Syddansk Universitet Odense
Det Tekniske fakultet
Objektorienteret Programmering, E17

Efteråret 2017
Afleveringsdato:
18-12-2017

SwagCity - Semesterprojekt

Gruppeprojekt 1. Semester

Projektgruppe 02

Udarbejdet af:
Sigurd Eugen Espersen
Antonio Ivanoe Gonzales

Under vejledning af:
Erik Sørensen

Opgaven indeholder 75 000 anslag
svarende til 31,25 normalsider

I. Abstract

The assignment were given in the fall of 2017, together with the framework, World of Zuul, that will serve as the main catalyst for the game. World of Zuul contains six different classes that involves walking through empty rooms with simple commands. The main idea presented by the group, was to create a world, where you need to achieve as much swag as possible, to battle the evil celebrity, Sidney Lee. To do this, the player needs to gather up, so-called *"swag items"* through interactions with friendly celebrities, however the swag items will not be offered freely, therefore you will be sent to all kinds of danger, to earn their trust and achieve the item.
By strengthening your swag level enough, you will gain permit to set foot into Sidney's lair and battle him in a glorious battle.
During second phase, the original game was expanded with functionalities like gathering coins, broad variety of commands, different kinds of scenarios, a highscore system, separate inventories for coins and swag, save/load functionality and others small enhancements, handful of them are described further in the report. The game design was implemented with a layered architecture and a user interface, through the program *SceneBuilder*. These improvements were added over two phases and the working process have also been defined in the report. Furthermore, a discussion on how the software development has progressed throughout the entire project were added as well.
A conclusion of the entire project, declares its lacking in documentation development, the reason to this can be determined in the number of group members, but also by the competences that is required to develop such a documentation in a structured way. Giving the chance to make the project anew, the focus would be on documentation and implements of a graphical user interface.

II. Forord

Rapporten er udarbejdet af projektgruppe 2, i objektorienteret programmering i efteråret 2017, med rådgivning af Erik Sørensen. Rapporten tager udgangspunkt i semesterprojektet, hvori der blev udleveret source kode fra World of Zuul frameworket. Hensigten med projektet er skabelsen af et unikt spil, der fungerer uden fejl, navnet på det pågældende spil er SwagCity.

Fundamentalt gennemgår rapporten hvilke visioner, tanker og åbenbaring der har været i den løbende arbejdsproces. Desuden indebærer rapporten hvilke objekter, designvalg og metoder der er blevet fravalgt undervejs i projektet. Essentielt vil rapporten informere læseren om spillets konstruktion, design og hvilke ideer der har været. Desuden vil der blive argumenteret, for de valg der blevet truffet med perspektiv på det bedst mulig resultat.

Derudover vil der være en dybdegående redegørelse for overgangen fra fase 1 til fase 2, med de forskellige udfordringer og ændringer gruppen stod for. Til sidst bliver der vurderet og analyseret på, hvilke fejl og mangler spillet indeholder, samt hvilke muligheder gruppen havde for at forbedre spillet, hvis der var mere tid og ressourcer tilgængelig.

Rapporten er tiltænkt tredjeparts læsere, som gerne vil have mulighed for at læse og forstå arbejdsprocessen for SwagCity, uden et bredt kendskab til programmering.

Opgaven er udarbejdet af:

Sigurd Eugen Espeseth & Antonio Ivanoe Gonzales

Indholdsfortegnelse

I. Abstract 2

II. Forord 3

III. Læsevejledning 6

IV. Ordliste 7

1. Indledning 9

2. Hovedtekst 11

2.1 Funktionalitet i Zuul frameworket 11

2.1.2 Command 11

2.1.3 CommandWord 11

2.1.4 CommandWords 11

2.1.5 Game 12

2.1.6 Parser 12

2.1.7 Room 12

2.2 Visionsdokument 13

2.2.1 Overordnet beskrivelse af spildesignet 13

2.2.2 Spillets formål og funktion 13

2.3 Analyse og Design 15

2.3.1 Klasser i fase 1 15

2.3.2 Metoder i fase 1 16

2.3.3 Fase 2 17

2.3.4 Fase to, arkitektur 18

2.3.5 Mangler i Fase 2 19

2.4 Centrale komponenter 20

2.4.1 Kommandoer 20

2.4.2 Events 21

2.4.3 FXML format og overvejelser 22

2.5 Beskrivelse af brugergrænseflade 24

2.6 Manual 25

2.7 Arbejdsprocessen 26

2.7.1 Ændringer i spillet i forhold til begyndelsen 26

2.7.2 Det iterative arbejde 27

2.7.3 Softwareprocesmodeller 28

2.8 Fejl og mangler 29

2.8.1 Bugs i spillet 29

Interaktion med npcs *29*

Timer *29*

2.8.2 Mangler i spillet 30

Highscore *30*

Bevægelse mellem rum *30*

GUI *30*

3 Diskussion 31
3.1 Enums 31
3.2 Arv 32
3.3 Polymorfi 32
3.4 Save/Load 33
3.5 Indkapsling 35
3.6 Vedligeholdelsesevne 36
3.6.1 Lagdelt arkitektur *36*
4 Konklusion 38
5 Perspektivering 39
A. UML diagram: Zuul framework 40
B. UML diagram: Fase 1 41
C. UML diagram: Fase 2 42
D. Rapportkontrolskema 43
E. Rapporttekniske elementer 44

III. Læsevejledning

I denne rapport skal hvert afsnit kunne læses uafhængigt af hinanden. Dog anbefales det, at man læser analyse og design, samt centrale komponenter før påbegyndelsen af diskussionsafsnittet, da det kræver kendskab til spillet og dets opbygning. Derudover skal det belyses, at hver gang centrale komponenter bliver nævnt, så vil der være en kort introduktion til hvad det er. Dette kan medføre adskillige gentagelser, hvis læseren læser rapporten grundigt igennem, dog medvirker det til, at man kan forstå afsnittene uafhængig af hinanden. Dette er blevet valgt, da læseren skal have mulighed for at gå i dybden med et emne, uden nødvendigvis at skulle læse de andre færdige.

De forskellige midler, såsom metoder og klasser i projektet SwagCity, er blevet udformet via dette mønster: `printWelcome()`, derved vil der vides hvornår der refereres til koden. Når der refereres til metoderne i koden, vil deres parametre dog blive undladt, eftersom dette ikke er nødvendigt for at kunne forklare metodernes anvendelse.

Desuden skal det nævnes, at samtlige diagrammer er vedlagt i rapporten i appendiks A til C. Når disse diagrammer bliver nævnt, refereres det i teksten til hvilket diagram. Selvom rapportens afsnit skulle kunne læses uafhængigt af hinanden, vil der være stunder hvor afsnit referer til hinanden, for at undgå alt for meget gentagelse. Dette foregår under et indledende afsnit eller i selve brødteksten, der nævner at det bliver dybdegjort i et senere afsnit. Eksempelvis opleves dette i afsnittet med Analyse og Design i underafsnittet 2.2.1.

Til sidst vil der visse steder i rapporten være figurer i teksten, årsagen til denne fremgangsmåde er hovedsageligt, for at give læseren et billede af hvordan det er blevet lavet, imens man læser igennem metoderne/klasserne. Ligeledes gældes dette for simple linjer af kode, der er blevet anvendt til at referere til en specifik metode.

IV. Ordliste

Ord	*Beskrivelse*
Pakker	Pakker fungere lidt ligesom mapper på en computer, man kan bruge dem til at opdele klasser i forskellige undermapper (pakker). Dette bruges desuden også for at skjule metoder og attributter fra andre klasser som ikke ligger i samme pakke som klassen selv.
Interface	Et interface definere en generel kontrakt, som kan implementeres af andre klasser, hvorved klasserne får at vide fra kontrakten hvad klassen skal kunne gøre, så er det blot selve implementeringen af metoderne der skal skrives i klasserne.
Klasser	En klasse er et samlingssted for metoder og attributter. Man bruger i mange tilfælde også klasser til at skabe et objekt, som bruges i andre klasser.
Abstrakte klasser	En abstrakt klasse minder om et interface, men forskellen er bl.a. at abstrakte klasser godt kan have implementering i dets metoder, desuden kan man kun implementere én abstrakt klasse, modsat interfaces.
3-lags modellen	3-lags modellen er en måde at opdele sin kode på, hvor man placere sin funktionalitet i ”Business” laget, den grafiske brugergrænseflade i ”Presentation” laget og til sidst, håndtering af filer i ”Data” laget.
Præsentationslag	Præsentationslaget håndtere hvad og hvordan programmet bliver vist til skærmen, dette lag er hvad brugeren ser og derfor systems front end.
Forretningslag	Forretningslaget indeholder alt koden som definere systemets back end. Med andre ord, systemets funktionalitet.
Data lag	Data laget sørger for at lagre information til computeren, i tilfældet hvor der skal gemmes og indlæses filer, som spillet skal bruge.
Acquaintance	Acquaintance er en måde at skjule koden fra de forskellige lag i 3-lags modellen. Dette forbedre sikkerheden af koden og øger vedligeholdelsen af koden. Alt kode i forretningslaget bliver defineret i interfaces i acquaintance, og herigennem forbundet til præsentationslaget. Dette betyder at bl.a. udskiftning af kode bliver markant nemmere.
Facader	Facader er noget som ligger i hvert lag, det er en form for samlingspunkt til hvert lag. Forretningsfacaden bruges til at samle alt det kode, som skal sendes videre til præsentationslaget. Ligeså ligger der en præsentationsfacade, som samler koden fra forretningslaget og deler det ud til de andre klasser i laget. Data laget fungerer ligeledes.
Gluecode	Gluecode eller ”limkode” bruges som navnet hentyder, til at lime alt koden sammen, dvs. at man i gluecoden opretter referencer imellem alle lagene og forbinder hele systemet. Gluecoden indeholder desuden main-metoden som er den første metode der bliver kaldt i et hvert system, og

	som starter hele systemet op. I dette projekt er gluecoden lagt i pakken ”Starter”.
Metoder	En metode er en blot en samling for statements som udfører en eller anden defineret handling. Metoden bliver tildelt et navn og derved kan metoden kaldes af andre metoder.
Constructor	En constructor bruges til at skabe objekter fra klasser.
Accessor	Accessor er blot et andet ord for ”getter”, og en getter metode bruges til at hente værdien af en privat instansvariable.
Mutator	Mutator er blot et andet ord for ”setter”, og en setter metode bruges til at sætte værdien af en privat instansvariable.
Indkapsling	Indkapsling bruges til at skjule data i en klasse ved at gøre dem private, og herefter for at tilgå adgang til disse data, bruges getter og setter metoder.
Enum	Et enum er en speciel klasse, som bruges til at definere konstanter, disse er gode at bruge da det sikrer at en variable, som bruger enums kun kan bruge de definerede konstanter.
Arv	Arv bruges til at få klasser til at arve fra andre klasser, hvorved klasserne som arver, kan bruge superklassens (den klasse de arver fra) metoder eller lave helt nye metoder.
Polymorfi	Polymorfi er når den samme metode har forskellige implementeringer, alt efter hvilket objekt den er defineret på.
Immutability	Immutability betyder at et objekt ikke kan ændres efter det er lavet, dette ses brugt oftest ved at instansvariabler er sat til ”final”.
Timer	En Timer klasse bruges til at sætte opgaver i gang i baggrunden, som kan køre parallelt mens systemet kører.
TimerTask	En TimerTask kan sættes i gang af en timer, og bruges til at lave en reel timer som man kender den.

1. Indledning

Kursuset objektorienteret programmerings grundlæggende aspekt var, at give de nye studerende en række færdigheder til selv at konstruere, debugge og designe eksisterende kode i deres pågældende program. I efteråret 2017 blev frameworket ”World of Zuul” derfor udleveret, som er et simpelt tekstbaseret spil, der giver spilleren mulighed for at bevæge sig rundt i de 5 eksisterende rum. Opgaven var hertil, netop at anvende de færdigheder som kursuset skulle give, til at tage udgangspunkt i frameworket, og herigennem selv konstruere et computerspil, på basis af det skelet som blev udleveret. Herunder er World of Zuul arkitekturen blevet udvidet med nye metoder og opfyldning af krav, samt navnet for projektet er ændret til SwagCity.

Fase 1	**Fase 2**
• Flere rum.	• Spillet skal have en lukket dør, som kun kan åbnes efter indsamling af tre eller flere ”swag items”.
• Bevægelse mellem rummene.	• Andre karakterer/NPC’er i rummet, som spilleren skal kunne interagere med, både gode som onde.
• Genstande i rummene - Nogle genstande skal kunne samles op, andre kan ikke.	• Der skal implementeres en load/save metode, så det er muligt at gemme spillet.
• Spilleren har en begrænset kapacitet ift. Hvor mange genstande, der samtidig kan være opsamlet.	• Der skal tilføjes en tidsparameter i systemet.
• Hver ting har en bestemt værdi som de er værd.	• Det skal være muligt at ”tanke” mere tid.
• Når genstandene samles op indsættes de i en liste af genstande.	• Systemet skal have lagdelt arkitektur.
• Det skal være muligt at vinde.	• Grafisk brugergrænseflade.
• Minimum 4 nye kommandoer, udover de kommandoer, som allerede eksisterer i spillet.	• Dokumentation i form af UML diagrammer.

