklet sig i slutningen af 3.iteration, hvor der er blevet implementeret liv, health, score og grafik til spillet. Det er Hero-klassen, som holder styr på scoren i spillet. Derudover har den implementeret interfacet

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Comparable. I de medfølgende metoder er compareTo() kommet til at afhænge af scoren, da denne skal sorteres i forbindelse med highScoren.

Helten bruger polymorphism i forbindelse med reducehealth() fra Character. Når en helt er nede på 0 i liv, skal der nemlig kun køres removeLife(), modsat fjender, som er fuldstændig døde, når de når 0 i health.

Heltens grafik bliver oprettet i konstruktøren, og metoden getCurrentImage kan virke lidt avanceret, da helten indeholder 8 billeder. Først tjekkes der på heltes retning – isDirection(), som peger til højre hvis true, og venstre hvis false. Hvis helten er på vej op i sit hop (hasJumped==true) skal et billede vises, hvor han hopper. Hvis helten er på vej ned og stadig er i luften (hasJumped==false && isInAir==true) skal billedet, hvor helten er på vej ned vises. Når helten står stille (isStanding==true) skal hans stilbilledet vises, og hvis denne er false, skal hans animation, som viser at han går til højre eller venstre sættes til currentImage.

Enemy

Selve Enemy-klassen er ikke særlig kompleks, da denne igen er en generalisering af vores fjender, henholdsvis Dog, Hiphopper og Boss. Den blev først for alvor implementeret i starten af 3.iteration, men der var dog lavet en slags prototype-fjende[20], der kunne illustrere brugen af fjender i spillet i 2.iteration. Klassen har til opgave at præsentere to nye booleans i forbindelse med fjendernes kunstige intelligens. noSpaceLeft og noSpaceRight. Disse booleans opstod i forbindelse med en use-case over enemyAI(), der beskrev fjender, som patruljerende en Structure. Vi bestemte at en Dog kun kunne gå på en struktur, hvorimod en Hiphopper også kunne både hoppe og skyde. Boss'en skulle være en del anderledes, da visionen ønskede en flyvende Boss, som kunne skyde med en vis intelligens. For at undgå at skulle oprette specielle lister til en Hiphopper og Boss, valgte vi at bruge polymorphism i forbindelse med skud og AI.

enemyAI() og enemyShot()

Disse to metoder blev lavet ud fra use-cases i 3.iteration. Vi ville helst undgå at lave specifikke lister til Boss og Hiphopper. Vi definerede derfor enemyAI() i Enemy-klassen, for på den måde at kunne overskrive dem i de enkelte subklasser. Metoden enemyAI() i Enemy-klassen blev kodet ud fra use-casen om netop denne. Som tidligere nævnt havde vi oprettet to booleans noSpaceLeft og -Rigt. Disse blev brugt i forbindelse med enemyCollisionController, som satte noSpaceLeft til true, hvis fx en fjendes X-koordinat var lig en Structures X-koordinat. Ved hjælp af disse kunne der kodes en simpel AI, der fik fjenden til at skifte retning når han enten ramte siden af en Structure, eller var nået til enden af denne.

[20] Med dette udtryk menes, at der blev kodet en enkelt, på ingen måde generel fjende i forbindelse med delafleveringen.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

```
public void enemyAI(){
        if (isNoSpaceLeft() || isCantGoLeft()){
            setDirection(true);
        }
        else if (isNoSpaceRight() || isCantGoRight()){
            setDirection(false);
        }
        if (isDirection()){
            this.walkRight();
        }
        else {
            this.walkLeft();
        }
    }
```

Hiphopperen skulle egentlig opføre sig på samme måde som hunden, bortset fra at en hiphopper skulle hoppe en gang imellem. Hunden behøves altså ikke at overskrive metoden, hvilket Hiphopper og Boss gør.

Metoden enemyShot(), blev implementeret i Enemy, selvom Dog ikke skulle have denne egenskab. Metoden returnerer et Shot-objekt, men dette er blot null, så når listen af fjender løbes igennem og der laves et enemyShot() ved fx Dog, er dette blot null-objekt. Ved Hiphopper og Boss er disse Shot-objekter, som bliver tilføjet til enemyShotList i den aktuelle level. Det er igen polymorphism, som gør dette muligt. Metoden blev desuden udvidet, da Boss blev kodet, hvilket bliver uddybet i næste afsnit.

Boss

Vi havde lavet en ekstra grundig use-case og SD på Boss, da vi gerne ville slutte spillet af med manér. Det specielle ved Boss i forhold til helten og fjenderne, var at Bossen kunne flyve, og måtte af den grund ikke være påvirket af vores gravity. Dette problem blev løst ved at lave en ekstra boolean i Character-klassen (affectedByGravity), som blev tjekket i Characters metode activateGravity(). Hvis boolean'en var false ville denne Character ikke blive hevet nedad. Bossen skulle have 8 retninger han kunne bevæge sig i[21]. Disse bliver alle oprettet i metoden chooseDirection(String direction), som tager en streng med, som så vælger den retning man har givet med. Det er her metoden moveXY(), som er nedarvet helt oppefra Elements, der muliggør disse retninger. Det, som gør Bossen speciel, er hans tilfældige movement. Derfor importerer vi Random-modulet fra Javas bibliotek. Derudover laves en ArrayList med strenge, og der tilføjes de 8

