Gnomes and Gnoblins

Semesterprojekt

Aarhus Institue of Technology

Author:
Morten Høgsberg, 201704542

Date: May 25, 2022

Contents

1 Introduction

1.1 Spilbeskrivelse

PC = Player Character RNG = Random Number Generator
Spillet designes som et text-based spil, det er en spilgenre hvor brugeren interagerer med spillet igennem tekst beskrivelser. Spillet består overordnet af fire dele med hver deres ansvar, en grafisk brugergrænseflade, en database, en back-end til netværkskommunikation samt selve spil logikken. Brugergrænsefladen gør det muligt for brugeren at integrere med spillet. Den vil bestå af en række forskellige vinduer med hver sit formål (login screen, gameplay screen og settings screen), der hovedsageligt vil indeholde knapper og beskrivende tekst. Databasens ansvar er at gemme de enkelte spil, det er en relationel database som sammenholder de enkelte brugeres profiler (Username og password) med informationer omkring deres fremskridt i spillet, såsom placering, items og stats. Det skal være muligt for en bruger at hente sine gemte spil ned på flere forskellige enheder, til dette skal der bruges en netværksforbindelse til databasen.

1.2 Login-screen

Det første brugeren bliver mødt af en login-screen. Herfra skal brugeren enten indtaste sine login-oplysninger eller oprette sig en profil i systemet. Systemet har en database hvor alle profiler er lagret. Herunder ses en illustration af spillets login-screen.

Figure 1

1.3 Core gameplay layout

PC kommer ind i et rum (se billede 2). Brugeren bliver præsenteret med en beskrivelse af rummet, en liste af elementer i rummet, og en række muligheder for at interagere med rummet (interaktionen foregår ved at trykke på knapperne markeret "Button" i billede 2). Når brugeren er færdig med at interagere med rummet, forlader de det ved at vælge en retning ("go north/south/east/west"). Dette fører dem ind i et nyt rum, mappet opdateres og loopet starter forfra. Bevægelsen i spillet kan gøres i de forklarede fire retninger. Disse retninger indikerer for spilleren hvilke rum de kan bevæge

sig ind i relativt til det rum de er i, der kunne f.eks. være rum, som kun har en vej ind og en vej ud, så kan man ikke gå andre veje end vejen man kom ind. North/south/east og west retningerne er baseret på et kompas, så derfor ville North resultere i at bevæge spillerens karakter opad, South nedad osv. Det tilhørende map image.....(diskuteres) Rum på mappet loades efethånden som man besøger dem.

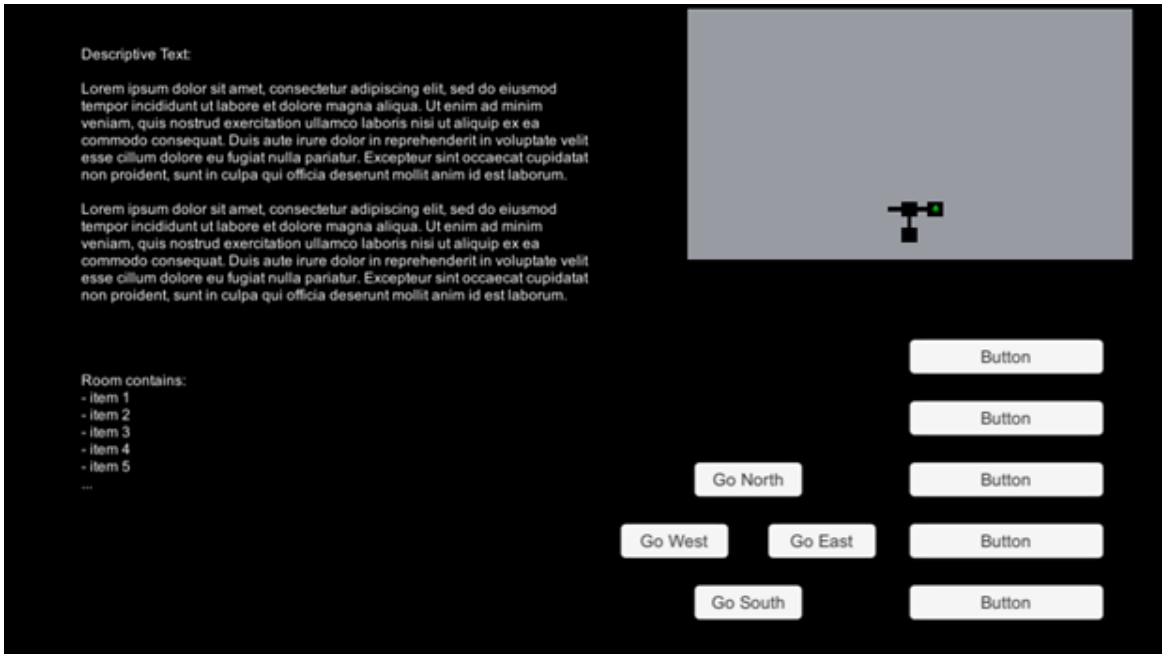

Figure 2

Et rum kan indeholde fjender, som skal bekæmpes før man kan tilgå resten af rummet. Kamp foregår ved at spillet "slår en terning" (RNG) for både PC og fjenden og lægger deres "combat stat" til slaget. Den som slår højest, giver en vis mængde skade til modstanderen, afhængigt af deres "damage stat". Denne skade trækkes fra karakterens liv. Når en karakters liv bliver mindre eller lig 0 dør karakteren. Hvis spiller karakteren dør taber brugeren spillet. Hvis hverken PC eller fjenden er død efter en kamp får brugeren valget mellem at flygte (gå tilbage til det rum de kom fra) eller fortsætte kampen (start loopet forfra). Det er muligt at finde våben og udrustning i banen, som kan bruges til at forbedre karakterens stats.

1.4 Settings

Det skal være muligt for spilleren at tilgå en menu med spillets indstillinger. Her skal det være muligt for brugeren at tilpasse spillets layout indstillinger så som resolution og window size. Det skal ligeledes være muligt at justere lydstyrken for spillet. En illustration af hvordan denne menu kunne se ud er vist på figur x.