• Andre karakterer (personer, monstre eller lignende).	• Der skal implementeres en highscore tabel, som gemmes i en fil.
• Nogle karakterer skal kunne bevæge sig imellem rummene.	• Der skal vises en score for hvor godt man har klaret sig.
• Spillet skal have et pointsystem.	
• Det skal være muligt at interagere med nogle af karakterne i rummene.	
• Der skal være en player karakter.	

Minimumskravene er opfyldt i første fase, og de ovenstående krav er løbende blevet tilføjet, da det i starten var begrænset, hvor meget kode der kunne tilføjes, på grund af mandefald i gruppen samt manglende viden. Men efterhånden, som de første par uger gik og undervisningen blev tilegnet arbejdsprocessen, omhandlede det ikke længere at programmere spillet, men derimod spille det igennem og se efter mangler, således at det kunne implementeres. Eftersom gruppen dog kun bestod af to medlemmer, var der meget fokus omkring programmering og god kode frem for alt andet til sidst i projektet.

2. Hovedtekst

2.1 Funktionalitet i Zuul frameworket

Arkitekturen for det oprindelige framework kan ses i appendiks A. Følgende er en uddybning af de enkelte klasser og deres vigtigste metoder.

2.1.2 Command

Denne klasse repræsenterer et `command` objekt. En kommando indeholder et navn, og eventuelt en parameter, som i frameworket kaldes *`secondWord`*. Commands er immutable, det vil sige at de ikke kan ændres efter oprettelse.

Desuden indeholder `Command` klassen instansmetoder til at teste om en kommando repræsenterer en ukendt kommando, `isUnknown()`, og til at teste, om en kommando har en parameter, `hasSecondWord()`.

2.1.3 CommandWord

Denne klasse er en `enum`, der ved projektets start beskrev fire kommandoord: `GO`, `QUIT`, `HELP` og `UNKNOWN`. Det fjerde kommandoord `UNKNOWN` er specielt, idet det benyttes til at repræsentere alle kommandoer, der ikke genkendes af spillet. Hvert kommandoord indeholder en instansvariabel `commandString`, som angiver den streng, der skal associeres med kommandoordet.

Constructoren i `CommandWord` er privat, eftersom det kun skal være muligt, at oprette nye instanser inde i selve `CommandWord` klassen, dette er desuden også et krav for enums.

Derudover indeholder klassen en `toString()` metode, som overskrider den nedarvede metode fra `Object` klassen. Denne metode returnerer kommandoordets strengrepræsentation.

2.1.4 CommandWords

`CommandWords` indeholder en liste af alle gyldige kommandoer, som bliver lavet med en for-each løkke. I CommandWords findes metoder til at returnere en kommando ud fra deres strengrepræsentation, vise alle kommandoer i en liste, og slå op om en strengrepræsentation er en gyldig kommando.

2.1.5 Game

`Game` klassen er den klasse, der binder hele frameworket sammen. Den indeholder en instans af `Parser` klassen til indhentning af brugerinput, samt en reference til det nuværende rum, som spillet befinder sig i. `Game` klassens constructor kalder desuden en metode, der opretter og forbinder alle spillets rum og definere spillerens start rum.

Efter at en instans af `Game` klassen er blevet oprettet, skal instansmetoden `play()` kaldes for at starte spillet. Denne metode udskriver en velkomstbesked, hvorefter game loopet startes. For hver iteration i game loopet indhentes den næste kommando fra spilleren ved brug af parseren, hvorefter kommandoen udføres af instansmetoden `processCommand()`.

Metoden `processCommand()` benytter en række kombinerede if-else statements, til at bestemme hvilken kommando spilleren har indtastet, hvorefter metoden forbundet til den kommando kaldes. Fx kan der udskrives en hjælpebesked, spillet kan skifte rum, eller spillet kan afsluttes.

2.1.6 Parser

Denne klasse håndterer spillerens input. Metoden `getCommand()` modtager input fra spilleren, deler den op ved første mellemrum, og returnerer et `Command` objekt ud fra dette.

2.1.7 Room

Denne klasse repræsenterer et rum i spillet. Ved projektets start er et rum defineret ved at have en beskrivelse samt en række udgange til andre rum. Beskrivelsen gemmes som en privat instansvariabel, mens udgangene repræsenteres ved et *HashMap*, hvor nøglen er en streng af formen `north`, `south`, `east` eller `west`, og værdien er det rum, som udgangen fører til.

Klassen indeholder instansmetoder til at ændre rummets beskrivelse og til at tilføje exits. Desuden findes metoden `getDescription()` til at hente rummets beskrivelse, samt `getLongDescription()` til at hente en længere beskrivelse, som også inkludere rummets udgange.

Derudover indeholder Room klassen, en metode til at hente et af de rum, som rummet er forbundet til, ved at angive en streng der beskriver retningen, hvori rummene er forbundet, denne metode bruges til at opdatere spillerens nuværende rum, og derved spillerens position.

2.2 Visionsdokument

Herunder beskrives visionen for semesterprojektet, som stillede til opgave at udarbejde et java-baseret computerspil. De følgende afsnit beskriver hvilke tanker og idéer der ligger til grund for spillets udførelse, dog er tekniske detaljer, såsom arkitektur, den generelle kode osv. udeladt, disse overvejelser er belyst i afsnit 2.3. Det er også vigtigt at notere at visionsdokumentet ikke er beskrivelsen af det færdigudarbejdede spil, men derimod en beskrivelse af spillet, som gruppen havde forestillet sig det skulle være under projektets forløb. Forskellen mellem spillet som det er beskrevet i visionsdokumentet, og det færdiggjorte spil bliver forklaret i afsnit 2.7.

2.2.1 Overordnet beskrivelse af spildesignet

Spillet ”Swagcity” er en kombination af mange idéer, smeltet sammen i én bunke, det eneste krav gruppen havde til computerspillet, var humor. Spillet starter ved Swagcity byskiltet, her er din primære mission, at blive den nye sejeste/swagste person i byen. Pt. er byen styret af Sidney Lee som er bossen i spillet, spillet vindes ved at besejre Sidney Lee i en kamp, men for at overhovedet komme i betragtning til at udfordre Sidney, skal du først opnå et højere ”Swag-niveau”. Dette gøres ved at hjælpe byens borgere med forskellige missioner, hvorved belønnes man med mere swag. Disse borgere er forskellige bekendtheder, hentet fra den virkelige verden, som giver en belønning der kendetegner den kendte person særligt godt, når man har fuldført deres mission. Når spillet spilles, skal man sørge for at holde sig på benene, eftersom der er en timer i spillet og når den løber ud, er spillet tabt. Igennem spillet er der diverse modstandere, som man helst skal holde sig fra da en forkert handling med dem, også kan resultere i et tabt spil. Ligeledes kan spillet tabes hvis ikke man besejre Sidney Lee, når man har opnået nok swag til at kæmpe mod ham.

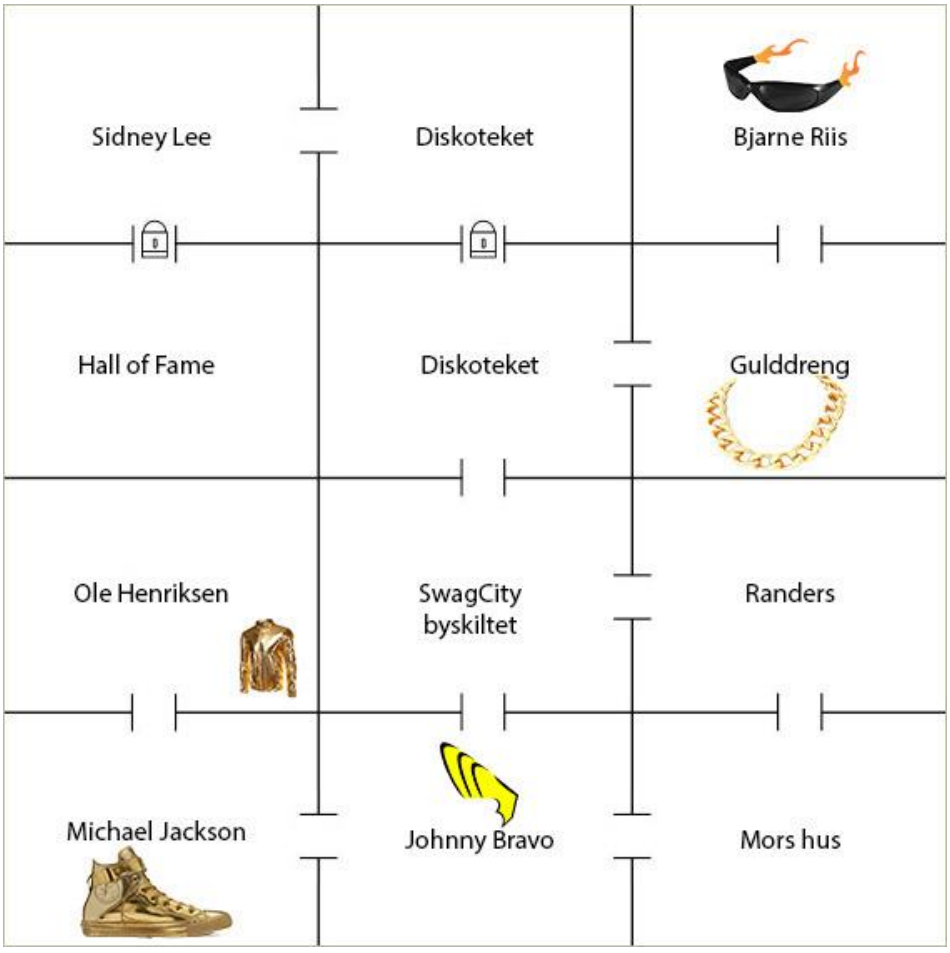
2.2.2 Spillets formål og funktion

Den grundlæggende tanke bag spillet, var at skabe et spil som kunne give et godt grin, samtidig med at man pga. timeren skulle skynde sig lidt. Sværhedsgraden af spillet og spillets genspilbarhed skulle ikke være særlig høj, men derimod fokusere på at give en god førstegangsoplevelse. Formålet med spillet er, at man som spiller skal komme ind på byens diskotek, dog er døren til diskoteket spæret af en dørmand, som kun lukker de mest seje personer ind på diskoteket. Derfor er din første mission som spiller, at skaffe dig nok swag til at få adgang til diskoteket. Hertil kan man finde og interagere med bekendtheder såsom Johnny Bravo, Ole Henriksen, Michael Jackson,

Bjarne Riis og Gulddrengen. Hver af disse bekendtheder giver en mission, som skal klares for at få en swag ting fra dem som belønning. Ved opnåelse af mindst 3 yderlige swag ting, er adgangen til diskoteket åben, og man kan kæmpe mod Sidney Lee for at vinde spillet.
Spilleren bevæger sig rundt i spillet som er opdelt af forskellige rum, hvert indeholdende en bekendthed eller en modstander, sidstnævnte kommer i form af spillerens mor, og Randers typer, som begge prøver vha. trick spørgsmål, at snyde dig og derved tabes spillet.
Spilleren er derudover udstyret med en `inventory` og en `wallet`, som henholdsvis bruges til at holde de swag ting man får fra missioner, og mønter som man kan samle op rundt omkring i spillet. Desuden er det muligt for spilleren at gemme sit nuværende spille og indlæse det på et andet tidspunkt, hvor antallet af missioner fuldført er gemt, spillerens gemte position, samt antal mønter man har samlet op og den resterende mængde tid tilbage på spillet, bliver indlæst.

Til højre kan et kort over spillet ses, som samtidig viser hvilke ting man kan få fra hver bekendthed, Ole Henriksen (fabolous tøj), Michael Jackson (Guldsko), Johnny Bravo (Bravo håret), Gulddreng (guldkæden), Bjarne Riis (hurtigbrillerne).

Man starter ved SwagCity byskiltet, herfra kan man bevæge sig i præcis den retning man har lyst til. Målet er at besejre Sidney Lee og derved opnå indgang til Hall of Fame og vinde spillet.



Figur 1 - Kort over spillet

2.3 Analyse og Design

Afsnittet vil gennemgå de analyser og designmæssige overvejelser ved projektet. Strukturen er opført i to dele, hvori første fase og anden fase bliver beskrevet med dets ideer og overvejelser.