[21] De 8 retninger: N, NE, E, SE, S, SW, W, NW.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

retninger. I enemyAI() vælges der tilfældigt mellem tre retninger afhængig af hvilken flade han har ramt[22]. Nedenfor ses de to metoder, som viser hvordan enemyAI(), tilfældigt vælger en blandt tre mulige retninger, som bliver gemt i variablen currentDirection. Bagefter køres metoden chooseDirection(), hvor currentDirection gives med som parameter, og bossen flyttes så i den aktuelle retning.

```
public void chooseDirection(String direction) {
                if (direction.equals("n")) {
                        this.moveXY(0, -1);
                }
                if (direction.equals("ne")) {
                        this.moveXY(1, -1);
                } (…………)
```

```
public void enemyAI() {
        if (isCantGoUp()) {
                currentDirection = directions.get(generator.nextInt(3) + 3);

        }
        if (isCantGoLeft()) {
                currentDirection = directions.get(generator.nextInt(3) + 1);
        }
        if (isCantGoRight()) {
                currentDirection = directions.get(generator.nextInt(3) + 5);
        }
        if (isInAir() == false) {
                int i = generator.nextInt(3) + 0;
                if (i == 2) {
                        i = 7;
                }
                currentDirection = directions.get(i);
        }
        chooseDirection(currentDirection);
}
```

Bossens skud skulle også være specielt. I forbindelse med Boss use-casen, var det oppe at vende, at det måske blev nødvendigt med en bossCollisionController, som hele tiden skulle vide hvor helten var. Men fordi vi gerne ville have Boss til at optræde som en Enemy, vurderede vi, at det var at miste den høje samhørighed, og valgte i stedet at give Heroens koordinater med til metoden enemyShot(). Det betød ikke noget for Hiphopperens skud, men i Bossens enemyShot(), kunne vi opfylde kravet fra use-casen, om at han skulle skyde hvis Hero kom indenfor Bossens to Y-akser. Vi genbrugte noget kode fra

[22] Se use-case: Boss

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

HeroCollisionController for at tjekke, hvornår helten er indenfor farezonen. Hvis ikke helten er inden for fare-zonen vil Bossen skyde i en tilfældig retning, med et bestemt tidsinterval.

Controller

Som tidligere nævnt, fandt vi hurtigt ud af, at det var nødvendigt at fordele ansvaret ud til nogle use-case controllere og dermed undgå én bloated controller, som ville få for meget ansvar. I 2.iteration begyndte vi at implementere CollisionController til Hero og til sidst også til Enemy. I 3.iteration indså vi, for opretholdelse af den lave kobling og høje samhørighed, at Shot også måtte have sine egne controllere. Til sidst i 3.iteration bestod controller-delen af i alt 8 controllere.

LevelController

LevelController er en Creator i forhold til modellen. Den opretter som tidligere nævnt alt det, der skal være i spillets "verden", men har også et par enkelte metoder. Den vigtigste er metoden changeLevel(). Der er oprettet en ArrayList med levels, hvor de enkelte levels er tilføjet i den rigtige rækkefølge. Derudover er der oprettet en currentLevel variable af typen level, som hele tiden er den aktuelle level. Til at starte med vil den første level selvfølgelig være level 1. Metoden changeLevel(), bliver kørt det øjeblik, helten har kollision med en Levels winningPosition, og metoden sætter så den næste level i listen til at være currentLevel. Dernæst sættes Heroens X og Y koordinat til at være lig med Levels start koordinater for Heroen. Det er ligeledes igennem currentLevel, at resten af contollerne får adgang til den aktuelle Level.

HeroMovementController

HeroMovementController har ansvaret for styringen af Hero, og da han skal styres af en bruger, har vi af denne grund også valgt at placere funktionaliteten til en keyListener her. Vi har lavet en "inner class", der implementerer Interfacet keyListener, som har 3 metoder: keyPressed, keyReleased og keyTyped. Her skulle vi bruge de to første metoder til at tjekke, når der blev trykket på nogle bestemte taster og igen løftet fra dem. For at opnå, at vores helt flyttede sig kontinuerligt oprettede vi fire booleans: up, right, left og space. Disse bliver sat til true, hvis de bliver trykket ned og false når der bliver løftet igen. Disse booleans bliver fortolket nede i klassens run-metode[23]. Eksempelvis, hvis pilop tasten er blevet trykket ned:

```
if (getUp() == true && controller.getLevelController().getHero().isInAir() == false) {

        controller.getLevelController().getHero().setHasJumped(true);
}
```

[23] Der vil senere i afsnittet blive uddybet hvad hvad run() og Runnable bruges til.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Hvis booleanen up er lig true og vores helt ikke er i luften (han må ikke kunne oppe hvis han allerede er i luften), skal hans boolean HasJumped sættes til true. Den indkapslede klasse DirectionListener, kan desuden returneres, da den skal tilføjes til MainFrame, hvor al interaktionen mellem brugeren og computeren foregår, hvilke stemmer i overensstemmelse med MVC.

Kollision

I forbindelse med udviklingen af de første use-cases og klasse-diagram, fandt vi hurtigt ud af, at det var nødvendigt med en form for metode (distanceToObject) , som hele tiden vidste hvor alle Levels elementer befandt sig i forhold til hinanden. Til workshoppen sidst i 1.iteration, blev der også udviklet en meget specifik kode, der præcist tjekkede om Hero ramte de to Structures, som var blevet oprettet. På baggrund af dette stykke kode, var der grundlag for at oprette et generelt stykke kode, som kunne virke uafhængigt af antallet af Structures og deres koordinater. Med udgangspunktet i skitsen fra vores workshop, blev HeroCollisionController implementeret. Den eneste metode klassen fik, er calculateDistance(), som kender Heros koordinater, og currentLevel. Alle Levels lister løbes i gennem og der tjekkes ved if-sætninger om deres koordinater passer sammen. Her illustreres beregningen, hvis Hero rammer toppen af en Structure:

```
for (Structure s : currentLevel.getStaticList()) {

        if ((hero.getY() + hero.getYLen()) == s.getY() &&
        hero.getX() <= (s.getX() + s.getXLen() - 1) &&
        (hero.getX() + hero.getXLen()) >= (s.getX() + 1)) {

                hero.setInAir(false);
                break;

        } else {
                hero.setInAir(true);
        }
}
```