2 Analyser

2.1 Teknologiundersøgelse: Unity - .Net

I projektgruppen har vi identificeret at der er to oplagte frameworks, til at lave projektet i. Det første framework er Microsofts eget, nemlig "WPF" til frontend og "ASP.NET" til vores backend. Det

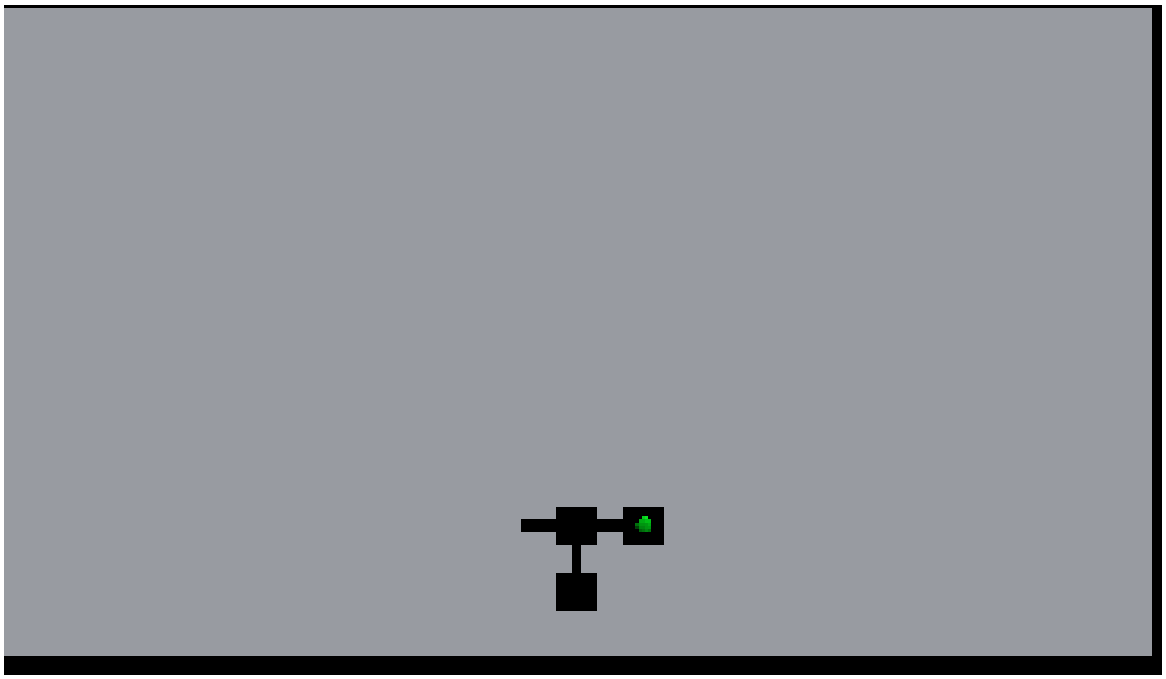

Figure 3: Closeup Screenshot af mappet

andet framework er Unity Technologies' framework "Unity" til kombineret frontend og backend. Ens for de to framework er at de begge understøtter blandt andet: - Understøtter C# - Versionskontrol i form af Git - integrerede/let integrerbare Testframeworks

2.1.1 Unity:

Unity lader dig bygge fulde spil fra bunden, dette giver rig mulighed for udvidelse og ændringer af spil designet, dette kombineret med at Unity er bygget til cross-platform kompilering, hvilket gør at ens spil automatisk bliver portabelt og ens målgruppe bliver udvidet og derved bliver markedsføring af produktet mere bred. Dette har naturligvis en pris, i form af at en Unity licens til firmaer med en årlig omsætning på over $100.000, kosterminimum1.800$. Derudover er Unity umiddelbart intuitivt og nemt at bruge, med det menes der at Unity's frontend er informativ, dog er hele frameworket nyt og skal derfor læres fra bunden. Dette resulterer i at der skal undersøges mange basale ting for at der kan laves et spil, og tiden det tager at lave første iteration kan hurtigt eksplodere. Unity er også et program i aktiv udvikling og det betyder for det første at features hele tiden bliver udviklet og udvidet. Dette resulterer også i at dokumentationen omhandlende Unity hele tiden skifter og derfor kan det være svært at søge hjælp på nettet, når man sidder fast i udviklingen. Unity har udover sit udviklingsværktøj, en Asset store, som giver mulighed for at købe allerede programmerede Assets og trække dem direkte ind i ens program. Dette giver selvfølgelig mulighed for at spare en masse arbejde, men da projektarbejde netop handler om at lave tingene selv, giver dette ikke meget mening at bruge i dette tilfælde. Et negativ ved at bruge et så gennemført og feature rigt udviklingsværktøj, er at kompilerings, loading og test tider bliver markant højere end ved brug af lettere udviklingsværktøjer. Til sidst skal der nævnes at udvikling i Unity ikke bliver opdelt i frontend/backend, sådan som det bliver lært at udvikling skal opdeles på 4. semester, Tværtimod sidder frontend/backend meget tæt sammen, når man udvikler med Unity.[**Unity.com**]