2.3.1 Klasser i fase 1

I denne fase skulle der udvælges forskellige navneord fra visionsdokumentet, til skabelsen af de forskellige klasser. Navneordene er som følgende: `Main Game`, `Game`, `Player`, `Room`, `Swag(item)`, `CommandWords`, `CommandWord`, `Parser`, `Moving`, `Look`, `Coin`, og `Pengepung` samt diverse `NPC klasser`, `Lock` og `Inventory`.
Det blev dog vedtaget, at `Look`, `Moving`, `Lock` og `Pengepung` klassen ikke skulle kategoriseres som klasser. `Look` funktionen gav et overblik over ting i rummet, men det gav ikke mening at have en funktion som informer om dette, da der allerede eksisterede en indledende beskrivelse af hvilke ting der var i rummet, som blev printet ved hvert skift af et rum. Desuden ville en funktion som `Look` slet ikke have noget brug, efter implementeringen af en grafisk brugergrænseflade. `Moving` funktionen er ligeledes blevet kasseret, eftersom en metode der tillod NPC'er at bevæge sig imellem rum, skulle implementeres senere i projektet, derfor gav denne funktion heller ingen mening at beholde.
Derudover blev `Inventory` og `Pengepung` (senere omdøbt til `Wallet`) implementeret, som en variabel af `Player`, i stedet for at lade det forblive i `Game` klassen. Dette gav mere mening, da `Player` klassen selv burde indeholde sin pung og inventar, eftersom dette ville være mere naturtro.
`Lock` funktion var tilegnet diskoteksdøren, idéen var at indsamle tre swagting for at kunne komme ind og bekæmpe Sidney Lee, derfor var det ikke nødvendigt at `Lock` funktionen havde sin egen klasse, så den blev derfor en metode beliggende i `Room` klassen.
Herefter gik overvejelsen på implementeringen af fleksible spilscenarier, i denne anledning blev NPC klasserne skabt, disse NPC klasser skulle indeholde de forskellige objekter, som spilleren kunne få via en interaktion med NPC'erne. I forbindelse med disse forskellige spilscenarier, skulle det være muligt at angive handlinger, som resultatet af spillerens valg. Derfor blev der implementeret en abstrakt klasse, `NPC`, og hertil lavet NPC-klasser for hver NPC, spillet indeholdte. Disse NPC-klasser kunne så igennem arv, arve fra overklassen `NPC`, eftersom alle

NPC'er havde hver deres separate mission som kunne gennemføres. Dette var særligt smart, da den abstrakte klasses opgave var at definere en skabelon, som alle NPC'er kunne gøre brug af når spilleren gav en kommando. Bl.a. indeholdte den abstrakte klasse `NPC`, en boolsk metode `setQuest()`, som kunne bruges til at sætte en missions tilstand til fuldført eller ej. Eksempelvis for at opnå et swag item fra Michael Jackson, skal du svare rigtigt på hans spørgsmål, hvis ikke du svarer korrekt, bliver hans `setQuest()` metode ikke sat til true, og dermed opnår man ikke den pågældende swagting.

2.3.2 Metoder i fase 1

Gennem visionsdokumentet er der blevet indsamlet en række metoder, som spillet indeholder. Mængden af metoder gør det urealistisk at skrive dem alle op, derfor er der blevet fokuseret på de mest centrale metoder i følgende afsnit.

Ethvert program skal have en main metode for at igangsætte det. Derudover, da spillet er tekstbaseret er det nødvendig med en `getCommand()`, som registrer spillerens input, således at det omformuleres til en handling, såsom at bevæge sig mod enten nord, syd, øst eller vest. Metoden vil dermed finde klassen CommandWords og eksekvere `getCommand()` metoden. Med hensyn til de forskellige spilobjekter og NPC'er, skal der ligeledes implementere metoder såsom `getGreeting()`, `getName()` i `NPC` klassen, således at spilleren kan få information omkring verden i spillet. Samme metoder skal implementeres i `Room`, dog er der setter og `remove()` metoder, grundet de andre indsamlings genstande ved navn `Coin` og `Swag`. `Remove()` metoden anvendes til at fjerne penge fra rummet, efter de er samlet op. Til sidst er der blevet undersøgt efter mangler i spillet, herved er der blevet konstateret, at der godt kunne implementeres en form for AI (Artificial Intelligence) der kunne køre spillet i forskellige scenarier, men eftersom det er udenfor gruppens kapacitet grundet gruppeantallet, er dette undladt. Klasserne samt metoderne som er beskrevet ovenfor, vil være udspecificeret i mere detaljeret form i appendiks B.

2.3.3 Fase 2

I anledningen af fase to blev kravene udvidet markant. Dette segment vil gennemarbejde kravene og beskrive tankerne ved de forskellige implementeringer.

Interaktion mellem spilleren og NPC'er er det mest fundamentale i spillet, tidligere var interaktionen mellem NPC'erne defineret i adskillige NPC klasser, med deres respektive navne, hver NPC's interaktion blev kaldt via `Game` klassen. Dog blev der senere konstrueret en abstrakt NPC klasse, der indeholdte forskellige metoder, som kunne implementeres i underklasserne for hver specifik NPC, ved en interaktion såsom `interact()`, `setQuest()` og `isQuest()`. F.eks. hvis man befinder sig hos Johnny Bravo og interagere med ham, vil metoden `interact()` blive kaldt, dernæst vil NPC'en afhængigt om man svarede korrekt på den pågældende interaktion, kalde metoden `setQuest()` og til sidst vil `isQuest()` tjekke om du har gennemført missionen.

Interaktionen mellem NPC og spilleren, blev yderst vanskelig i fase to, eftersom designet af præsentationslaget ikke godtog alle de if-else statements der blev foretaget. Derfor blev konstruktionen af interaktionen mellem NPC og spilleren lavet om til en switch, som ved hjælp af en "`interactionState`" kunne få det til at fungere. Endnu en metode der blev implementeret var `QuestQuit()`, dets formål var at afslutte spillet, hvis spilleren havde foretaget en handling, som resulteret i et tabt spil. Eksempelvis under Johnny Bravo missionen, skal du skaffe nummeret fra pigen, Beatrice, der befinder sig i Randers. Når dette er blevet gjort, skal nummeret afleveres til Johnny Bravo, som sørger for at belønne dig med en swagting, dermed bliver missionen fuldført. Desuden bliver det sikret, at man efter en missions fuldførelse, ikke kan gennemføre den flere gange.

Lukkede døre blev implementeret i fase et, her blev en døres tilstand tjekket via en boolsk værdi, om døren var låst eller ej. Dog er der ikke en fysisk dør i spillet, men derimod en illusion af en dør, af denne årsag blev der oprettet en boolsk metode, `isLocked()`, der brugte den retning som spilleren prøvede at gå i som parameter, til at tjekke om en dør var låst eller ej. Efter spilleren havde indsamlet tre eller flere swagitems vil spillet kalde en `setLock()`, som vil sætte dørens låste positions til *false* og herved vil man kunne gå igennem døren.

Pointsystemet blev gennemført via en ny klasse, `HighScoreManager` dette fungerede fint i forhold til at gemme en enkelt score, efter hvert vundet spil, men det var ikke muligt at

implementere en sorteret liste til de ti bedste forsøg, da mandefald i gruppen, lagde resten af gruppen under et stort tidspres.
Til gengæld blev der lavet en tidsparameter der har til funktion, at spilleren skal gennemføre spillet inden for en hvis tidsramme, ellers er spillet tabt. Yderligere kan der tilføjes mere tid, gennem færdiggørelse af diverse missioner. Denne tidsparameter blev lavet som en ny klasse, `gameTimer`, hvor start tiden er defineret til en start værdi 120 sekunder. Klassen har diverse metoder, bl.a. `timerStart()` der starter timeren, og herefter tæller et sekund ned pr. sekund, denne metode bliver kaldt, når selve spillet startes. Herfra kan spilleren få tildelt mere tid gennem metoden `addTime()`, som bliver kaldt når en mission bliver fuldført, på den måde bliver tiden forøget i takt med at man klarer sig godt i spillet. Derudover er der taget højde for, at spilleren skal læse instruktionerne, eller andet via hovedmenuen, derfor bliver timeren ikke startet før efter introduktionen, og man som spiller har trykket på *"videre"* knappen. I forbindelse med dette blev metoden `timerStop()` indset at være nødvendig, dets funktion aktiveres når spilleren vender tilbage til hovedmenuen fra spillet. Her bliver `timerStop()` kaldt, hvilket stopper timeren og dermed afsluttes spillet ikke, hvis man vil vende tilbage til hovedmenuen. Der er dog ikke blevet implementeret en virtuel tidsparameter, grundet mandefald i gruppen, men spillet tæller stadig selv ned via metoden `run()` som kører i baggrunden. `Run()` metoden ligger i `gameTimer`, som registrerer hvis spilleren ikke har mere tid tilbage og lukker spillet ned.

2.3.4 Fase 2, arkitektur

Ud fra de nye krav i fase to skulle spillets arkitektur ændres til en lagdelt arkitektur (se appendiks C). Arkitekturen har tre forskellige lag: `Presentation`, `Forretning` og `Data`, derudover har projektet et `acquaintance` lag og et `gluecode` lag.
Af denne årsag blev `highscoreManageren` der indeholdte logikken til udskrivningen af score filerne, flyttet til datalaget.
I forretningslaget ligger `businessFacaden` som har til opgave, at kommunikere med datalaget og forretningslaget således at der er så lav kobling som muligt. Tidligere var funktionerne, `inventory`, `wallet`, `time`, `score` og `currentRoom` blevet spredt i projektets kode. Senere valgtes det at implementere disse funktioner i `Player` klassen, da det ikke gav mening at `Player` klassen ikke selv indeholdte sin inventar, pung, tid, score og position. Desuden gjorde

dette spillet langt nemmere at gemme, fordi spilleren selv indeholdte de værdier som skulle gemmes, og man kunne derfor nøjes med kun at gemme spilleren som det eneste objekt.
Derudover blev spillet udvidet med en brugerinterface, som indeholdte præsentationslaget. Dette lag er koblet til businesslaget gennem adskillige nye interfaces ved navn `IBusiness`, `ICoin`, `IScore` osv. der ligeledes står for at kommunikere mellem præsentationslaget og forretningslaget. Hvorpå kommunikationen foregår gennem strenge, void og boolske metoder, således at der forekommer en så lav kobling som muligt.
Det skal pointeres at lagdeling altid bør være nedadrettet, derfor skal præsentationslaget slet ikke indeholde nogen form for forretningslogik, men derimod igennem en acquaintance pakke opnå en kobling til forretningslaget. Når spillet startes, bliver `main` metoden kaldt, denne metode indeholder alt gluecoden, og herigennem realiseres `businessFacaden` som interfacet `IBusiness`, dette skaber koblingen mellem præsentationslaget og forretningslaget. Herved kan man kalde metoder fra `IBusiness` interfacet, og få implementeringerne fra `businessFacaden`. Ligeledes kan forretningslaget heller ikke kommunikere med præsentationslaget, andet end igennem acquaintance pakken.
Arkitekturen vil blive dybdegjort i et senere afsnit 2.5 og 3.6.1. Derudover vil forretnings arkitekturen kunne ses i appendiks C.

2.3.5 Mangler i Fase 2

Planen var at etablerer bevægelse mellem rum, således at nogle af NPC'erne selv ville interagere med spilleren, ligeså snart spilleren trådte ind i rummet. Dette blev foretaget i første fase af projektet, hvori sidste bossen, Sidney Lee ville udspørge spilleren om, hvor mange af kendisserne i SwagCity, han kendte personligt. Dette blev udført via et if-else statement, men under anden fase og implementeringen af en grafisk brugergrænseflade, var det ikke længere muligt. Grundet mandefald i gruppen blev dette nedprioriteret i forhold til andre ting, og denne funktion blev derfor ikke realiseret igen i anden fase. Planen var også at få nogle NPC'er til selv at bevæge sig imellem rummene, og af samme grund som før, blev denne funktion ligeledes ikke implementeret.
Idéen om animationer, såsom skabelsen af *"sprite sheets"* blev afskaffet i fase to, samt at skabe et platformsspil i retning af Super Mario. Dog blev dette også droppet grundet mandefald i gruppen.

2.4 Centrale komponenter

Næste afsnit omhandler at beskrive og diskutere de vigtigste komponenter i spillet. Derudover er et nøje antal komponenter valgt, hvori resten vil blive snakket om til den mundtlige eksamen, hvis der skulle være nogen form for specifikke spørgsmål.

2.4.1 Kommandoer

I starten af projektet blev de mulige kommandoer fra Zuul frameworket, udvidet med flere enums end hvad der allerede befandt sig i CommandWord klassen. Enums er den faktor som udgør samtlige kommandoer i spillet, heriblandt er nogle af de enums `GO`, `QUIT`, `HELP`, `INVENTORY`, `GET`, `INTERACT`, `WALLET`, `SAVE` og `LOAD`, samt et enum af typen `UNKNOWN`. Disse enums tilknyttes til et `Command` objekt i spillet, og med hjælp fra dette enum vil Command klassen, sørge for at samtlige kommandoer bliver udført i spillet. Derudover er disse instanser af objekter nogle containers, der indeholder forskellige informationer, som er nødvendig for at iværksætte kommandoen. Objekterne indeholder nødvendigvis ikke kun enums, men kan også have et `secondWord`, der fx kunne være et `Map`, hvori nøglerne kan være navne på parametre såsom, `direction`, `coin`, `north` og `NPC`.