Som det ses, køres currentLevels statiske liste igennem via en foreach løkke. If-sætningen er så sand, hvis heltens fødder (hans y-koordinat + længden) er lig med en Structures top (y-koordianat). Der tjekkes altså på om deres koordinater er præcis ens. Dette medførte et problem, da vi i 1.iteration arbejdede på en

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

virkelighedstro gravity, hvor man falder hurtigere og hurtigere (dvs. flere og flere pixels per enkelt move). Dette kunne CollisionControlleren ikke tage højde for, og vi valgte derfor også at droppe den advancerede gravity i 2.iteration. Der tjekkes også om heltens venstres side (X-koordinat), er indenfor en Structures højre side (Structures x-koordinat + længden). Det samme gør sig gældende med Heros højre side. Med andre ord, hvis Hero står på toppen af en Structure, skal hans boolean isInAir sættes til false, hvorefter der laves et break, da han kun kan stå på én Structure af gangen. Uden et break, ville Hero falde igennem alle Structures. Vi anvendte her TDD, i form af print-statements, hvor vi printede Heros y-koordiant + længde, samt en Structures y-koordinat. Hvis ikke han ramte toppen af nogen Structure, må det jo betyde at Hero befandt sig i luften, og hans boolean blev sat til true. Udover at tjekke om han ramte Structures, tjekkede HeroCollisionController også om han ramte alle de andre elementer i en Level, altså fjender, items, fjendeskud og winningPosition.

Det er samme matematiske udregninger, der bruges i EnemyCollisionController og ShotCollisionController, så derfor vil disse ikke blive yderligere uddybet.

Timer-Runnable-Thread

I computerspil, som vores, er det meningen at spilles skal afvikles i en hastighed så menneskets øjne kan nå at percipere handlingerne. Det er altså nødvendigt at skabe et delay, da computeren uden problemer kan udregne millioner af beregninger i sekundet. Da vi før havde beskæftigede os med en Timer[24], valgte vi at bruge denne til at kontrollere spillets afvikling med. Umiddelbart kørte det problemfrit, men vi fik problemer med, at spillet blev afviklet for langsomt, selvom delayet var sat til det mindst mulige. Vi havde ikke vurderet dette som et risikoområde, da vi gik ud fra det godt kunne fungere med en timer. Vi fik dog inspiration fra andre grupper, samt Internettet om, at man kunne afvikle spillet ved hjælp af Threads. En Thread kan ses som en opgave som Java kan få udregnet. Hvis der oprettes flere Threads, det vil sige multiple-threads, skal computeren og styresystemet anvende multitaskning[25] for at løse opgaverne. En måde at anvende Threads på i Java, er ved at implementere interfacet Runnable. Det har en metode som hedder run(), som kan startes ved hjælp af en Thread. Derfor oprettes der en variable af typen Thread, som blandt andet har metoderne start(), stop(), resume() og pause(). Den kan så tage en klasse, der har implementeret Runnable med i sin konstruktør, og får opgaven at kontrollere metoden run(). Det vidste sig i 3.iteration, at vi fik flere problemer end forventet ved brugen af Threads, og for sent fokus på dette, har medført nogle fejl vi ikke kunne nå at løse indenfor vores tidsperiode. I afsnittet kodekritik, bliver emnet taget op igen, hvor der vil blive diskuteret en mulig løsning.

[24] Modul i Javax.swing biblioteket, som har samme funktion som et stopur. Du kan starte og stoppe det, men samtidig også styre hvor hurtigt det skal skifte til næste handling.
[25] Multitasking er når en single-core processer, skal udregne to eller flere opgaver samtidigt. Dette gøres ved hjælp af time-slices, hvor der skiftes mellem opgaverne, afhængig af prioritet og andre faktorer.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

View

View-delen eller vores GUI består af et MainFrame, som nedarver JFrame, samt et antal paneler, som alle nedarver JPanel. GUI'en indeholder ingen kompliceret logik, hvilket den heller ikke må i forhold til MVC-princippet. Dette illustreres også ved at vores spil kan fungere uden en GUI. Det er MainFrame, der opretter paneler, og har metoder til at skifte disse. Det er GamePanel, der er mest kompliceret, da den skal tegne alle spillets elementer i skærmbilledet på 800 gange 600 pixels. GamePanel får tilføjet en keyListener fra HeroMovementController, og den får den aktuelle Level fra GameControlleren. I metoden paintComponent() bliver alle levels lister løbet igennem ved hjælp af foreach løkker, hvor det er muligt at få deres X og Y koordinat, da alle ting i modellen nedarver fra Elements. Dette muliggør så, at vi kan tegne billeder (ImageIcon), ved brug af metoden getCurrentImage(), som kun skal bruge disse to koordinater. Derudover bliver helten og et baggrundsbillede også tegnet. Metoden bliver kørt hver gang der bliver repaintet. Dette ansvar har vi givet til gameController, som ved hjælp af en Thread gør dette med et hvis interval.