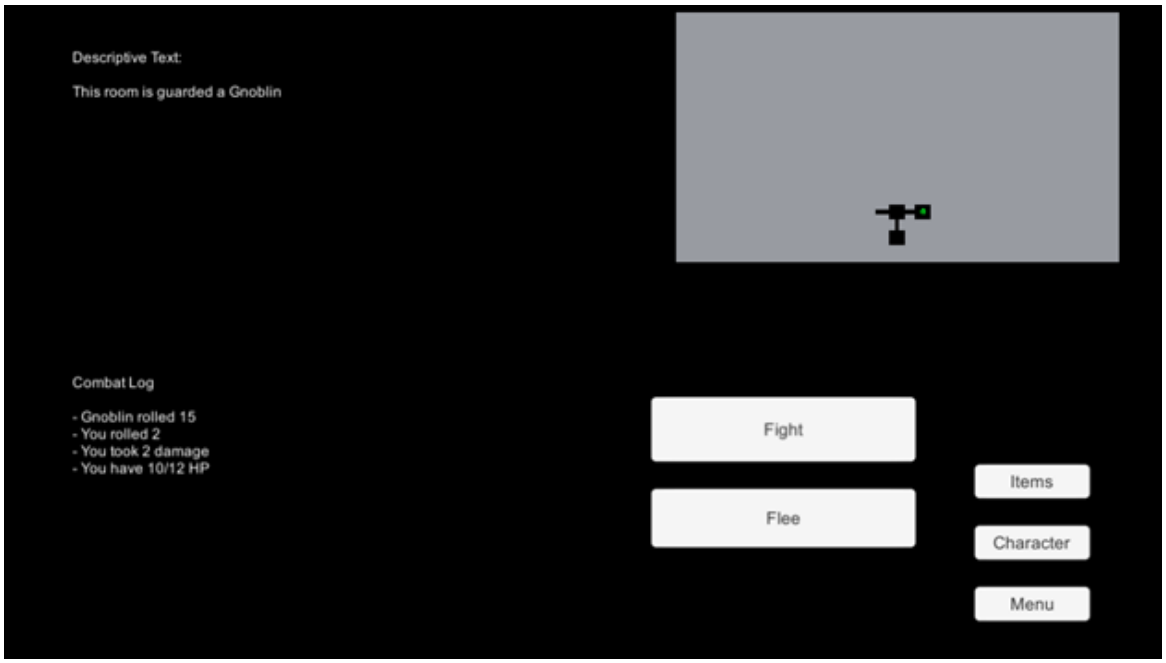

Figure 4

2.1.2 .NET framework:

Til forskel fra Unity, er .NET Frameworket noget mere letvægt. Spillet kan stadig laves fra bunden og stadig med rig mulighed for ændringer og udvidelse af spil designet. Hvad man ikke får, er Unity's indbyggede Cross-platform kompilering og derved kan denne del af ens udvidelses fase godt blive mere kompliceret, hvis man vil kompilere til flere styresystemer. At man misser den indbyggede Cross-platform kompilering, og Unity's Asset store er også med til, som sagt, at gøre .Net frameworket mere letvægt, det vil sige at man kan se frem til mindre overhead og derved potientielt hurtigere load og compile tider. Kigger man på de mere økonomiske forskelle, ser man hurtigt at de to frameworks er som to modsætninger, nemlig fordi at .NET frameworket kan fås gratis som extension til Visual Studio, som der også findes gratis versioner af. Ser man på .NET frameworket gennem mere praktiske øjne for os som studerende, dukker der umiddelbart tre argumenter frem. 1. Vi ved fra undervisningen, hvor vi har arbejdet med netop .NET frameworks, at der er massere af kendt viden at hente, altså dokumentation at hente på nettet og fra vores undervisere, om hvad og hvordan vi bedst kan gribe lige netop dette framework an. 2. Frontend og Backend er i .NET framworket separeret, hvilket passer rigtig godt med vores projektarbejde i projektgruppen og med hvordan vi har lært at man skal udvikle software systemer i undervisningen, nemlig lige præcis med den separerede frontend og back end. 3. Sidst men ikke mindst, har vi også tidligere arbejdet med hvordan CI (Continious Integration) kan sættes op, netop sammen med .NET frameworket. En opsumering af de sidste 3 punkter giver altså at vi i .NET framework, på trods af at det er mere letvægt end unity, kan fokusere mere intenst på at lave noget godt projektarbejde og kode, og ikke fokusere så meget på de basale ting, som vi måske skulle hvis vi arbejdede med Unity.

2.2 Frontend Arkitektur

Frontend applikationen vil have til formål at håndtere brugerinput og output. Dvs. at der i Frontenden vises det data fra gamelogic, som brugeren skal have, og at det præsenteres på en overskuelig og brugervenlig måde. Dette resulterer i at brugeren kan forstå og kan finde ud af at bruge spillet på