BusinessFacaden som har implementeret IBusiness interfacet, indeholder metoder der bl.a. er i stand til at udføre handlinger, der er forbundet til kommandoerne. Metoden `goToDirection()` med dets implementering, kan ses herunder:

```
@Override
public String goToDirection(String direction) {
    Command c = new Command(CommandWord.GO, direction);
    return game.goRoom(c);
}
```

Herved kan der konstateres, at via `goToDirection()` metoden, bliver der konstrueret et nyt `Command` objekt, som bliver initialiseret med et enum af typen `GO`. Således vil spillet få at vide, at det er `GO` funktionaliteten som skal bruges. Dernæst sender objektet en parameter af typen `direction` som er en streng, der i projektet kun er defineret som enten `north`, `south`, `east` eller `west`. Kommandoen bliver videreført til game klassen der initialisere kommandoen. Metoden bliver anvendt gennem en `switch` i `Game` klassen, der tager et commandWord som værdi. Som set ovenfor bliver metoden `goToDirection()` kaldt, dertil indsættes `GO` enum i switchen, herefter bliver strengen `direction` analyseret, hvilket bliver akkumuleret til en

retning systemet skal tage. Når switchen har fundet ud af hvilken case den skal arbejde med, bliver de pågældende metoder kaldt. Denne struktur gør det simplere at arbejde med `Game` klassen, da man så ikke er nødsaget til at skrive utallige mængder koder, men derimod begrænse koden til det mest nødvendige og drage nytte af det.
Eksempelvis hvis spilleren gerne vil mod nord, bliver der igennem andre metoder, sikret at spillerens nuværende placering bliver opdateret. Lagdeling kommer til udtryk eftersom IBusiness, ikke behøves at kende noget til den logik der bliver foretaget i forretningslaget, udover den kode der er beliggende i `businessFacaden`.

2.4.2 Events

I dette projektet har et af formålene været, at skabe fleksible spilscenarier. Derudover skulle rummene gøres mere unikke, samt humoristiske for spilleren, af denne årsag er missionerne markant anderledes i forhold til hinanden. Under en interaktion er der lignende mønstre for at modtage missionen, men når missionen er igangsat er de væsentligt anderledes. Eksempelvis ved Bjarne Riis skal man fremskaffe hans Epo, her må man ikke interagere med andre NPC'er efter man har modtaget EPO'en end Bjarne Riis, for hvis man gør, bliver man busted og skal starte forfra. Siden at rummene skulle være anderledes, og de forskellige handlinger skulle medføre en specifik metode blev kaldt, derfor blev eventsystemet designet. Måden at skabe events er ens med kommandosystemet, der bliver implementeret parametre og disse eksekveres, derfor dykkes der ikke længere i dybden med dette. I spillet er der tre måder hvorpå events kan aktiveres på: Den ene indebærer at man trykker på en knap, fx *"tag penge"*, via denne knap bliver en ActionEvent kaldt, der medfører at man samler pengene der ligger i rummet op. Dette er bestående af adskillige knapper med hver deres ActionEvent såsom: Hjælp, Hovedmenu, NORD, SYD, ØST og VEST, Pung, Swag ting, Gem og Indlæs. Disse events bliver anvendt via Controllerne, samt diverse FXML filer der indeholder scenes til hver ActionEvents funktion. Eksempelvis for at samle pengene op, anvendes metoden `PlayerGetAction()` derefter via implementeringen af SceneBuilder bliver metoderne sat til knapper, så hvis en knap trykkes på, akkumuleres det pågældende ActionEvent. Den anden type event er ligeledes programmering med SceneBuilder, men i dette tilfælde er det at skabe events, der kan skifte mellem de forskellige scener i spillet, samt at returnere tekst til at informere spilleren om hvad der foregår, fx i instruktions scenen.

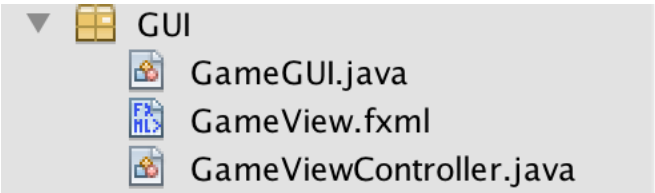
Event anvendes desuden til at afslutte spillet, og vinde spillet hvor ens highscore bliver vist. Det tredje event er et *"MouseEvent"* der blev brugt til instruktions filen, denne type event blev implementeret i de forskellige ikoner i scenen. MouseEventet tjekker om der har været et tryk på musen, hvorefter indholdet på instruktionsscenen skifter ud fra dette event.
Når et event kaldes, vil det grundet spillets design af den grafiske brugergrænseflade, resultere i at noget tekst skal printes til skærmen, da GUI'en er tekstbaseret. Til dette formål bruger eventsne metoderne `appendText()` og `setText()`, disse metoder skriver deres parameter ud til et `textArea`. Eksempelvis hvis `appendText()` bliver kaldt med den førnævnte `goToDirection()` metode, som parameter, vil det tekst som `goToDirection()` returnere blive printet til skærmen. Dette ses også ved spillets intro tekst, som bliver printet i en `textArea`, hvorefter man kan trykke på en ActionEvent, der starter selve spillet op.

2.4.3 FXML format og overvejelser

Afsnittet vil generelt forklare hvilke designvalg, der er blevet truffet med hensyn til de forskellige spilscenarier. Samtidig vil der blive gennemgået hvilke filformater der er blevet valgt, samt de overvejelser der skulle foretages med designet af fillæseren.

Lige fra begyndelsen var målet at lave et sammenhængene spil, som skulle kunne fungere så flydende som muligt. Heraf kan det ses fra præsentationslaget hvor der er en FXML fil, samt en controller til hver FXML fil for hver stage (vindue) spillet indeholder. Valget FXML, er hovedsageligt grundet at det er brugbart til at repræsentere komplekse strukturerer, dvs. et led kan være opbygget af andre led, som ligeledes kan være opbygget af flere led osv. Et lignede format der kunne have været anvendt var *"Sprite sheets"*, som kunne have givet spillet animationer, hvilket kunne have hjulpet på spiloplevelsen. Dog blev disse animationer som nævnt undværet, da der ikke var tid og ressourcer nok i gruppen til at foretage dette, af denne grund blev der arbejdet udelukkende med FXML. Derudover understøtter Java, FXML, med et eksisterende bibliotek som kunne læse FXML filerne, hvilket gør formatet let at implementere.

Pakkestrukturen er opbygget som vist på billedet herunder:



Figur 2 - Præsentations lagets pakkestruktur

I pakken GUI, ligger Java filen `GameGUI`, denne fil indeholder metoden start, der starter den første *stage* (vindue), som skal kaldes når hele programmet starter. Derefter er der FXML filer for selve scenen og dets controller. Denne struktur med FXML scene og controlleren, gælder ligeledes for de resterende elementer i GUI mappen. Årsagen til at der kun er én GameGUI fil, er fordi man kun har brug for én `start()` metode, der sørger for at starte spillet op. Eftersom spillet er blevet tekstbaseret, er strukturen anderledes end hvad der er blevet forventet, men hovedsageligt følger det mønsteret med en form for *"Pane"*. Billedet herunder viser et eksempel på en FXML fil for en scene, i dette tilfælde er det `AnchorPane`, `labels` og `buttons` der er blevet brugt.

```
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0" style="-fx-background-color:
xmlns:fx="http://javafx.com/fxml/1" fx:controller="GUI.WinViewController">
   <children>
      <TextArea fx:id="textConsole" editable="false" layoutX="173.0" layoutY="100.0" prefHeight="200.0" prefWidth="255.0" wrapText="true" />
      <Label layoutX="166.0" layoutY="23.0" text="Du vandt!" textFill="WHITE">
         <font>
            <Font size="62.0" />
         </font>
      </Label>
      <Button fx:id="scoreShow" layoutX="222.0" layoutY="188.0" mnemonicParsing="false" onAction="#scoreShowAction" text="Tryk her for at se din score" />
      <Button fx:id="mainViewButton" layoutX="260.0" layoutY="323.0" mnemonicParsing="false" onAction="#mainViewButtonAction" text="Hovedmenu" />
   </children>
</AnchorPane>
```

Figur 3 - spillets FXML struktur

Som nævnt består FXML strukturen af et *"Pane"*, dette *"Pane"* samt dets indhold, er hvad der bliver vist til skærmen. Strukturen med et `AnchorPane` er gældende undtaget ved `GameView` hvor der anvendes et `BorderPane` i stedet. Via MouseEvents er alle scenes desuden konstrueret således, at det er muligt at flytte rundt på stagen, som det passer en, for mere information omkring MouseEvents, læs afsnit 2.4.2. FXML strukturen vist herover er for `WinView` scenen, der udover at indeholde et `AnchorPane` også har en label, der informerer spilleren om at de vandt spillet. Samtidig kan spilleren få deres score printet, vha. af knappen med teksten, "tryk her for at

se din score" hvori det vil stå i `TextArea`. Til sidst eksistere knappen "Hovedmenu", som vil føre en tilbage til hovedmenuen.

2.5 Beskrivelse af brugergrænseflade

Følgende afsnit beskriver projektets brugergrænseflade, set fra et teknisk perspektiv. Instruktionsguiden til hvordan brugergrænsefladen bruges, bliver belyst i afsnittet omkring manualen, 2.6.

Brugergrænsefladen som er udviklet til dette projekt, benytter event-baserede handlinger, som styrer interaktionen mellem bruger og system. Det vil sige, at der i baggrunden af JavaFX kører `EventHandlers`, som lytter til hvornår knapperne i GUI'en trykkes på, herfra vil der så blive kaldt på de pågældende metoder, som er koblet op til hver knap.

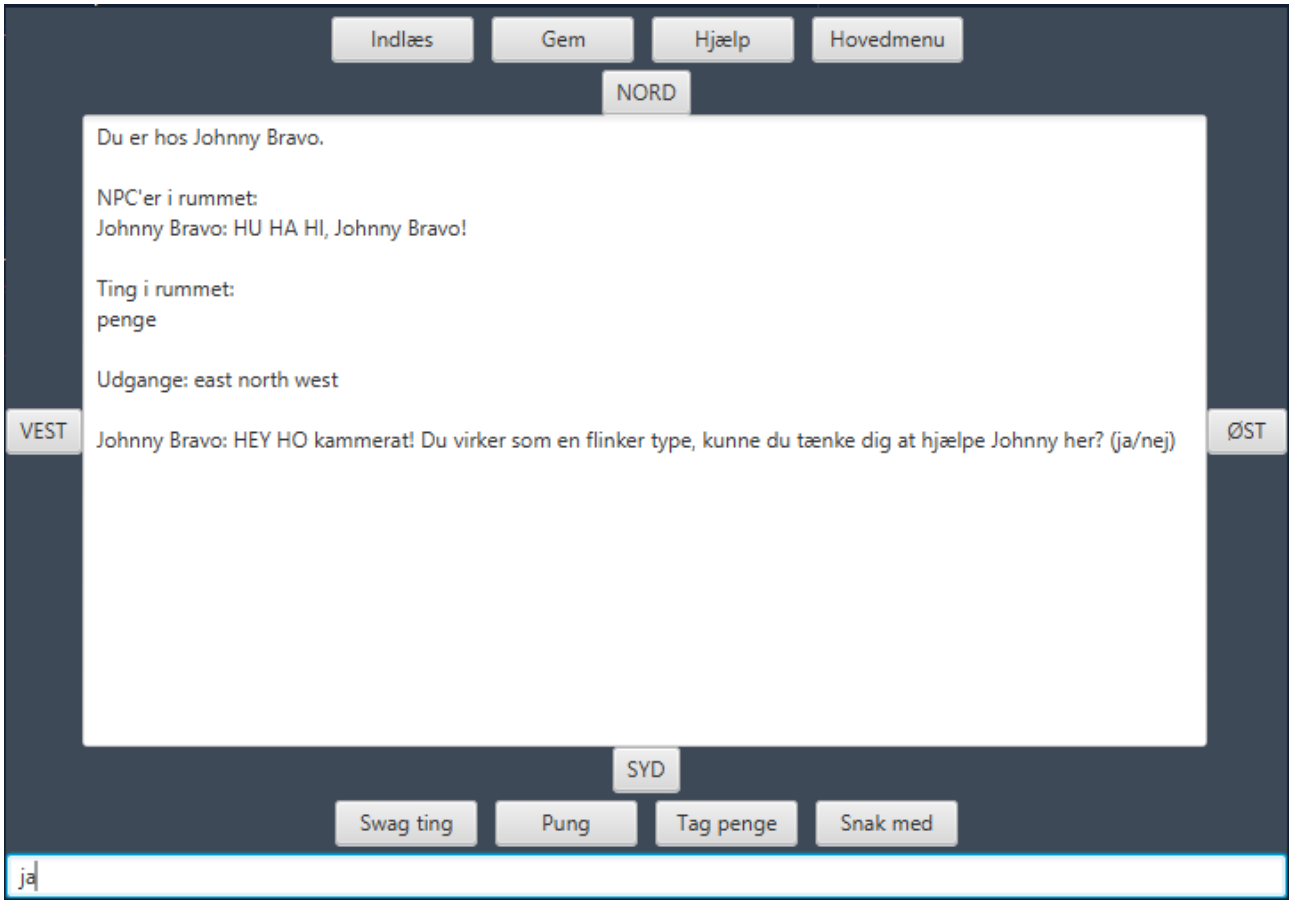
Den grafiske brugergrænseflade, bruger et `TextArea` til at skrive spillet ud på. Dvs. at hver gang der bliver kaldt en metode fra en event handler, bliver der returneret en streng, som bliver printet på `TextAreaet`. Dette design af GUI'en har derfor resulteret i, at alt kode som skulle vises på front end, skulle returnere strenge.