Kode-Kritik

Selvom spillet må synes at opfylde opgaveformuleringens krav, er der alligevel nogle elementer i koden, som har skaft problemer, der til dels har været med til at påvirke nogle beslutninger. Det største problem som vi stødte imod, var brugen af Threads. Vi havde ikke vurderet dette som et risiko-område, og vi kunne derfor efterfølgende se, at det nok havde krævet mere dybdegående studier for at kunne kontrollere disse. Fejlene kan dog ikke ses i GUI'en, og har heller ikke en noget alvorlig betydning for afviklingen af spillet. Vi har dog identificerede fejlen, og ved også hvorfor den opstår. Fejlen sker i forbindelse med vores Levels ArrayLister, hvorfra der løbende bliver fjernet og tilføjet elementer. På grund af brugen af multiple-threads, altså flere samtidige opgaver, gør dette at listerne ikke når at blive opdateret i forhold til hvem der har brugt dem. Fx fjerner shotCollisionController et skud fra listen, hvis det rammer siden af en Structure, hvilket kan medføre en fejl i GUI'en, hvor GamePanel, måske samtidigt er i gang med at løbe listen igennem for at tegne skuddene. Måden vi kunne løse problemet på, er ved hjælp af en ThreadPool. En ThreadPool kunne fx bruges til at kontrollere en gruppe af Threads. Med det menes, at en Thread ikke vil blive kørt førend en anden er blevet færdig. Eksempelvis kunne man bede en Thread om at vente, ved hjælp af metoden wait(). Derefter kunne denne så først køre den når den første Thread siger notify(). Dette ville sikre en mere stabil afhandling af vores spil. Det har dog ikke været muligt at implementere på grund af manglende viden om ThreadPools i Java.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Refleksion over brug artefaktbrug i forhold til kode

Som tidligere nævnt, er det umuligt at danne sig et overblik over et helt system, uden at have artefakter. Som programmør kan man godt have udtænkt mange gode idéer, og måske også få delelementer kodet uden brug af artefakter, men det ville aldrig kunne lade sig gøre at få et så stort system som vores. Den mest hjælpende artefakt, har uden tvivl været klasse-diagrammet, som allerede fra start gav et stort overblik over hele systemet. Derudover har risiko-vurderingerne været med til at danne et fokus i forhold til koden, og mindre komplekse dele såsom items, blev udskudt til de sidste iterationer. Use-cases og tilhørende SD'er, dannede grundlag for de mere specifikke systemelementer i spillet, såsom komplicerede metoder. Mange artefakter er også blevet tegnet på whiteboards og diverse mock-ups, har ligeledes hjulpet til en bedre forståelse og dermed en nemmere kode-udarbejdelse.

Grafik

Den grafisk side i et spil er, efter vores mening, et vigtigt element, da det skaber en stemmening og en stil, som gør det sjovere at spille. Vi ville gerne give det et tegneserie look, der passede til den historie vi gerne ville fortælle. Til vores spil er der oprettet 5 unikke baggrunde, 4 characteres, 3 items og 2 skud samt en intro og gameover splashscreen, i alt ca. 30 billeder og animationer. Til udviklingen har vi brugt Adobe Photoshop der tilbyder et væld af forskellige værktøjer der egner sig til formålet.

Vi konstruerede først banerne i Java, hvor vi kunne teste gameplayet ved at flytte rundt på Strucuture-objekter. Baggrundene er designet ovenpå et screenshot, hvorpå den endelige opbygning er illustreret. Ud fra disse screenshots kunne vi skabe den rigtige stemmening til hver Level. Dette viste sig at være en ret besværligt, og på ingen måde generel. Hvert baggrunds billede er tilpasset hver Levels opbygning og vi kunne derfor ikke ændre i koden uden også at ændre i baggrundsbilledet. Vi overvejede derfor at basere grafikken på et *protected variation mønster*, hvor dataen for en levels opbygning er gemt udenfor systemet. Vi havde set eksempler spil, der havde implementeret en klasse der kunne fortolke et tekstdokument, og derfra oprette en Level ved at oversætte bogstaver og tegn, til elementer i spillet. Hvert tegn ville være tilknyttet et billede, som så fortolkedes som et bestemt objekt ud fra typen af tegn og dets placering i tekstdokumentet. Denne løsning ville gøre det mindre besværligt at oprette mange forskellige Levels og dermed gøre systemet mere generelt i forhold til det grafiske. Vi valgte dog at beholde vores første løsning, da vi mente at individuelt designede billeder tilførte spillet en helt anden dimension, hvor vi kunne placere vores helt i en ghetto, beskidte kloaker osv. Dog muliggør måden vores spil er designet på ved hjælp af XY-koordinater, at der kunne oprettes en "GUI-opretter". Med det menes et simpelt Java program, hvori der kan tegnes bokse ved hjælp af drag-and-drop med musen. Det koordinat, hvorpå muse-knappen er blevet trykket ned, kunne blive X og Y koordinat i forhold til en Structure i vores spil. Når fingeren løftes fra knappen, bliver længden og bredden beregnede ud fra dette koordinat. Det vil på

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

samme måde være muligt at placere helten, fjender og items blot ved hjælp af et enkelt muse-klik. Når programmet lukkes ville det oprette en Level på baggrund af de indsamlede informationer, som efterfølgende vil kunne spilles.