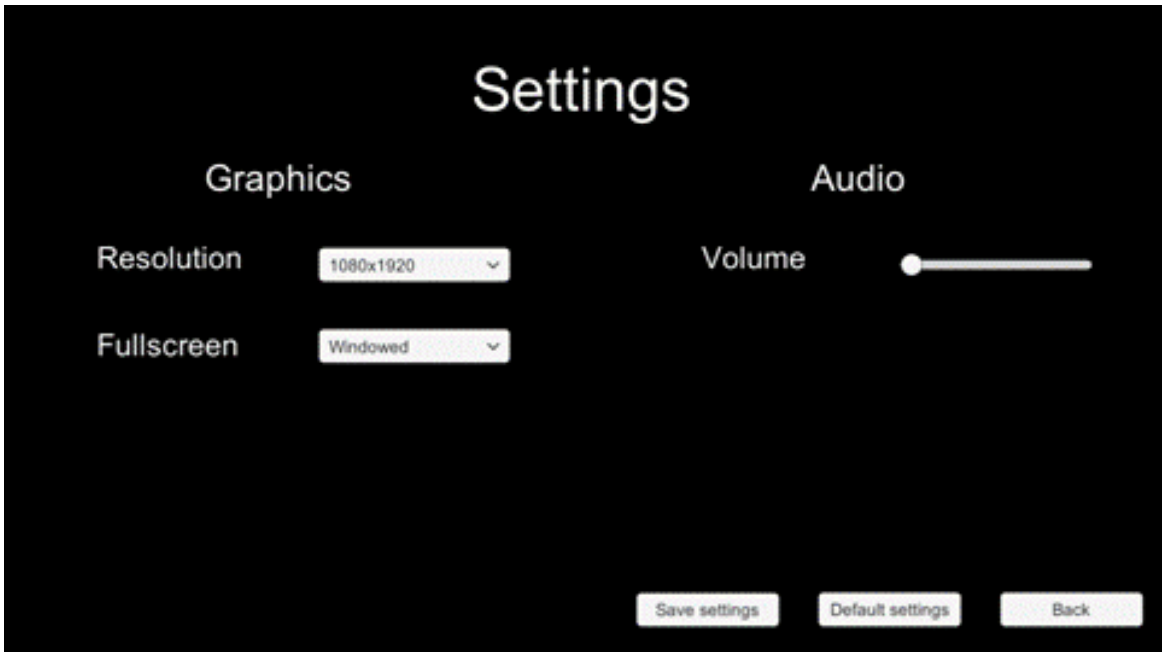

Figure 5: Screenshot af settings menuen

den tiltænkte måde. Derudover skal Frontenden tage hånd om bruger input, og sørge for at brugeren giver korrekt input og at der tages hånd om eventuelt forkert input.
Da der er mange forskellige menuer og skærme i spil i frontenden er der lavet følgende C3 model for Frontenden (Figure 6), som giver en idé om hvilke skærme og menuer der kan gå til hvilke andre menuer/skærme. Udover dette fortæller modellen også om hvordan og hvilke skærme og menuer der snakker med noget uden for frontenden selv f.eks. skal der ved Login/Register kontaktes databasen for at få verificeret logind-oplysninger, og samtidig skal der ved save og load-game hentes en liste af gemte spil i databasen hvorefter der skal henholdsvis skrives og hentes fra databasen alt efter om man gemmer eller henter et spil. Der skal hertil nævnes at Login og Register står til at snakke med backenden direkte og ikke igennem gamelogic blokken, dette er valgt da funktionen ikke kaldes i gamelogic blokken, men den kaldes direkte i backend controlleren. Derudover er modellen mere overskuelig på denne måde og fungerer bedre til at give overblik over navigationen igennem menuerne.

2.2.1 Pseudo Frontend Arkitektur

(Hovedrapport stuff) For at give overblik over, hvordan kommunikationen mellem frontend, backend og gamecontroller kommer til at foregå, er der lavet et pseudo sekvensdiagram for følgende UserStories:
- Login
- Register
- Save Game
- Load Game

(/Hovedrapport stuff) Der er ikke lavet sekvensdiagrammer for alle af projektets userstories, da mange af disse fungerer på samme måde og derfor ikke bidrager med noget nyt ift. dokumentationen. Nedenfor ses pseudo sekvensdiagrammer for de fire userstories:
- Login
- Register
- Save Game

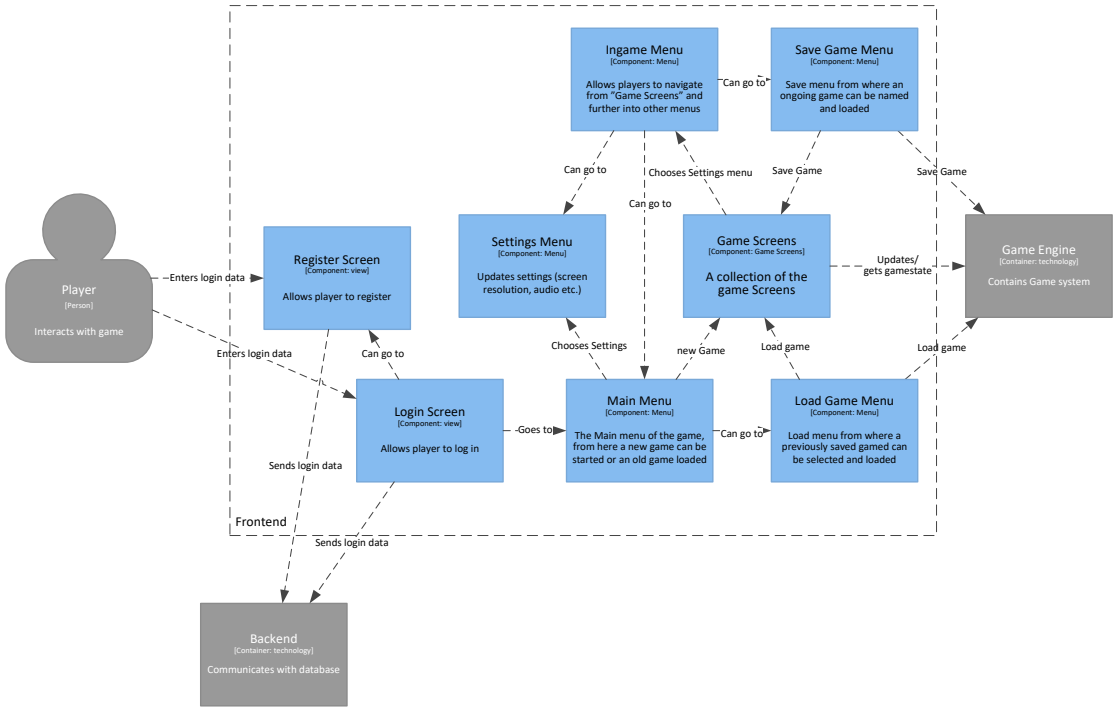

Figure 6: C3-Model for Frontend. Modellen fortæller hvordan man kan navigere igennem forskellige menuer og hvilke menuer der kan føre til hvad. Derudover kan man se hvilke blokke der snakker ud af frontenden og sammen med resten af systemet.

- Load Game

Først ses "Login"(Figure 7), som viser forløbet af userstory "Login", med en reference til userstory "Register", hvis brugeren ikke er registreret i forvejen. Udover dette er der ydermere vist håndtering fejlet login. Det skal her nævnes at brugeren kan ikke spille spillet uden at logge ind først, der tillades ikke at spillet kan spilles i Offline-tilstand.

Dernæst ses "Register" (Figure 8), som viser forløbet af hvordan en bruger kan registrere sig med en profil i systememt. Der kan ses på Figure 7, at hvis en bruger ikke er oprettet skal man gøre dette først. Derefter skal man vælge et brugernavn og kodeord, hvis brugernavnet er ledigt og alt ellers går godt, kommer man tilbage til "Login" skærmen. ellers får man en fejlbesked på skærmen og bliver bedt om at prøve med et andet brugernavn.