Eftersom der bruges acquaintance, kalder alle metoder fra præsentationslaget til acquaintance pakken, som indeholder alle interfaces. Via gluecoden bliver `BusinessFacaden` realiseret som et interface, `IBusiness`, `BusinessFacaden` indeholder alt implementering af metoderne, som skal bruges i brugergrænsefladen. Herved sikres det at brugergrænsefladen, ikke kender til andet end acquaintance laget, hvilket gør spillet meget mere sikkert i forhold til snyd, men også i forhold til vedligeholdelse ved forbedringer til spillet.

I projektet er separationen mellem hvert lag gjort meget tydeligt, ved at anvende 3-lagsmodellen samt acquaintance. Dette betyder at der er pakken ved navn *"GUI"*, som håndtere alt kode og funktionalitet til den grafiske brugergrænseflade. Det eneste link der er mellem præsentationslaget og forretningslaget er igennem acquaintance. Dernæst er der i projektet en forretningspakke ved navn *"Business"*, denne pakke styrer hele skelettet af spillet, det er her alt den funktionelle kode finder sted. Dette lag har ligeledes kun forbindelse til acquaintance, men også datalaget. Datalaget håndtere filer som skal gemmes eller indlæses fra computeren, laget har derfor ingen forbindelse til brugergrænsefladen, eftersom det er forretningslaget som styrer dette lag, og sikrer at skrivning og indlæsning af filer sker på de rigtige tidspunkter.

2.6 Manual

Når spillet startes bliver spillets hovedmenu indlæst, denne menu indeholder fire knapper, start, instruktioner, highscore og afslut. Afslut knappen afslutter spillet, highscore skulle have vist top 10 highscores af spillet, instruktioner åbner en ny scene som forklarer hvordan spillet spilles. Sidste knap ”Start” starter selve spillet og en ny scene bliver fremvist, dette er introduktions scenen som kort forklarer hvad det egentlige formål med spillet er. På introduktions scenen findes knappen ”videre”, som sender dig videre til selve spil scenen, her startes en timer på 120 sekunder, når timeren rammer 0 har man tabt spillet, dog kan der indsamles mere tid ved at fuldføre missioner.



Figur 4 - Spillets GUI

Herover ses spilscenen, i det store hvide tekst felt, bliver hele spillet skrevet ud i en tekstbaseret version, knapperne bruges til at spille spillet. Afhængigt af hvilke udgange som er tilgængelig, kan man gå en retning ved at trykke på enten, ”SYD”, ”NORD”, ”ØST” eller ”VEST”. Knappen ”snak med”, interagere med den npc som er i rummet. Hertil skal man svare på spørgsmål, som bliver stillet af npc’erne, man svarer ved at indtaste svaret i inputfeltet i bunden af skærmen, og trykker en gang mere på ”snak med” for at sende beskeden til npc’en. ”Tag penge” knappen samler

pengene op fra rummet hvis der er nogen, hvorefter den sletter pengene fra rummet og lægger pengene i din pung. Pungens indhold kan vises ved at trykke på ”pung” knappen, som fortæller hvor mange mønter du har på dig lige nu. ”Swag ting” knappen er spillerens udstyr, ved fuldførelse af fx Johnny Bravo’s mission får man en paryk der ligner hans hår, denne paryk vil blive placeret i ”swag ting”. Swag ting er desuden det som skal bruges, for at opnå adgang til diskoteket. ”indlæs” og ”gem” gemmer og indlæser spillet. Når spillet gemmes, gemmes spillerens nuværende position, antal mønter, antal swag ting og hvor meget tid man havde tilbage af timeren. ”Hjælp” knappen forklarer kort, hvordan man bruger de vigtigste knapper i spillet, i tilfælde af at man havde glemt det, eller ikke læste instruktionerne inden man startede. ”Hovedmenu” knappen bruges blot til at vende tilbage til hovedmenuen.
Spillet kan tabes ved enten at miste alle sine swagting, som kan mistes fra forskellige personer i spillet, som prøver at frarøve dig dine ting. Ellers tabes spillet også hvis timeren løber ud eller man taber over Sidney Lee. Spillet vindes ved at besejre Sidney Lee og komme ind i Hall of Fame.
Når spillet tabes ellers vindes, bliver man sendt henholdsvis til en game-over scene eller også en victory scene. På victory scenen kan man se sin score og hvor meget tid man havde tilbage. Disse værdier skulle have været gemt, i en highscore fil på computeren.

2.7 Arbejdsprocessen

2.7.1 Ændringer i spillet i forhold til begyndelsen

Ændringerne i spillet i forhold til begyndelsen er ikke det store, visionen for spillet er i store træk det samme. Man starter stadig ved Swagcity byskiltet, og skal herfra blive den nye sejeste person, ved at klare missioner for bekendtheder, og derved opnå adgang til diskoteket og besejre Sidney Lee.
I forhold til selve spildesignet, skulle spillet have været lavet med *”sprites sheets”*, som skulle vise den grafiske brugergrænseflade. Men eftersom gruppens medlemmer faldt fra 6 mand til 2, blev det nødsaget, at skære ned på kravene til spillet, derfor blev den grafiske brugergrænseflade lavet om til en tekstkonsol som fungerer vha. knapper. Dette har haft stor påvirkning på koden, eftersom dette har betydet, at meget af koden skulle ændres til at returnere strenge, derved er meget af koden blevet slettet og nyt har taget dets plads. Fx er parseren som læser input helt

fjernet, sammen med CommandWords klassen, da ingen af dem havde nogen funktion på GUI'en. Eftersom den grafiske brugergrænseflade ikke bliver lavet, som oprindeligt tiltænkt, er designet af spillet blevet nedprioriteret. Humoren af spillet som var en stor faktor, har derfor lidt under dette, da meget af det komiske element lå i at have spilleren, bevægende sig rundt i spillet, iført alle spillets genstande fra bekendthederne, som var at finde i spillet, hvilket nu bare bliver vist via tekst.
Oprindeligt var alt koden placeret i den samme pakke, dette kan ses på UML diagrammerne fra fase 1 i appendiks B, siden da er systemet blevet lagdelt via 3-lagsmodellen, således præsentationslaget, forretningslogikken og datalaget er særdelt, derudover er der blevet implementeret acquaintance og gluecode, som sikrer høj vedligeholdelse og sikkerhed i koden til spillet. Fase 2 har haft høj fokus på at rette op på koden fra fase 1, hvor der blev lavet fungerende kode, har fase 2 fokuseret på at lave effektiv kode som gav mening, bl.a. ved at rykke spillerens swag ting og pung, samt hans nuværende position til spiller klassen selv, eftersom det gav mere mening at det lå der. I fase 2 skulle der desuden være mulighed for at gemme og indlæse spillet, derfor var flytningen af koden meget smart, da player klassen nu kunne nøjes med at blive gemt, som det eneste objekt på filen, da playeren indeholdte alle de informationer som skulle bruges til at indlæse spillet igen.

2.7.2 Det iterative arbejde

Projektet har været delt op i to faser. I fase 1 fik blev den grundlæggende funktionalitet for spillet lavet, alle krav for spillet kørte på en tekstbaseret version, som blev skrevet ud til brugeren i konsollen via `System.out.prints`. Dog var det også i denne fase, at gruppens medlemmer for alvor lærte at programmere, hvilket klart kan reflekteres i koden. Koden var meget rodet og det var hvad man ville kalde 'spaghettikode', desuden var der ikke lagt meget tanke bag koden, hvis det virkede, så var der ingen grund til at betvivle det. Siden da har gruppen i fase 2 udviklet sig markant, og er blevet mere bevidste omkring kodevalget, hele projektet er desuden blevet lagdelt, så der ikke længere er spaghettikode. Dette har resulteret i et meget mere overskueligt UML diagram for fase 2, som kan ses i Appendiks C.
Grundet mandefald i gruppen, kan det dog konkluderes at fase 1 var 100% færdiggjort i forhold til funktionalitet, modsætning til fase 2, der stadig mangler en del som gruppen har været nødsaget

til at skære fra, eftersom der simpelthen ikke var timerne til det. Efter fase 2 står gruppen derfor ikke med et færdigt produkt, men derimod en prototype som kan vise intentionerne for spillet.

2.7.3 Softwareprocesmodeller

Igennem forløbet af projektet, kan der ikke udelukkes bestemte softwareprocesmodeller, da der ikke er en model som beskriver præcis hvad gruppen har gjort, dog kan det ses at der er blevet fulgt en agil arbejdsmetode, eftersom gruppens medlemmer har arbejdet på deres del, og egentlig derfor har haft frie tøjler, på deres respektive område. Der har derfor ikke været nogen overordnet plan, som skulle følges udover de krav, som blev stillet af de forskellige faser. Dette modstrider hvad en plandreven arbejdsmetode fremskriver, derfor er der arbejdet mere agilt, dog har gruppen internt aftalt forskellige tidspunkter, hvor bestemte dele af koden skulle være færdig til, og disse tidspunkter er blevet overholdt, dette har også været nødvendigt for overhovedet at kunne nå, at komme så langt, som projektet er skredet frem i fase 2. Til at starte med blev alt arbejde dokumenteret, men i løbet som projektet skred frem, stoppede dokumentationen eftersom gruppen begyndte at arbejde på projektet på alle tidspunkter af døgnet, var det ikke længere tidsmæssigt forsvarligt, at blive ved med at skrive ned, hver gang der blev foretaget noget. De eneste modeller som er blevet udarbejdet er derfor også kun UML diagrammer, hvilket igen passer på en agil procesmodel, da man her ikke har den store dokumentation af arbejdet.
En metode som er blevet brugt meget i projektet er *"Extreme programming (XP)"*, hvor *"refactoring"* er blevet anvendt, da hele systemet skulle lagdeles, her blev der flyttet rundt i koden og forbedret den markant, som optimerede vedligeholdelsen og sikkerheden. Desuden er der brugt *"Pair programming"* hvor gruppens medlemmer har sat sig sammen og programmeret, hvilket har kunne medvirke til, at forbedre både forståelsen af koden, samt skabe et overblik. Derudover er der også brugt *"Small releases"* eftersom projektet startede ud med at få forretningslogikken til at fungere, herefter blev der tilføjet de to andre lag.

2.8 Fejl og mangler

Følgende afsnit vil belyse de fejl og mangler, som spillet indeholder, de fleste mangler som spillet har, skyldes gruppestørrelsen på blot to medlemmer. Fejlene kan have ændret sig siden rapportens afleveringsdato, eftersom man konstant bliver klogere, kan det være at et muligt fiks til nogle af spillets bugs, bliver opdaget.

2.8.1 Bugs i spillet

Interaktion med npcs

I fase 1 under den tekstbaserede version, fungerede interaktionen mellem spilleren og npcs (Non-Playable Characters), ved at man som spiller skrev i konsolen, ”interact” og derefter navnet på den npc som man ville kontakte, herved startede interaktionen op til den npc man skrev til. I forhold til koden, blev der tjekket om spilleren stod i et bestemt rum, fx Johnny Bravo rummet, og først derefter, blev der tjekket om den rigtige kommando og npc’s navn blev skrevet. I fase 2 skulle projektet forbedres med lagdelingen og den grafiske brugergrænseflade, disse ændrede på hvordan interaktionen fungerede. Grundet designet blev de tjek, som skulle sikre man snakkede med den rigtige npc, skåret ned til at blot tjekke for spillerens nuværende rum, og derefter om man trykkede på knappen ”snak med”. Dette har resulteret i, at det nu ikke er muligt at interagere med mere end én npc pr. rum, da det ikke er muligt at indsætte navnet for mere end en npc pr. rum pr. knap.

Løsningen til problemet ville derfor være et design spørgsmål, her kunne man fx implementere én knap pr. npc pr. rum, eller tilføje en tjekboks til hvilken specifik npc, man prøvede at snakke med, men på grund af mandetimer, har gruppen været nød til at leve med denne fejl.