Implementeringen af billeder er gennemgående for alle in-game objekter i vores spil. Alt vores grafik ligger i gfx packagefolder. Herfra bliver billederne hentet ind, når der oprettes et nyt objekt, via contructoren i objektets klasse. Som vist i eksemplet her:

```
Level level1 = createLevel1();
```

```
Level level1= new Level(this.hero,newImageIcon("gfx/back2.jpg"),700,500,50,50,50,500);
```
[26]

Karaktererne i vores spil er designet i Photoshop hvor hver bevægelse er gemt som et GIF billede. Derefter har vi lagt dem sammen i en GIF animation via AnimationShop. Dette eksempel viser hvordan hiphopperen oprettes med to forskellige animationer, og hvordan de skifter i forhold til den retning han bevæger sig i ved at returnere currentimage ud fra hans isDirection.

```
this.hipR = new ImageIcon("gfx/hipR.gif");
this.hipL = new ImageIcon("gfx/hipL.gif");
```

```
public ImageIcon getCurrentImage() {
        if (isDirection())
                currentImage = hipR;
        else
                currentImage = hipL;
        return currentImage;
```

Som tidligere nævnt nedarver alle in-game objekter et koordinatsæt fra Element klassen. Når et objekt oprettes, gives der i konstruktøren X-Y værdier, der definerer hvor i en Level objektet skal placeres. På denne måde kan vi konstruere hele spillets layout.

Da vores spil ikke er af en omfattende størrelse og derfor ikke kræver voldsomt meget grafik har vi kunne tage os visse friheder. Som nævnt ville protected variations have åbnet op for flere muligheder for at gøre spillet større og mere holdbart i længden. Vi kunne også have oprettet en imageBuffer, der indlæser

[26] Udover at definere baggrundsbilledet skal level også have en helt, samt en winningcondition, som får deres X-Y værdier med i metoden

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

billederne i hukommelsen inden de blev renderet til systemet. Det ville tillade en mere avanceret grafik uden at spillet ville lagge. Vi vil derfor lige fremhæve at det primært var funktionaliteten i vores system, der har været i fokus, og grafikken var det lille krydderi der satte programmeringen i perspektiv, hvor resultaterne fik en mere smagfuld form.

Refleksion over processen

Vi havde igennem semesteret fået præsenteret et omfattende teoretisk apparat til koordinering af processen og discipliner til struktureringen af systemet. Igennem øvelser havde vi gennemgået mange af elementerne i praksis, men nu skulle det hele sammensættes og gå op i en højere enhed, der skulle resultere i et velfungerende system og denne opgave.

Vi har i dette projekt forsøgt at inddrage relevante artefakter og begreber, der knytter sig til Unified Process. Vi mener dog, at vi kunne have udnyttet disse artefakter og teorierne bag dem bedre, hvis vi havde lagt større vægt på dem tidligere i processen. Vi følte, at udviklingen af artefakter var en smule kunstig, og derfor gjorde vi ikke så meget ud af dem. Et større fokus på dette ville have tilført den tidlige proces en større værdi. Til trods for denne iagttagelse, har vi som processen skred frem, oplevet at artefakterne gav os et større overblik og hjalp os med at udvikle vores system. Vi har oplevet, at brugen af artefakter, illustrerer systemet fra en anden vinkel, og derved gav en bedre forståelse og simplificerede struktureringen af systemet. Det gjorde det også nemmere at implementere konkret kode, efter vi havde diskuteret og illustreret specifikke problemstillinger ved hjælp af artefakter, såsom design diagrammer, use-cases og SD'er. Disse gav os løbende overblik og hjalp os med at fordele ansvar til vores forskellige systemklasser. Set i bakspejlet lagde vi for meget fokus på implementeringen af kode, hvilket er gået ud over vores brug og udvikling af artefakter, da vi sent i processen indså, den store værdi de kunne tilføre.

Den teoretiske inddragelse af GRASP-principperne, har skabt en forståelig indgangsvinkel til generelle programmeringsparadigmer. Vi har brugt dem som inspiration i processen til, hvordan vi kunne fordele ansvaret på bedst mulige måde, for at sikre et stabilt og generelt system. Vi mener at det er lykkedes tilfredsstillende at implementere lav kobling og høj samhørighed, som er to af grundbegreberne i GRASP. Efter 2. iteration havde vi struktureret koden ud fra designmønstrene, hvilket sikrede at det var forholdsvis problemfrit, at ændre og tilføje nye klasser i systemet. Dette betød at vi i 3. iteration havde meget nemt ved at implementerer den manglende kode, for at vores system skulle leve op til opgaveformuleringens krav.

Ved at dele processen op i mindre dele, som den iterative proces foreskriver, har vi opnået et godt overblik over opgaven. Opdelingen i iterationer skabte en rytme i vores forløb, hvor vigtige discipliner som risikovurdering tillod en timeboxing af den fremtidige fase og skabte derved en overskuelig plan. Dette bidrog til at holde vores fokus afgrænset til det væsentlige i netop den fase vi befandt os i.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Dermed undgik vi vandfaldsmodellen og sikrede os en dynamisk udvikling, hvor vi havde mulighed for at tilpasse os forandringer. Vi havde ingen erfaringer med udviklingen af systemer af denne størrelse, og vi tvivler på, at vi kunne have løst opgaven, hvis vi havde benyttet vandfaldsmodellen, hvor stor viden og overblik er krævet fra starten. Derimod har den iterative proces gjort det muligt at starte fra vores udgangspunkt, som uerfarne systemudviklere, og løbende i processen opbygge vores viden og erfaringer.

Arbejdsfordeling

I gruppen var der et grundlag for arbejdsfordeling, da nogle var mere praktisk orienterede, andre mere teoretiske. Derfor var det vigtigt for os, at vi alle blev involveret i alle dele af udviklingen. Det er vores opfattelse at det er sammenhængen imellem de forskellige områder, der er afgørende for forståelsen af processen, og det, der i sidste ende, sikrer skabelsen af et ordentligt produkt. For at sikre den fælles forståelse har vi forsøgt at fordele arbejdsopgaverne, hvor vi skiftedes til at programmere, udvikle UML, analysere og vurdere designet i forhold til de forskellige mønstre.