På Figure 9 ses "Save Game", som viser forløbet når en bruger gerne vil gemme sit igangværende spil, set fra Frontends perspektiv. Her kan man bide mærke i, at når der skiftes skærm, vil den nye skærm få initialiseret sine variabler i sin constructor og derfor er der kun et selv kald hver gang skærmen skiftes. Hertil skal der nævnes at hvis brugeren er i "Combat State" er det ikke muligt at gemme spillet og knappen "Save Game" på "In Game Menu" vil ikke kunne ses eller bruges.

Tilsidst kan der på Figure 10 ses "Load Game", som viser forløbet når en bruger gerne vil hente et tidligere gemt spil fra databasen, set fra Frontends perspektiv. Denne userstory minder på mange måder om "Save Game", men i stedet for at sende et element til databasen, skal der hentes et gemt

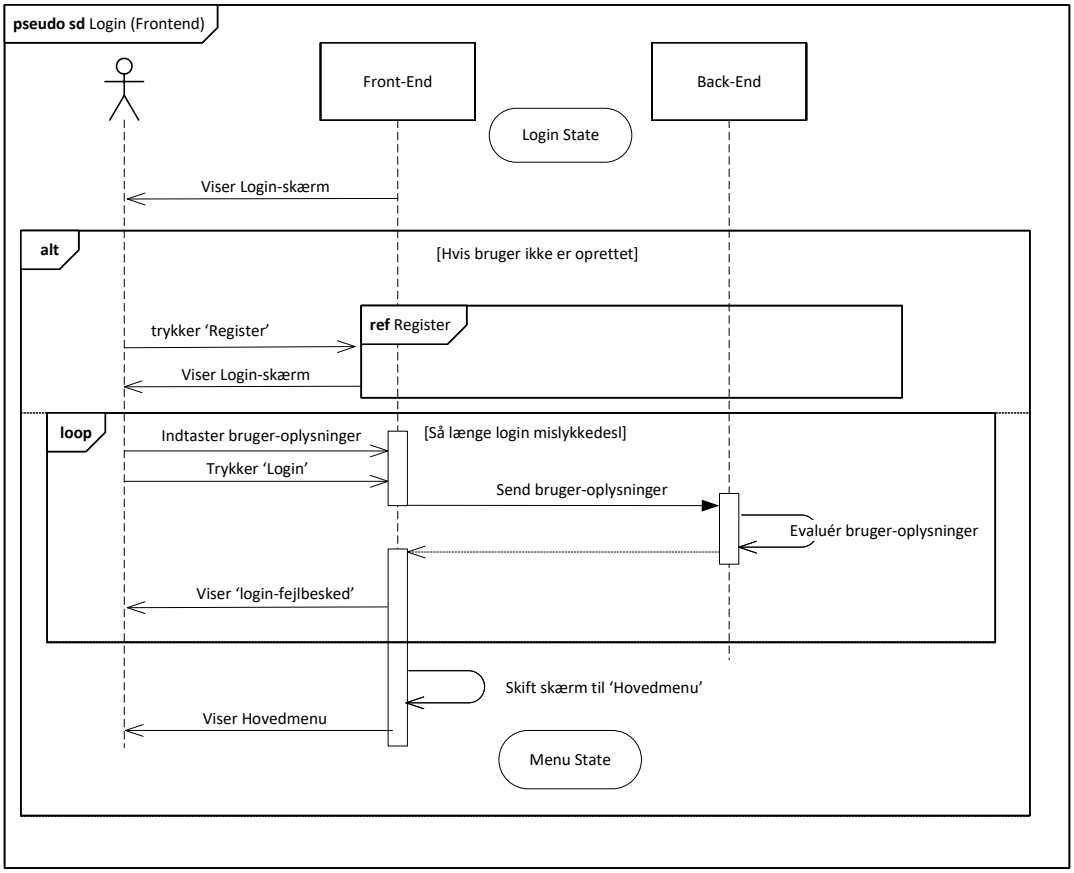

Figure 7: Pseudo sekvensdiagram af forløbet af userstory "Login", set fra Frontends perspektiv. Med reference til "Register" userstory og håndtering af forkerte login oplysninger.

spil fra databasen, med alle de data der skal bruges for at kunne loade sit spil op præcis som man forlod det.