Timer

Spillet har en timer funktion, selve funktionen af timeren fungerer fint, dog kunne funktionaliteten til at skifte scene til ”Game over scenen” når timeren løb ud, ikke implementeres grundet manglende tid, så derfor blev et voldsomt `System.exit()` nødsaget til at få denne funktionalitet til at virke, for at lukke programmet ned. Dette ville gruppen have kigget på, men grundet tidspres, måtte fejlen blive accepteret.

2.8.2 Mangler i spillet

Highscore

Et af kravene til spillet, var at der skulle gemmes en highscore til en fil, efter hvert spil. Denne highscore skulle så indlæses på et scoreboard, som skulle vise de top ti bedste forsøg af spillet. Denne funktion virkede også næsten, eftersom der blev gemt en highscore til en fil, efter hvert fuldført spil. Men pga. manglende gruppemedlemmer, blev gruppen nødsaget til at nedprioritere noget. Eftersom funktionaliteten for at gemme top 10 highscore og sorterere dem, ud fra hvilke scores der var de bedste, ikke kunne nås, droppede gruppen at færdiggøre dette, dog eksisterer klasserne (HighScore, Score og ScoreComparator) stadig i projektet, hvor forsøget på at få det til at virke er vist. Alt dette skulle så køres igennem datalaget, og her blive lagret på en fil.

Bevægelse mellem rum

Et andet krav som blev stillet, var at nogle npc'er selv skulle være i stand til at både interagere med spilleren, og bevæge sig imellem rum. I fase 1, interagerede npc'en Sidney Lee selv med spilleren, men på grund af implementeringen af den grafiske brugergrænseflade, virkede dette ikke længere, eftersom der ikke var nogen event-handling, som gruppen kunne finde ud af. Derfor måtte der endnu en gang blive nedprioriteret, og undlade denne funktion. I stedet blev interaktionen implementeret, ligesom resten af spillets npc'er, hvor man selv skulle manuelt snakke med Sidney Lee. For at kompensere, blev døren ind til det sidste rum "Hall of Fame" låst, og denne dør blev først låst op, når man havde vundet over Sidney Lee.
Selve bevægelsen af nogle npc'er havde gruppen tænkt sig at gøre, ved at få nogle modstandere fra en bande i Randers til at jagte spilleren, men denne implementering led også under frafaldet af gruppen.

GUI

Den grafiske brugergrænseflade var den del som led hårdest under gruppens størrelse, eftersom det var planlagt at lave animationer og sprites til spillet. Alt dette blev desværre udeladt, da det var langt over hvad var tidsmæssigt muligt at nå. Spillet skulle ellers have et design for hele byen, med animationer, så det ikke længere var tekstbaseret. Funktionaliteten af spillet skulle ikke styres via knapper, men med tastatur og mus inputs. Dette ville have forstærket det humoristiske aspekt

af spillet, som var det eneste krav som gruppen havde stillet til spillet. Derudover var det også planlagt at kampen med Sidney Lee skulle være en dance battle, og ikke et enkelt spørgsmål der skulle besvares. Ligeledes var det ikke planen at alle npc'ers missioner, skulle bygges op på samme skelet. Dog var animationer og sprites ikke et krav, stillet i opgavebeskrivelsen men noget som gruppen havde sat sig for.

3 Diskussion

3.1 Enums

Dette projektet har anvendt enums i form af, kommandoord (`CommandWord`). Enums er blevet anvendt da de er exceptionelt gode, hvis der allerede kendes instanser ved klassen i forvejen, som er tilfældet ved projektet. I projektet bruges et kommandoord, til at repræsenterer de forskellige ord såsom `GO`, `GET`, `WALLET` osv. Eksemplet på dette ses i Game klassen med `processCommand()` metoden, som er bygget op omkring en switch, hvori samtlige kommandoer bliver tilknyttet til hvert enum.

```
public String processCommand(Command command, String textInput) {
    CommandWord commandWord = command.getCommandWord();
    switch (commandWord) {
        case HELP:
            printHelp();
            break;
        case GO:
            goRoom(command);
```

Fordelene ved at benytte en switch, er at det er vedligeholdelsesvenligt, i forhold til den tidligere løsning fra fase 1, her blev der brugt adskillige if-else statements, som fik koden til at se meget rodet ud, et problem som gruppen gerne ville have elimineret i fase 2. Desuden er en switch meget overskueligt, i det det giver et hurtigt overblik over, hvilke metoder der bliver kaldt for hvert enum. Hvis der skulle sammenlignes med andre måder, at konstruere en overordnet metode til at håndtere spillets kommandoer på, kunne oprettelsen af et nyt interface implementeres. Dog ville måden at gøre dette på, resultere i oprettelsen af nye klasser, der skulle håndtere hver type kommando, hvilket ville være uoverskueligt, da kommandoerne ville blive spredt ud i mange klasser, fremfor at samle det i én enkelt klasse, som det er i tilfældet med switchen.

3.2 Arv

Arv findes mange steder i projektet, dog blev det først anvendt i NPC klasserne. Derefter blev der undersøgt, om diverse NPC klasser indeholdte navn, hilsen, interaktion og en mission, som også nævnes i polymorfi afsnittet. Eftersom at det meste af polymorfi afsnittet, også gennemgår funktionaliteten af arv, da arv deler mange lignende aspekter med polymorfi, vil der ikke blive gået i dybden med det. Det essentielle ved arv, er at de enkelte NPC underklasser extender fra superklassen NPC, hvori at metoderne der findes i superklassen kan blive anvendt i subklasserne. Ud fra dette kan det verificeres, at polymorfi og arv har nogle ens aspekter, dog skal det pointeres at forskellen mellem de to, er at arv bruges til at nedarve metoder fra superklasser, hvor polymorfi vedrører mere at anvende de samme metoder, men med forskellige implementering.

3.3 Polymorfi

Polymorfi bliver anvendt bl.a. ved NPC klasserne. Hver NPC subklasse til den abstrakte superklasse har deres egen implementering af `getName()`, `getGreeting()`, `isQuest()`, `setQuest()` og `interact()` metoderne. Fælles for alle klasserne, er deres opbygning, hvilket vil sige at metoderne bliver brugt på samme måde. Derfor er der blevet implementeret en abstrakt klasse frem for et interface. Da konstruktionen af metoderne er nogenlunde ens, da alle NPC subklasser indeholder en switch med et specifikt scenarie, kunne en abstrakt klasse bruges til at indeholde implementeringerne af metoderne, således dette ikke skulle gøres konsekvent i alle subklasser, derfor gav det mere mening at implementere en abstrakt klasse end et interface.

Ved brug af nedarvning fra den abstrakte superklasse `NPC`, vil subklasserne anvende de samme metoder, som superklassen har i sit arsenal. Implementeringen af `interact()` metoden, indeholder i alle tilfælde en switch bestående af forskellige stadier, hvis det første stadie opfylder de specifikke krav, skal det switchen springe til næste case, hvorimod hvis spilleren ikke opfylder kravene, bliver stadiet nulstillet og casen går tilbage til startpositionen. Hvis personen fuldfører den pågældende mission, vil der være en belønning af ekstra tid og tilføjelsen af en specifik swag ting, til spillerens inventar. Ulempen ved at bruge en abstrakt klasse er, at hvis der skal tilføjes nye metoder, skal disse metoder også implementeres i alle de pågældende klasser, som arver fra den abstrakte klasse. Heraf kan det konstateres, at brugen af abstrakte klasser kan anvendes til forskellige scenarier, fx kan brugen af interact dermed anvendes til hver enkel NPC, og opleve ny

adfærd for hver implementering af `interact()` metoden. Herunder ses et eksempel på et specifikt scenarie af `interact()` metoden.

```
@Override
public String interact(String textInput) {
    switch (interactionState) {
        case 0:
            interactionState = 1;
            game.removeSwag("Seddel fra Bjarne Riis");
            game.addSwag("EPO");
            return "EPO dealer: Nå det er den tid igen? Jeg sender en regning til ham.\n"
                    + "EPO dealer: *Giver dig en lille pose med EPO*.\n"
                    + "EPO dealer: Husk! Ikke snak med nogen før du har afleveret varen til Bjarne Riis.";
        case 1:
            if (game.getSwag("EPO") != null) {
            return "Du har allerede fået en pose EPO.\n"
                    + "Måske du skulle aflevere den hos Bjarne Riis.\n";
        }
        default:
            return "";
    }
}
```

I første case af switchen, fjernes swag tingen ”Seddel fra Bjarne Riis”, hvorefter swag tingen ”EPO” tilføjes til spillerens inventory, herefter printes noget tekst ud på skærmen. Denne eksekvering sker, da det kun burde være muligt at interagere med NPC’en, ”EPO Dealer”, efter man har påbegyndt missionen hos Bjarne Riis, og derfor allerede har ”Seddel fra Bjarne Riis” i sin inventory. Hvis du har modtaget swag tingen, ”EPO” fra NPC’en og forsøger at interagere med ham igen, så vil det udløse den anden case, som blot returnerer noget tekst, dette sikrer at man ikke kan gennemføre missionen flere gange.

3.4 Save/Load

Et af kravene for projektet var, at det skulle være muligt at gemme og indlæse spillet. Denne funktionalitet blev implementeret i `highscoreManager` klassen, som er beliggende i datalaget. Måden hvorpå det er muligt at gemme et computerspil, er ved at gemme et objekt og skrive det til en fil, som senere kan indlæses af Java, og konverteres om til et påbegyndt spil.
I dette projekt opretter `highscoreManageren` en ny fil, ved navn ”`GameSaver`” denne fil bruges til at gemme et objekt af `player` klassen. Årsagen til at det er et player objekt som bliver gemt, er som nævnt tidligere fordi `player` klassen indeholder alle de variabler, som er essentielle for spillet, med andre ord er de eneste variabler der skal gemmes, de variabler playeren indeholder.

```
public void savePlayer(Player player) {
    try {
        FileOutputStream outputStream = new FileOutputStream(PLAYER_FILE);
        ObjectOutputStream savePlayerStream = new ObjectOutputStream(outputStream);
        savePlayerStream.writeObject(player);
        outputStream.close();
        savePlayerStream.close();
    } catch (FileNotFoundException ex) {
        Logger.getLogger(HighscoreManager.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
        Logger.getLogger(HighscoreManager.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Herover ses metoden `savePlayer()`, som ligger i `highscoreManager`. Metoden opretter en `FileOutputStream`, der kan skrive til en fil, herefter opretter den en `ObjectOutputStream`, som skal bruges af FileOutputStreamen til at skrive et objekt ned på en fil. I dette tilfælde skrives der til filen `PLAYER_FILE`, som er `GameSaver` filen og objektet, som bliver skrevet ned på filen er `player` objektet.

Når spillet skal indlæses, kaldes `loadPlayer()` metoden, denne metode er identisk til `savePlayer()` metoden, dog skal der ikke skrives ned på en fil, men derimod indlæses fra en fil, hvilket betyder at der ikke længere er brug for et output, men et input. Derfor er konstruktionen af `loadPlayer()` metoden den samme, som `savePlayer()` metoden blot hvor output er udskiftet med input.

Fordelene ved `File-` og `ObjektStreams` er, at de er lette at bruge eftersom `player` klassen forsyner spillet, med alle nødvendige variabler og attributter, for at gemme og indlæse et spil. Ulemperne ved en `ObjektStream` er, at alle objekter som skal læses, er nødsaget til at implementere `Serializable`. Dette har resulteret i at der i projektet skulle oprettes en ny "`MyTimer`" klasse samt "`MyTimerTask`" klasse, som skulle extende henholdsvis `Timer` og `TimerTask` samt implementere `Serializable`, grundet `Timer` og `TimerTask` ikke extender `Serializable`, var dette nødvendigt. Herved kunne spillets `GameTimer` klasse bruge `MyTimer` og `MyTimerTask`, som nu var `Serializable`, og derfor var det nu muligt at gemme timeren.

3.5 Indkapsling

Objektorienteret programmering består af fire søjler, et af disse er indkapsling. Indkapsling vil sige at man sætter sine attributter til private, og igennem getter- og setter metoder kan man tilgå attributterne, og herigennem modtage eller ændre værdien af attributten. Både getter og setter metoder er public metoder, som derfor kan tilgås fra andre klasser. Dette er særligt smart da det muliggøre, at hver klasse har fuld kontrol over dets attributter, og der er derfor ikke andre klasser som kan ændre ved dem. Herved kan man sikre sin kode mere sikkerhed og pålidelighed, da der ikke kan ændres i værdierne fra andre klasser.

Yderlige fordele ved indkapsling er, at man herigennem bruger tankegangen omkring black boxes. Inden for programmeringsverden, er en black box et system eller objekt, som man kan give nogle inputs, hvorefter der vil blive genereret et output, uden man kender til implementeringen af black boxen. Et eksempel på dette kunne være en klasse, som skulle gemme adgangskoder til at logge på en bruger i et computerspil, disse adgangskoder er til at starte med, gemt på et HashMap uden access modifier, for at tilgå adgangskoderne for denne klasse, bliver der lavet en accessor metode. Senere bliver der besluttet, at det nok ville være smartere at ændre HashMappet til private, således det ikke længere er muligt at fx slette eller ændre adgangskoderne, fra en anden klasse. Men eftersom man bruger en accesser metode til at tilgå HashMappet, er dette intet problem, da man stadig vil kunne få fat i værdierne fra HashMappet fra en anden klasse, uden at vide hvordan HashMappet er implementeret, dette havde ikke været muligt uden en accessor metode. Så kort sagt sikrer indkapsling, at andre klasser ikke behøves at vide hvordan andre klassers data lagres og manipuleres, da de blot har brug for de offentlige getter og setter metoder, som er stillet til rådighed fra klassen.

I projektet er der brugt store mængder af indkapsling. Største delen af spillets attributter i alle klasserne er private, som kan blive tilgået af henholdsvis accessor og mutator metoder. Følgende eksempel fra projektet viser også dette via setter metoden, `setCurrentRoom()`.