Der er visse elementer i spillet, som kunne være gjort anderledes. Som vi nævner under afsnittet grafik ville det have været en mere elegant løsning at implementere de grafiske elementer under protected variations princippet. Vi kunne have valgt at oprette Character og Enemy klasserne som Abstrakte klasser, da de ikke opretter objekter.

Men generelt er vi godt tilfredse med vores produkt, specielt vores minimale erfaringer taget i betragtning. Der har været ting som vi kunne have gjort bedre eller anderledes, men det vil der som regel altid være, og med tiden vil vi oparbejde mere erfaring, og vil derfor nemmere kunne se mulighederne tidligt i udviklingen.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Konklusion

Til denne eksamensopgave blev vi stillet den opgave at udvikle en simulation i form af et computerspil. Vi har ved at benytte Agile Unified Process og dennes beskrivelse af den iterative proces, skabt et produkt, der opfylder opgaveformuleringens krav.

Opdelingen af processen i iterationer har sikret en dynamisk udvikling af systemet, hvor vi løbende har tilpasset os de komplikationer vi stødte på under forløbet. Vi har derfor aldrig lagt os fast på endegyldige løsninger, før flere løsningsforslag har været diskuteret og afprøvet. Den iterative proces har hjulpet til at reducere kompleksiteten, ved et ellers meget abstrakt system som vores.

Ud fra de artefakter iterationerne tilbyder, har vi udvalgt dem, som ville give vores proces værdi. De har givet en større forståelse, og en mere effektiv implementering af koden. Use-cases og SD'er har sat ord og billeder på vores tanker, og anskueliggjort fordelingen af ansvaret klasserne imellem.

Fra begyndelsen af processen har vi haft fokus på modellen, for at leve op til de elementære krav. Det lykkedes for os, ved brug af Model-View-Controller princippet, at adskille domæne-laget fra GUI'en. Dette betød i praksis, at modellen kunne fungere selvstændigt.

Ved at inddrage GRASP-principperne i vores udvikling, har vi sikret vores system en god ansvarsfordeling. Vores klasser er højt samhørige og lavt koblede, hvilket har skabt en god struktur og overskuelig kode. Yderligere tilføjer det også vores system en stor fleksibilitet, der muliggør hurtige ændringer, samt nemme implementeringer. Dette har vi blandt andet opnået ved vores controller-struktur, som varetager den nødvendige kommunikation til og i modellen.

Vi har i vores proces, gjort brug af forskellige programmeringsteknikker. PairProgramming har sikret os mod unødvendige fejl, samt en fælles forståelse af koden. Test-driven development har yderligere medvirket til en bedre og mere korrekt implementering, hvilket har medført en mere stabil udvikling.

Ved afslutningen på denne proces, har vi fået bedre forståelse for fordelene ved en iterativ udvikling, samt brugen af forskellige artefakter til illustration af problemstillinger. Vi har også fået større indblik i hvordan et projekt struktureres og måden, hvorpå en konkret opgaveformulering kan gå fra vision til produkt.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Litteraturliste

- Barker, Bret og Bracken, David og Vanhelsuwé, Laurance: *Developing Games in Java*. USA: New Riders Publishing, 2003
- Brookshear, J. Glenn: *Computer Science – an overview*. England: Dartmouth Publishing Inc., 2005
- Christensen, Henrik Bærbak: *Reliable and Flexible Software Explained: Architecture, Patterns and Frameworks*. Danmark: Institut for Informations- og Medievidenskab, 2006
- Larman, Craig: *Applying UML and Patterns – An Introduction to Obejct-Oriented Analysis and Design and Iterative Development.* USA: Prentice Hall, 2006
- Lewis, John og Loftus,William: *Java Software Solutions – foundations of program design.* USA: Prentice Hall, 2005
- Palmer, Daniel W. og Steinberg, Daniel H.: *EXTREME SOFTWARE ENGINEERING – A Hands-On Approach*. USA: Prentice Hall, 2004

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Bilag

Bilag 1

Use Case: Enemy Action

Scope: Rock Star spil

Level: Sub Function

Primary Actor: Dog

Stakeholders and Interets:
Dog: Kender sin koordinater som den har nedarvet fra koordinat klassen.
GUI: Skal kunne Observere en Observable dogklasse for at følge med i en Dog actions og opdatere for UI for i hvert interval.
Hero: Har et forhold til Dog gennem en kontroller af en art. Da en Dog kan dræbe en helt enten ved at være i samme koordinat som helten eller at heltens skud rammer Dog.
Items: Heltens skud er nedarvet fra Items.
DistanceTo: skal overvåge alle objekters koordninater i alle intervaller.

Preconditions:
Main opretter en GameController og en GUI
GameController opretter en LevelController
LevelController opretter en Level

Succes/Main Succes:
1. LevelController opretter en Dog og placerer den i Level ud fra X/Y koordinater
2. Dog vil bevæge sig imellem 2 definerede X koordinater eller/og et specifikt mønster.
3. DistanceTo vil for hvert interval checke Dogs distance til Hero.
4. DistanceTo observerer ikke nogen kollision mellem Hero og Enemy
5. Dog vil for hvert interval sende en notify op til GUI gennem Observable funktionaliteten om dens nuværende position.
6. GUI vil Observere Dogs nye position og opdatere GUI.

Extension
1. Dog rammer Hero
2. DistanceTo observerer dette.
3. Hero får besked fra DistanceTo at den er ramt.
4. GUI vil Observere DistanceTo har meddelt en kollision og opdatere GUI.