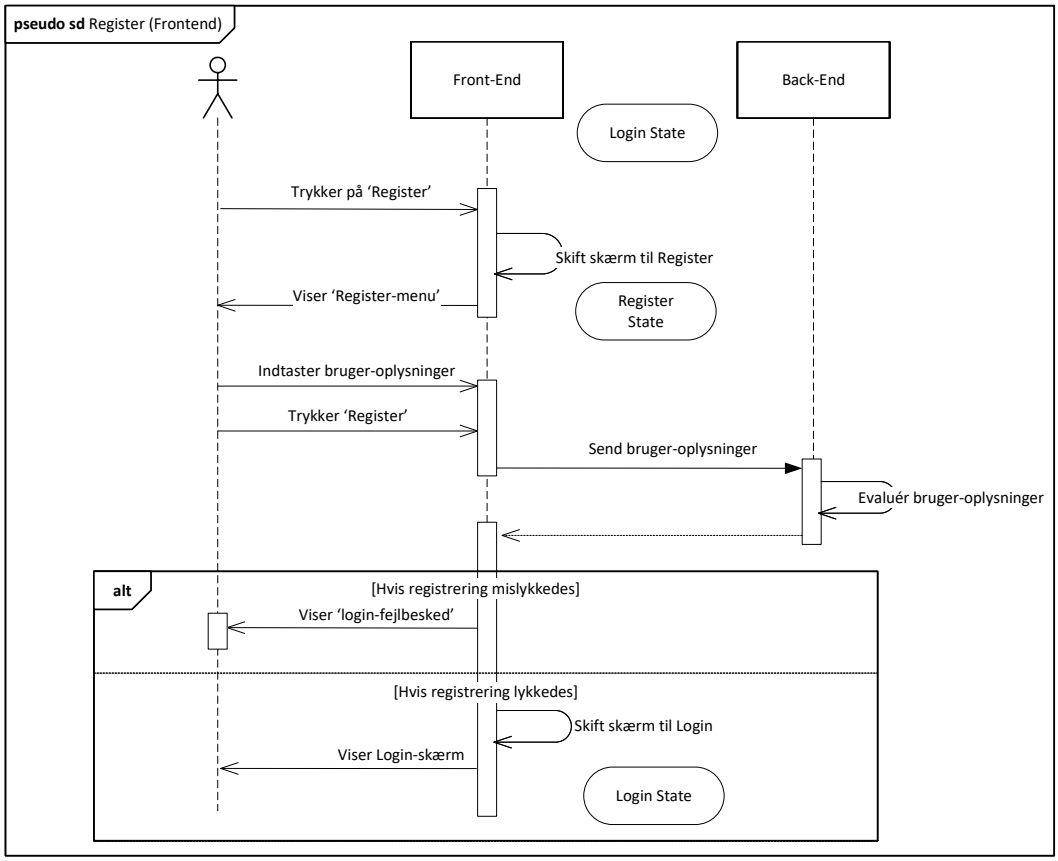

Figure 8: Pseudo sekvensdiagram af forløbet af userstory "Register", set fra Frontends perspektiv. Med håndtering af forkerte login oplysninger.

2.3 Backend Arkitektur

I dette afsnit redegøres for arkitekturen af backend containeren. Dokumentet vil først komme med en kort redegørelse for MVC mønstret opbygning og dets bidrag til arkitekturen. Dernæst gennemgås de relevante krav hvad angår backenden, disse krav bruges som input til arkitekturen. Med kravene på plads præsenteres et C3 diagram over backend'en, som viser hvordan Web api'et opbygges. Til slut præsenteres REST principper samt hvordan de anvendes. Systemets backend vil bestå af et Web API udviklet i frameworket ASP.NET Core. Web API'et skal give mulighed for containerne Frontend og GameEngine at tilgå databasen igennem HTTP request/responses, for at hente og sende den nødvendige data til databasen. Hertil skal backend'en sørger for Authentication og Authorization af de indkommende kald.

2.3.1 MVC

Selve Web API'ets arkitektur bygges op omkring MVC (Model-View-Controller) mønstret. En illustration af MVC mønstrets struktur kan ses på figur x (mangler ref til kilde).
Som det kan ses på figuren består mønstret overordnet af 4 dele client, controller, model og et DAL (Data Access Layer). Da der er tale om et Web API gøres der ikke brug af view moduller. De enkelte

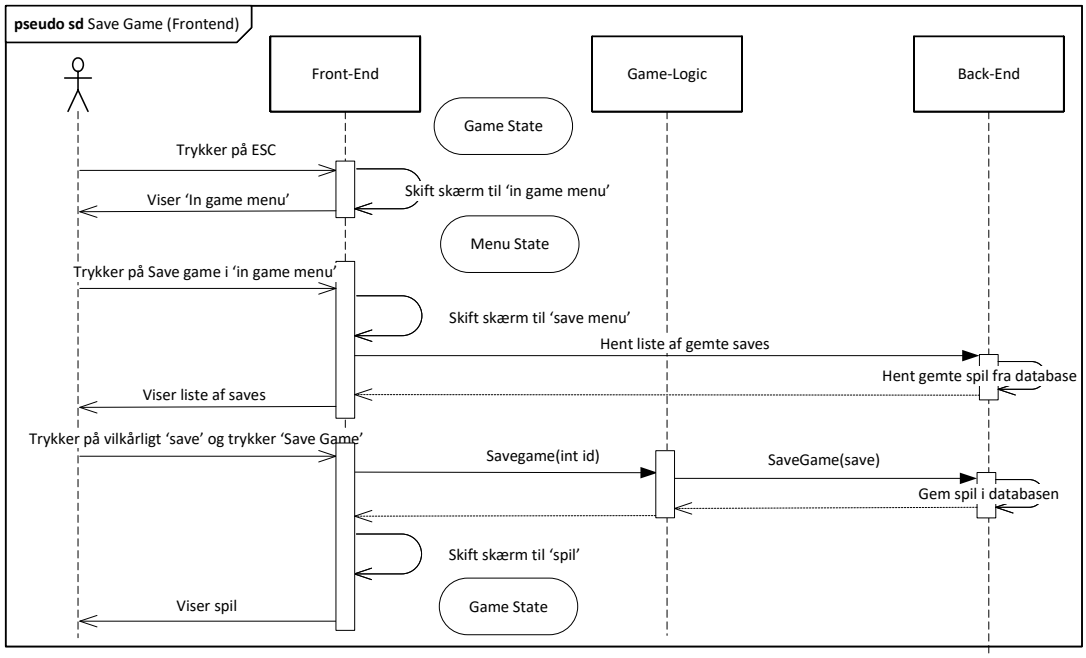

Figure 9: Pseudo sekvensdiagram af forløbet af userstory "Save Game", set fra Frontends perspektiv. Der laves 2 kald til databasen igennem Backenden, hvori der i det første kald, "Hent liste af gemte saves" hentes en liste af brugerens gemte spil og i andet kald gemmes brugerens nuværende spil henover det valgte spil.

moduler har følgende ansvar.

2.3.2 Client:

Clientens rolle er at kontakte Web API'et med henblik på at få udført en service, eksempel at logge ind.

2.3.3 Controller:

Controllernes ansvar er at håndterer indkommende kald fra clienten, og route til de rigtige Http endpoints, hvorefter databasen kontaktes igennem DAL.

2.3.4 DAL:

DAL's ansvar er at kontakte og administere kommuikation med databasen i form af queries. Det er også her relationerne imellem data objekterne defineres.