```
public void setCurrentRoom(IRoom currentRoom) {
    this.currentRoom = currentRoom;
}
```

Denne offentlige `setCurrentRoom()` metode kan bruges af andre klasser, til at sætte spillerens nuværende rum. Metoden tager `IRoom currentRoom`, som parameter og derigennem opdatere den spillerens nuværende rum, ved at sætte spillerens nuværende rum lig med et nyt rum, som så bliver spillerens nye nuværende rum. Dette er en af de mest væsentlige setter metoder i spillet, da det er ved hjælp af denne metode, at spilleren kan bevæge sig imellem spillets rum. Til at starte med brugte metoden almindelige `Room` objekter, men efter spillet fik implementeret et `IRoom` interface, gav det mere mening at bruge `IRoom` som parameter. Det geniale ved dette er, at andre klasser ikke behøver at bekymre sig om implementeringen af denne metode, andet end de blot skal sende værdier til metoden. Derfor skulle der ændres minimalt i koden, ved overgangen fra `Room` til `IRoom` objekter. Man kan ved brug af indkapsling derfor øge et systems vedligeholdelsesevne, eftersom en ændring af en implementering, kun påvirker resten af koden minimalt.

3.6 Vedligeholdelsesevne

I første fase af projektet, var størstedelen af spillets logik placeret i `game` klassen. Alle nye metoder blev oprettet i `game` klassen, uden at tænke på om det gav mere mening, at have det placeret i andre klasser. Eksempelvis var der metoder til at håndtere spillets kommandoer, som skulle udskrives i konsollen, der var metoder til at oprette spillet som helhed, samt ArrayLists til at indeholde items og mønter, og mange flere metoder. Sidenhen blev gruppens medlemmer bedre programmører, derfor blev projektets metoder delt ud i mange flere klasser, som hver styrede en tildelt funktionalitet. `Timer` klassen, `Coin` klassen, `Player` klassen og mange flere blev tilføjet, dette fandt gruppen ud af var meget smartere i forhold til vedligeholdelsesevnen, da det gjorde det meget nemmere at overskue koden, herved højnede det vedligeholdelsesevnen, eftersom at hvis der skulle ændres i hvordan playeren virkede, skulle der blot ændres i `player` klassen, og alle klasser som brugte playerens metoder, skulle derfor ikke have ændret andet, end højst deres parametre.

3.6.1 Lagdelt arkitektur

Den største faktor i projektet, som har ændret ved vedligeholdelsesevnen, er dog den lagdelte arkitektur. I første fase havde projektet kun ét enkelt lag, dette er sidenhen blevet udvidet til tre

lag, ved brug af 3-lagsmodellen. Ved at lagdele systemet, muliggøres det at alt kan ændres i hvert lag, uden at det påvirker de andre lag, dog kun så længe hvert lags interface opretholdes. Dette er særligt smart, da man herved kan dele mange af arbejdsopgaverne ud til hver enkel programmør. Projektet er ikke det største, men alligevel har gruppen kunne drage nytte af at kunne dele opgaverne ud. Et gruppemedlem har stået for meget af forretningslaget og et andet har stået for præsentationslaget. Ved at have lagdelt arkitektur, kan der derfor blive arbejdet uafhængigt mellem programmørerne, uden at påvirke eller komme til at ødelægge noget af den kode, som der arbejdes på, da man som programmør blot skal sikre sig, at overholde den kontrakt som interfacet giver. I et større projekt, vil dette være af enorm nytte, da udviklingstiden af et projekt ville tage markant længere tid, hvis hele holdet af programmører skulle arbejde på det samme, derfor kan man ved hjælp af lagdeling, dele flere hold af programmører ud til hvert lag, og herfra kan der så aftales hvem laver hvad inden for hvert lag.

I tilfældet hvor der er blevet arbejdet med acquaintance, sikres det at have alle interfaces samlet i et enkelt sted, og hvert lag har en facade, som implementere de interfaces som skal bruges. Dette er en udvidelse til 3-lagsmodellen, da man herved kan sikre sig igennem gluecode, at hvert lag ikke har en reference til hinanden, men blot til acquaintance laget, som indeholder alle systemets kontrakter. Dette øger ikke blot vedligeholdelsesevnen, men også sikkerheden af koden markant, da man udover at sikre at hvert lag, ved hvad de skal kunne tilbyde i form af kode, også sikre, at man ikke kan tilgå andre pakker end sig selv og acquaintance pakken. Dette gør det umuligt for nogen udefra, at finde ud af hvordan bestemte metoder, som bliver kaldt af events i præsentationslaget er implementeret, da det eneste kald der bliver lavet, er et kald til et interface, som ikke indeholder nogen implementering. Dog har acquaintance været meget svært at arbejde med, da forståelsen af dette ikke er helt så nemt at overskue, derfor har gruppen skulle bruge en del tid på at sætte sig ind i denne form for arkitektur.

Vurderingen af lagdelingen, er derfor at det har højnet hastigheden, hvormed gruppen har kunnet producere ny kode, forbedret sikkerheden af koden markant, og ikke mindst sikret en væsentligt bedre vedligeholdelsesevne, som der især kunne drages nytte af, hvis man på et senere tidspunkt følte for at færdiggøre nogle af de fejl og mangler spillet, på nuværende tidspunkt indeholder, eller blot tilføje helt ny funktionalitet.

4 Konklusion

Lige fra projektets begyndelse, blev det meste af det oprindelige Zuul framework anset for ikke at være praktisk, samt ustabilt, af denne årsag blev det meste af frameworket udskiftet undervejs i projektet, heraf opstod der nye ideer i henhold til de forskellige udskiftninger. Dette blev diskuteret i gruppen og herved udført, i henhold til medlemmernes kapacitet og evne. I første fase var de fleste af metoderne tilføjet i `game` klassen, da dette var upraktisk, blev de i stedet for implementeret i andre klasser, eller der blev oprettet nye. Samtidig blev der oprettet nye metoder der udbyggede spillets funktioner yderligere.

Spillet var et tekstbaseret spil og grundet mandefald fortsatte spillet således, til gengæld er der blevet tilføjet SceneBuilder der har medført, adskillige skiftende scener i forhold til før. I scenerne er der blevet implementeret knapper, der kan kommunikere med spillet i stedet for at skrive brugerinput i konsollen. Denne udvikling har forbedret spillerens vilkår markant, da de ikke skal huske alle spillets kommandoer. Gruppemedlemmerne har igennem projektet udvidet deres kunden gevaldigt, ved eksperimentering af projektet, undervisningsforløbet og dets tilegnede opgaver såsom Matador projektet og Workshoppen.

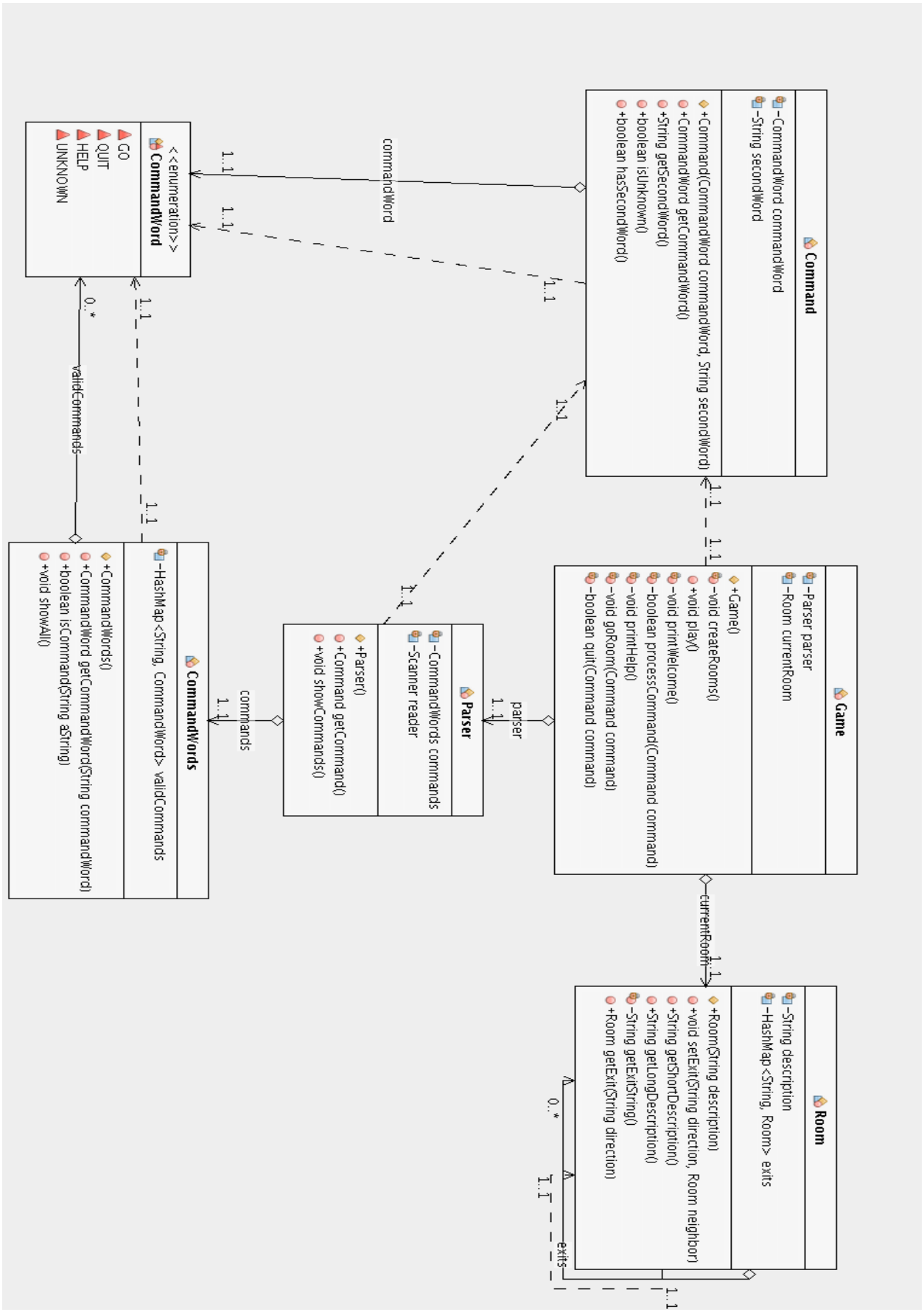