1. Hero rammer enemy med et skud.
2. DistanceTo observere dette.
3. Dog får besked fra DistanceTo at den er ramt
4. Dog reducerer dens HP.
5. GUI vil Observere DistanceTo har meddelt en kollision og opdatere GUI.
6. Dette gentages indtil Dog ikke har flere HP.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Bilag 2

Use Case Enemy AI

Scope: Enemy movement

Level: Subfunction

Primary Actor: Hiphopper

Stakeholders and interest:

- LevelController
- CollisionsController
- Level
- Hiphopper,

Preconditions:

- Level har fået tildelt en hiphopper
- Levelcontroller har sendt et start signal til hiphopperen

Succes Guarantee:

- Hiphopper bevæger sig korrekt, dvs. uden at falde ned

Mains Succes scenario:

- Level får tilldelt en hiphopper af levelcontroller.
- Levecontrolleren har givet start signal.
- Heri bevæger han sig mens han skyder med et bestemt interval.
- CollisionController giver besked til hiphopper når han rammer enden af en structure
- Hiphopperen går den anden retning til han rammer den anden ende af structuren
- Når levelcontrolleren får winningCondition besked skal de hiphoppere der ikke er døde fjernes.

Extension:

- Hiphopper vender sig ikke når han rammer en structure – collisioncontrolleren giver ikke besked når han rammer.
- Hiphopperen bevæger sig ikke, eller vender sig om hele tiden – collisioncontrolleren giver for mange beskeder.
- En random funktion der tilladere en hiphopper at hoppe udover kanten og forsætte til han rammer helten eller går udenfor skærmen. Skal poppes fra listen.

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Bilag 3

Usecase Boss AI

Scope: Boss

Level: Subfunction

Primary Actor: Bossen

Stakeholder and interests

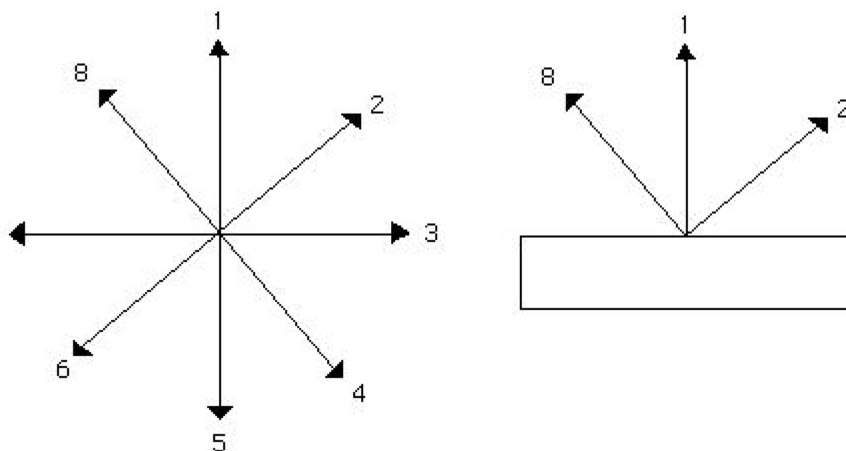
- Levelcontrolleren opretter bossen og tildeler ham til en bestem level.
- BossCollisionControlleren skal kende bossen, og beregne hans kollisioner med structures.
- EnemyMovementController kører AI for bossen (overskriver den normale AI) AI fortæller hvornår han skal skyde.
- ShotController
- ShotMovementController

Preconditions: Bossen er oprettet og placeret i en level.

- Succes Guarantee: Bossen skal flyve random rundt, og skifte retning når han rammer en struktur samt skyde i begge retninger. Måden han skyder og flyver på styres af hans egen AI-metode, samt EnemyMovementController og BossCollisionController.

Main Succes Scenario:

1. Bossen oprettes af levelcontroller og bliver placeret level 5
2. Han startes med at flyve en tilfældig retning ud af 8 mulige
3. Han skyder i et tilfældigt interval
4. Hvis hero kommer ind i bossens synsfelt (horisontalt) skyder bossen ekstra hurtigt.
5. Når bossen rammer en struktur vil enemycollisioncontroller informere ham om at han skal skifte retning. Han tvinges til at vælge en af de 3 retninger der peger væk fra strukturen. (se illustration)
6. Bossen dør og bliver fjernet fra level når han er blevet ramt af 50 skud fra hero.