2.3.5 Model:

Modelernes ansvar er at håndtere og definere den data som kan sendes frem og tilbage mellem clienten og databasen, de sørger altså for at binde alle modulerne sammen, således at der er enighed omkring hvordan data objekterne ser ud. Som det kan ses passer MVC mønstret perfekt til den funktionalitet der ønskes. Mønstret gør det muligt at lave en logisk gruppering af relaterede opgaver, og er med til

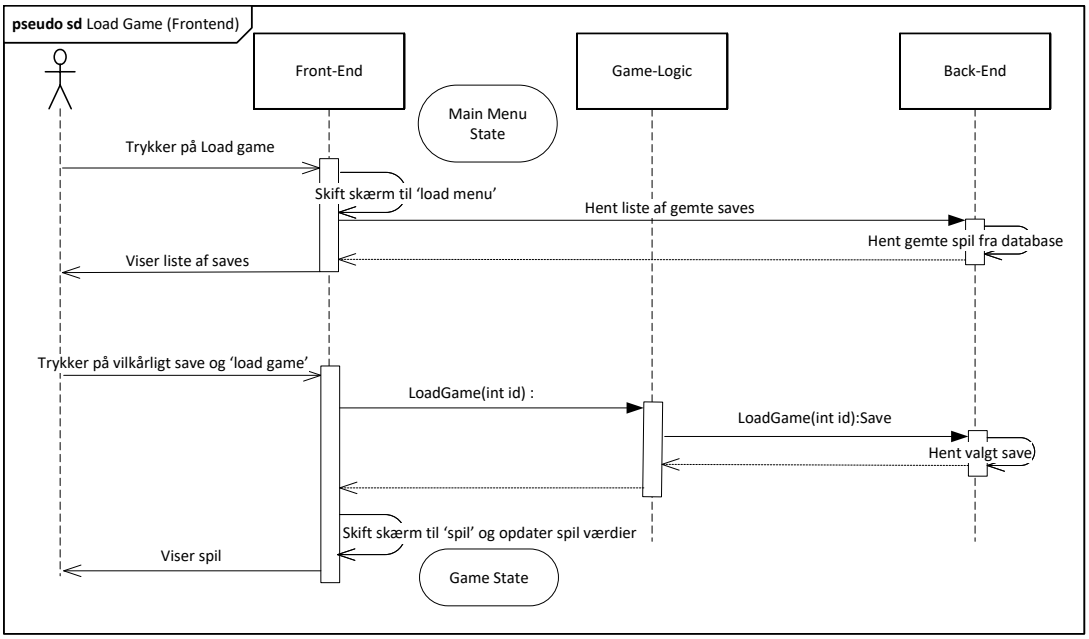

Figure 10: Pseudo sekvensdiagram af forløbet af userstory "Load Game", set fra Frontends perspektiv. Der laves 2 kald til databasen igennem Backenden, hvori der i det første kald, "Hent liste af gemte saves" hentes en liste af brugerens gemte spil og i andet kald hentes brugerens valgte spil og spillet startes op.

at sikre høj samhørighed i applikationen, ved at de enkelte ansvarsområder fordeles ud på hver sine moduler. På figur 1 ses det tydeligt hvordan koblingen mellem de enkelte moduler holdes på et lavt niveau, hvilket gør det lettere at bygge videre på og tilføje ny funktionalitet.

2.3.6 C3-Model for backend

De relevante user stories for backend containeren udvælges. Disse omhandler håndtering af bruger konti, samt loading og saving af et game state.

- User Story 1 : Log in
- User Story 2 : Opret Bruger
- UserStory 7 : Exit Menu -¿ Save and Exit
- UserStory 17 : Main Menu -¿ Load List Game
- UserStory 18 : Main Menu -¿ Load Game -¿ Load
- UserStory 19 : Main Menu -¿ Load Game -¿ Delete Game

Af de ikke funktionelle krav er følgende krav relevante:

- Skal kunne gemme maksimalt 5 save games
- Skal kunne loade et spil indenfor maksimalt 5 s.

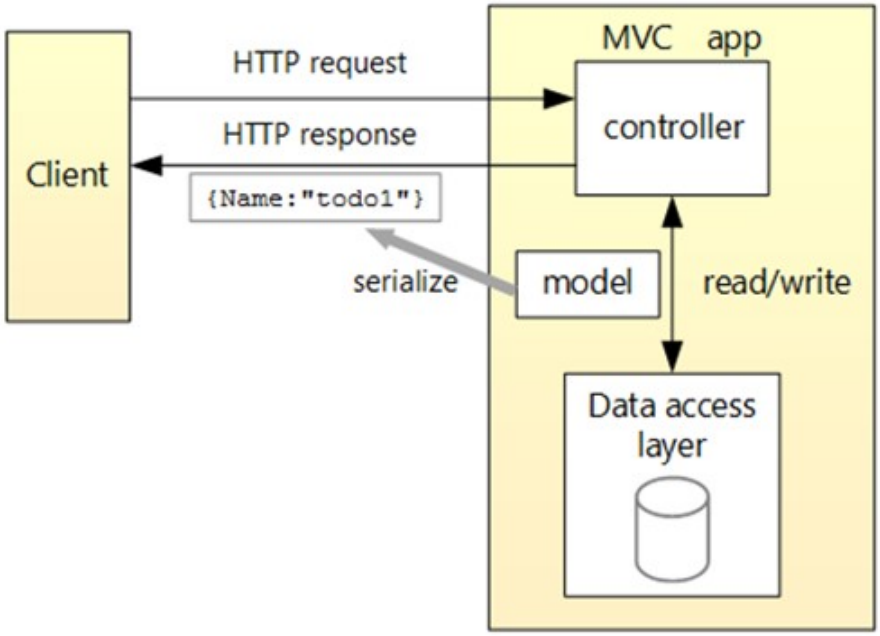

Figure 11: Illustration af MVC pattern, bestående af clienter, controller, modeller, og et DAL. som viser hvordan de kommunikerer

- Skal kunne respondere indenfor maksimalt 5 s.

De ikke funktionelle krav tages håndom under designet og implementeringen af backenden. For ydderligere detaljer omkring funktionelle og ikke funktionelle krav henvises til Bilag x
Der bygges videre på C4 modellen for systemet , et level 3 component diagram over backend containeren udarbejdes, som viser hvilke componenter containeren indeholder, deres ansvarsområder, hvordan de kommunikere indbyrdes og med de omkring liggende containere, samt hvilke teknologier der anvendes i implementeringen, diagrammet kan ses på figur X.
Figuren illustrerer en lagdelt struktur, hvor der ses en client som kalder ned i controllerne Authentication og Save Controller, som kalder videre til DAL, som så tilgår databasen denne kommunikationsvej går begge veje.