Hvis der havde været mere tid og ressourcer, ville der have blevet etableret flere funktioner, som kan ses i afsnittet *"fejl og mangler"*. Derudover er spillet blevet inddelt i en lagdeling, hvori det har forhøjet vedligeholdelsesevnen af projektet, og gjort det markant lettere at arbejde i præsentationslaget. Ydermere er der blevet implementeret adskillige nye funktioner i spillet, disse er som følgende: Genstande som spilleren kan samle op, NPC'er der kan interageres med, som samtidig indeholder diverse missioner der kræver en specifik handling for at færdiggøre missionen, tidsparameter der begynder ved spillets start, og kan forøges via missioner, save/load funktionalitet af ens karakter og slutteligt et pointsystem. I sammenligning med det oprindelige visionsdokument, er der meget som måtte undværes, da planen var tiltænkt 6-7 mennesker og ikke 2, men ligeledes kom der andre aspekter i spillet, såsom udeblivelsen af Randers typerne og animationer, samt NPC'ernes bevægelse mellem rum. Til gengæld er der blevet tilføjet to inventory's, som gælder coin og swag, disse var ikke med i det oprindelige visionsdokument.

Dermed kan der konkluderes, at arbejdsprocessen har været fyldestgørende, da der er blevet nået mange ting i forhold til gruppens situation. Samtidig har udviklingsprocessen været stigende, da projekt blev anset som en læringsoplevelse, fremfor fremstillingen af et produkt.

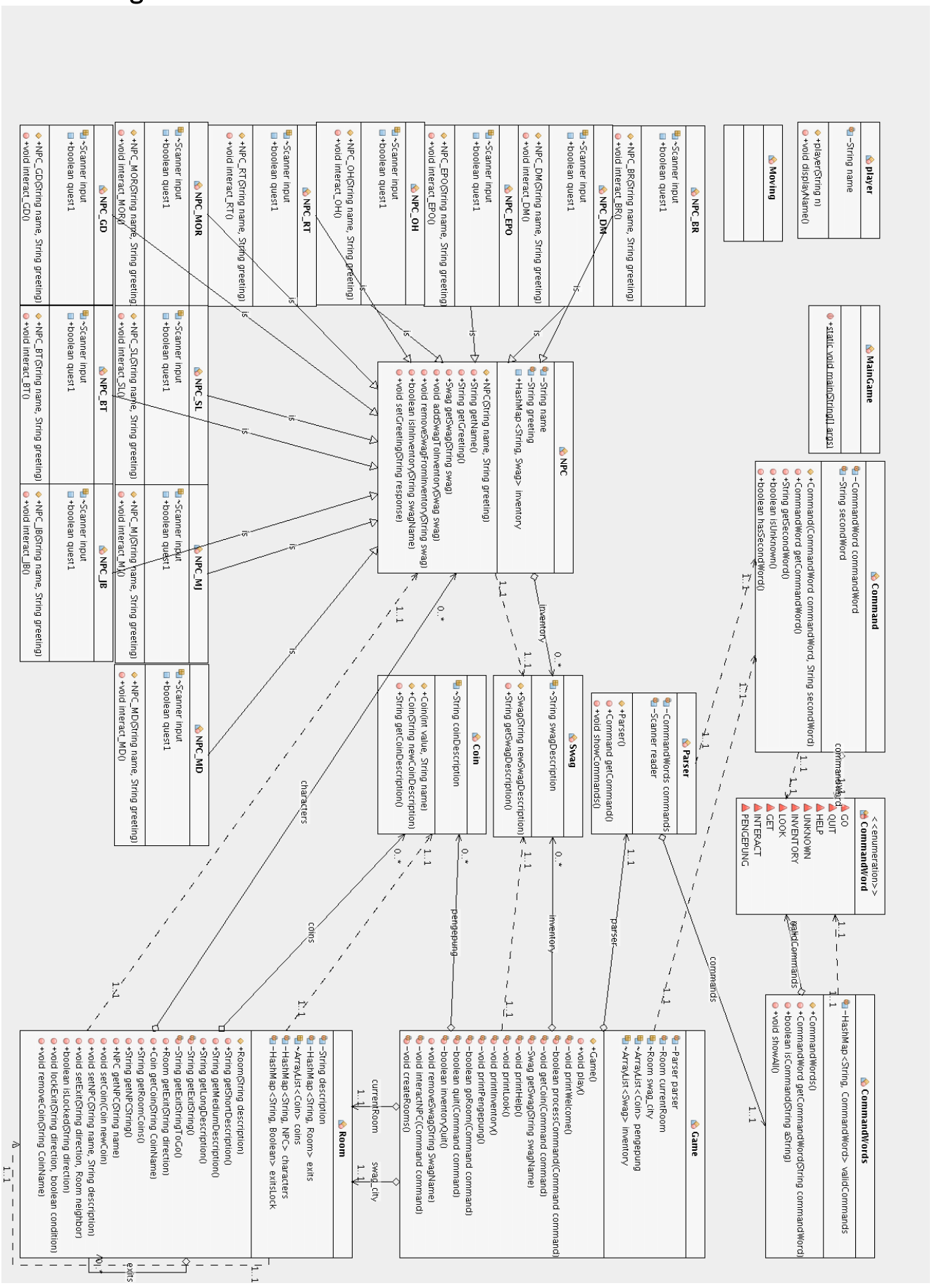
5 Perspektivering

Hvis gruppestørrelsen ikke havde gået fra i alt syv gruppemedlemmer til to, havde gruppen nok kunne nå markant mere, kombineret med endnu mere tid, ville der højst sandsynligt kunne slettes alle fejl og mangler, da alt dette kunne have været implementeret. Særligt implementeringen af animationer og sprites, kunne have forbedret spilleoplevelsen markant, da dette ville kunne højne den humoristiske del af spillet. Samtidig kunne man gjort spillet endnu bedre vil at tilføje flere funktioner og features, og generelt bare have udvidet spillet med flere npc'er, rum og missioner. Taget i betragtning af gruppens størrelse, ville gruppen nok stadig havde ændret på en del ting, hvis projektet skulle startes forfra. Særligt efter man er blevet en del klogere, ville projektet nok starte med at få en lagdelt arkitektur, og derfra definere hvilke interfaces spillet skulle indeholde, og hermed hvad spillet skulle kunne. Derudover er gruppens medlemmer bedre rustet til nu end før, at arbejde individuelt med koden, uden at skulle have brug af både hjælpelærer og internettet. Ved hjælp af dette projekt er gruppen kommet meget tættere på programmeringsverden, og man ser nu mere verden i objekter og klasser, og kan bedre se sig ud af et programmeringsproblem, derfor er dette projekt set mere som et læringsforløb, end fremstillingen af et produkt.

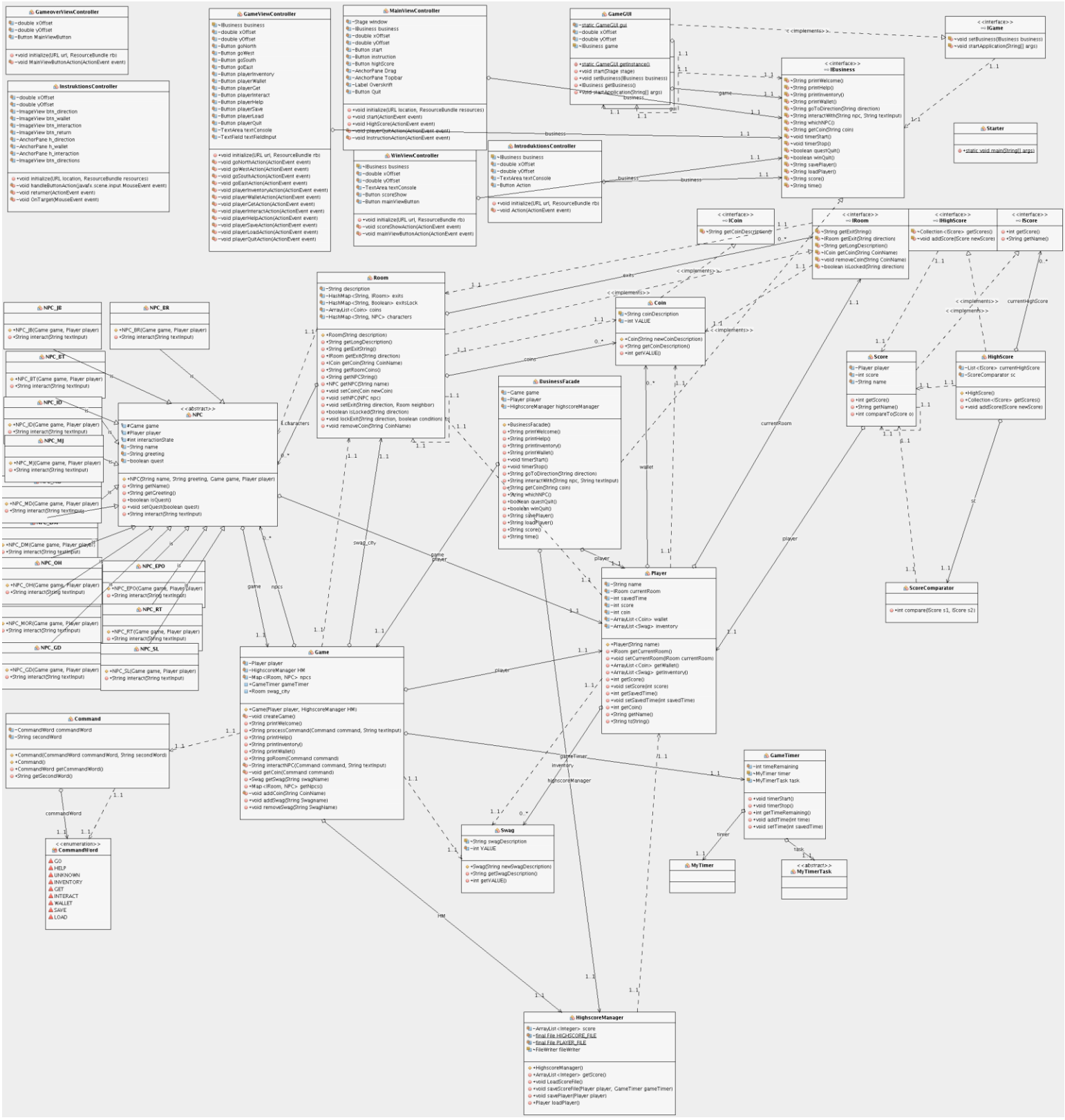
A. UML diagram: Zuul framework



B. UML diagram: Fase 1



C. UML diagram: Fase 2



D. Rapportkontrolskema

Kapitel	Krav	Opfyldt +/-
Omslag	Rapportens titel, projektgruppe, (fornavn, efternavn, SDUbrugernavn), rapporttype, vejleder(e), hvor og hvornår rapporten er afleveret (kursus/studieaktivitet, institut, uddannelsesinstitution, projektperiode, årstal)	+
Titelblad	Som omslag ekskl. evt. illustration + evt. kildehenvisning til evt. omslagsillustration (Omslaget kan udgøre både omslag og titelblad. Hvis der medtages selvstændigt titelblad, så er titelbladet rapportens første højre side)	+
Resume/abstract	• Problem – hvilken problemstilling arbejder I med i jeres projekt og hvad er jeres ide • Fremgangsmåden - hvordan angreb I problemet og hvordan realiserede I løsningen (hvem, hvad, hvornår og hvorfor) • Løsning – hvad er jeres løsning Omfang af resume: ca. 1 side	+
Forord	Hensigten med rapporten, målgruppe, forhistorie, anerkendelser, afleveringsdato, underskrifter af alle forfattere	+
Indholdsfortegnelse	Samlet indholdsfortegnelse for hele projektrapporten. Højst tre niveauer i indholdsfortegnelse, men der kan evt. flere i selve rapporten.	+
Læsevejledning	Er der en vejledning i, hvordan rapporten kan læses, eksempelvis i form af en hvilken rækkefølge hovedrapporten kan læses og hvordan sammenhængen er mellem hovedrapport og bilag?	+
Ordliste	Kort beskrivelse af der bruges gennem rapporten fagtermer	+
Indledning	Baggrunden for projektet (problemstilling), problemformulering og afgrænsning.	+
Hovedtekst	Alle delene under Hovedtekst i 8.1 er inkluderet.	+
Diskussion	Hvad er styrkerne og svaghederne ved jeres løsningen? Kunne I have anvendt andre og eventuelt bedre løsninger i jeres kode?	+
Konklusion	Sammenhold med jeres visionsdokument og problemformulering, hvad har I nået og hvad har I ikke nået?	+
Perspektivering	Future work: • Hvad ville de næste skidt i projektet være, hvis I havde mere tid? Refleksion: • Hvis I startede projektet igen i morgen, hvordan ville I så gribe det an? • Hvilke tekniske problemstillinger opstod undervejs og hvordan ville I håndtere dem med den viden og de kompetencer I besidder nu?	+

E. Rapporttekniske elementer

Rapporttekniske elementer		Opfyldt +/-
Layout	Er der anvendt samme layout i alle kapitler? Er layout overskueligt/harmonisk?	+
Sprog	Er rapporten skrevet i en neutral sprogtone? Er sproget let læseligt og flydende? Er der udført stavekontrol og kontrol af tegnsætning?	+
Sidenummerering	Er der korrekt og konsistent sidenummerering i rapporten?	+
Underskrifter	Er alle projektdeltageres personlige underskrift i afsnittet ”Forord”?	+
Figurer/diagrammer	Er Er alle figurer konsekvent nummererede? Er der figurtitel og figurtekst til alle figurer? Er figurtitler og figurtekster dækkende og afklarende? Er figurerne tydelige og læsbare? Er figurerne informationsgivende og i den rette sammenhæng?	+
Tabeller	Er alle tabeller konsekvent nummererede? Er der en forklarende tabeltekst til alle tabeller? Er alle søjler og rækker forsynet med parametre? Er der enheder på alle relevante rækker og søjler?	+
Litteraturliste	Er litteratur angivet på en anerkendt form? Er alle former for litteratur som bøger, artikler og hjemmesider medtaget? Er der kildehenvisninger i teksten? Materiale som gruppen ikke selv har fremstillet i dette projekt skal være angivet med kilde! Er alle kildehenvisninger i teksten anført på samme måde? Er der kildeangivelser på figurer, grafer etc. som projektgruppen ikke selv har frembragt?	+
Sporbarhed af begreber	Er der en konsekvent brug af samme betegnelse for et givet begreb igennem rapporten?	+