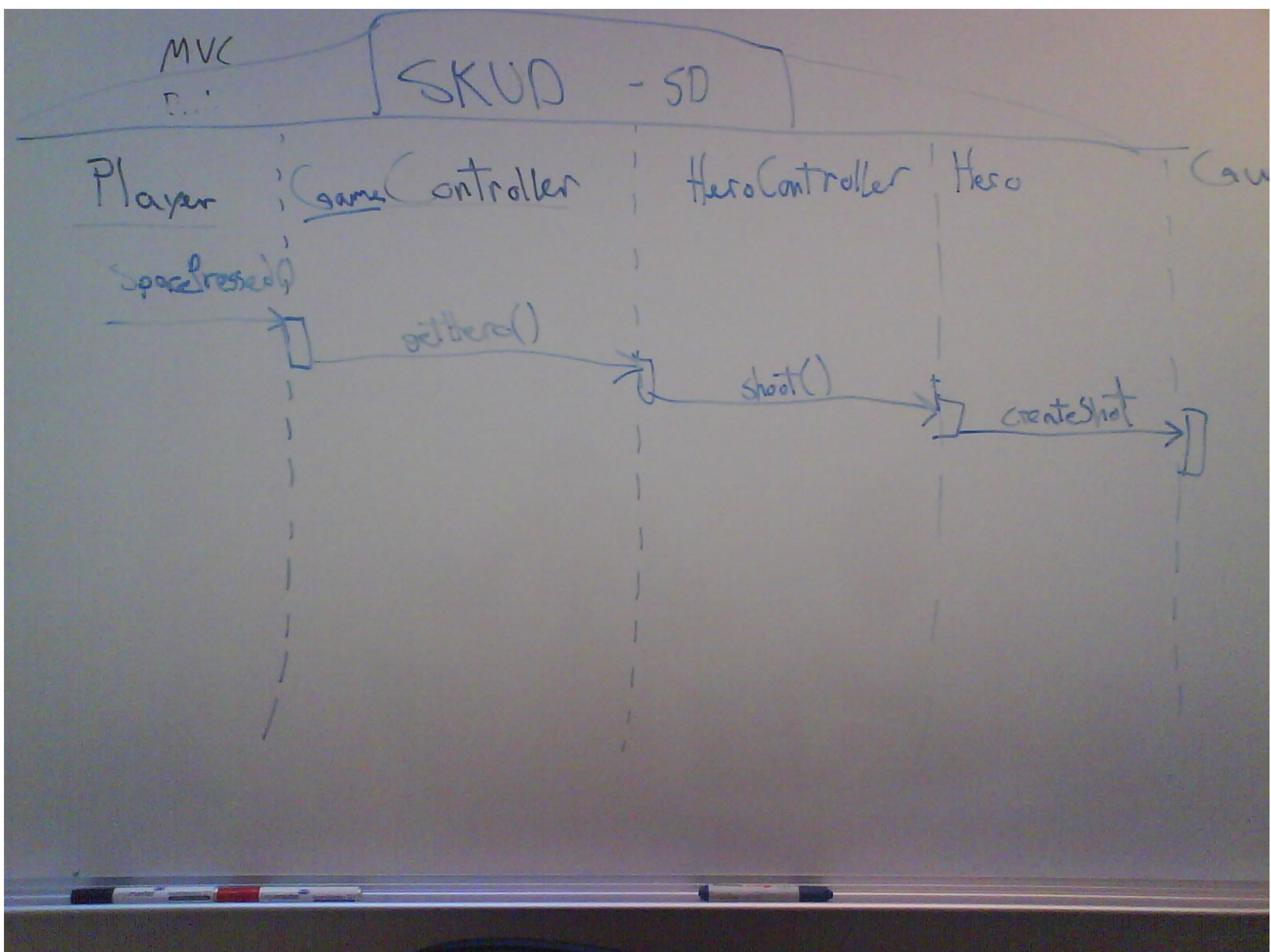
Extensions:

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

1. Structures kan være placeret forkert, el. de retninger bossen råder over kan medføre at bossen fanges i et område af banen hvor han ikke kan komme væk.
2. Der kan opstå blinde vinkler hvor Hero ikke kan rammes af bossen.

Bilag 4

SD for skud:



created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image

Bilag 5

Glossary

Koordinatsæt: Er superklasse for Character, Items og Elementer. Klassen skal indeholde attributterne x1, x2, y1, y2, så objekterne kan flyttes rundt i et koordinatsystem.

Character: Er superklasse for Hero og Enemy. Har navn og billeder, health osv.

Hero: Han er hovedperson i spillet og ham der styres af brugeren. Det eneste brugeren kan styre er Hero. Hero kan skyde, hvis han har en forstærker. Hero kan bevæges med piltasterne, pilop er jump. Hero mister health når han rammer end enemy og dens skud. Hero har en score af point og en healthbar der styrkes af powerups.

Enemy: Kan bevæge sig rundt i level, de kan dræbe hero på forskellige måder. Der findes tre slags enemies. Dog, hiphopper og manager(Boss).

Dog: Kan ikke skyde. Dog dør af et skud.

Hiphopper: De kan skyde med noget, der kan dræbe hero. De kan bevæge sig hen mod hero, når de for øje på ham. De er lidt klogere end dog. De har mere health end Dog. Hiphopper dør af to-tre skud.

Manager(Boss): Kan skyde mange skud og bevæge sig hurtigt. Han kan flyve. Han har en bestemt bane han flyver i. Den er ikke vilkårlig, men det virker sådan for brugeren. Han har mere health end alle andre fordi han er boss. Manager dør af mindst 20 skud.

Items: Powerup(Smøger og whisky), point(Dollars), Skud, forstærker, Gun(Guitar).

Powerup: smøgpakker og whiskyflasker, der hænger i luften. De giver både point og health. De forsvinder når de rammes af hero,

Point: smøger, whisky og dollars. Samme som powerup.

Skud: har kun et y-koordinat. De forsvinder når de rammer hero eller enemy. De medfører mindre health. De har en fed og funky lyd.

Forstærker: Når hero har samlet en forstærker op kan han skyde med sin guitar.

Gun: er det objekt alle skydegale bruger til at skyde med.

Level: Level er bygget op af et koordinatsystem.

har et start felt og et slutfelt. De har nogle point og powerups, samt nogle onde fjender med grimme tænder. Level kender koordinater på alle sine elementer og items(måske)

Elementer: Fungere som vægge og platforme ingen kan gå i gennem. De har flotte farver og lugter rigtig. De udgøre levels rammer.

created using BCL easyPDF Printer Driver
Click here to purchase a license to remove this image

Troels Ebbesen
Nis Sørensen
Tobias Langkjær
Simon Kunddal

Bilag 6

Klasse-diagram over den færdige struktur i 3.iteration

created using
BCL easyPDF
Printer Driver
Click here to purchase a license to remove this image