2.3.7 Client

Der eksisterer en client, som kommunikerer med backenden Game Module. Game Module's kald til backenden kan overordnet set indeles i to undergrupper, bruger håndtering og save håndtering. Bruger håndtering indeholder kald som omhandler login og registreing (User story 1 og 2), hvor save håndtering omhandler loading og saving af Game state (User story 7, 17, 18 og 19). Kommuniaktionen her vil foregå ved netværkskald med protokolen HTTPS. Clienten udfører request/response kald til backenden igennem specifikke URL's. For at data objekterne kan sendes frem og tilbage med HTTPS kald skal disse seralizeres og deseralizeres mellem JSON-objekter og .Net objekter.

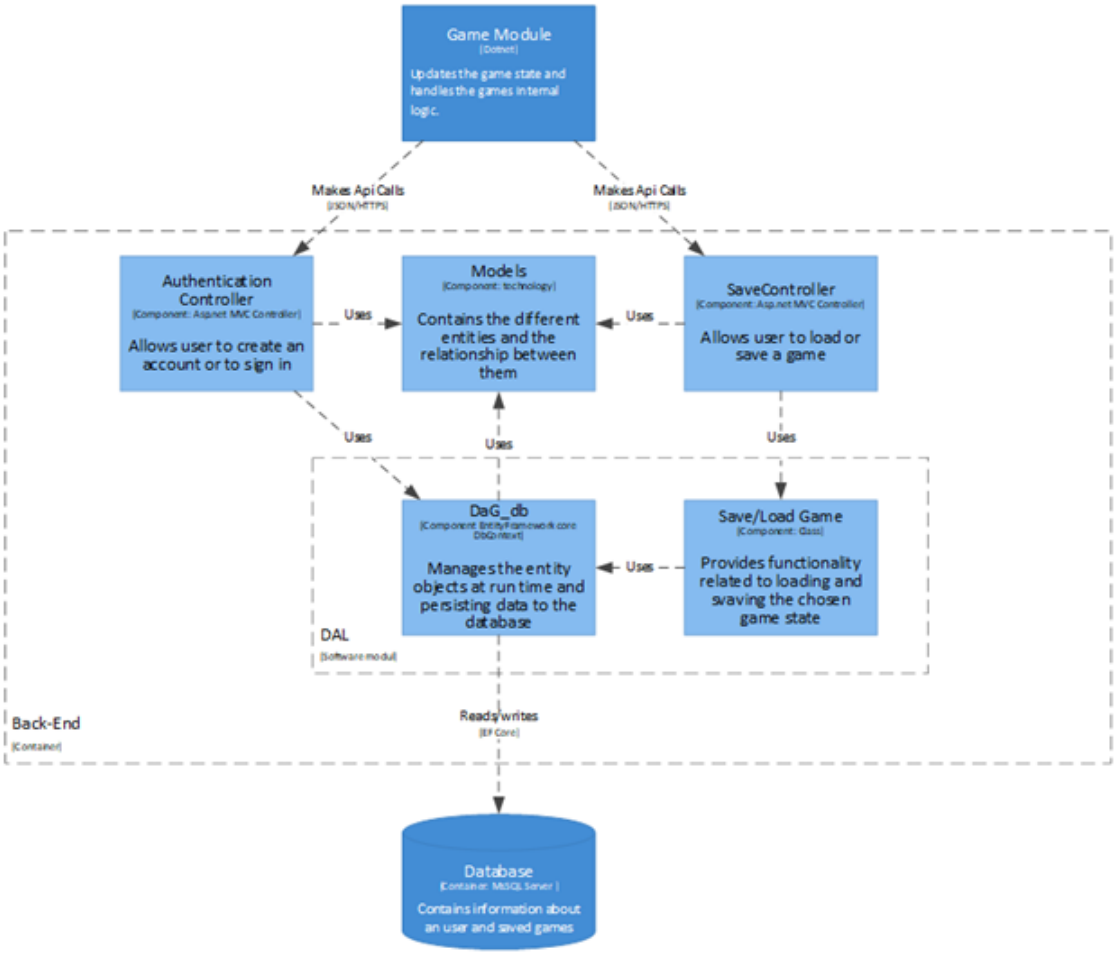

Figure 12: C3-Model over Backend container,figuren giver et overblik over de componenter backenden består, samt hvordan disse kommunikere indbyrdes og med omkring liggende containere.

2.3.8 Controller

Inde i backenden anvendes ASP. Net MVC controllere(Mangler ref) til at modtage og svare på kald fra clienten. Der laves en controller for Bruger håndtering og en controller for Save håndtering. Hver request mappes til en specifik controller. Controllerne vil så indeholde action metoder, som udføres når den modtagere et kald. Når controlleren har udført sin action retuneres et Action Result, hvor i et data objekt kan wrappes eller den kan indeholde en fejlmeddelse hvis noget går galt.

2.3.9 DAL

De to controllere kalder som sagt videre ned i DAL laget, dette lag består af en database kontekst DaG_db og en hjælper klasse til at udfører Queries. Her gøres brug af EF (Entity Framework) Core, som gør det muligt at arbejde med DbContext klassen, denne bruges til skabe en kontekst af databasen hvor i de enkelte modeller kan mappes til entities med Db¡set¿. Dette giver mulighed for at udfører operationer/queries på SQL databasen ved hjælp af LINQ (Language-Integrated Query).

2.3.10 Model

Modellerne fungerer som bindeledet mellem alle komponenterne, de definere den data som der arbejdes på, og vil blot bestå af en række C# klasser.

Som det kan ses passer MVC mønstret perfekt til den funktionalitet der ønskes. Mønstret gør det muligt at lave en logisk gruppering af relaterede opgaver, og er med til at sikre høj samhørighed i applikationen, ved at de enkelte ansvarsområder fordeles ud på hver sine moduler. På figur 1 ses det tydeligt hvordan koblingen mellem de enkelte moduler holdes på et lavt niveau, hvilket gør det lettere at bygge videre på og tilføje ny funktionalitet.

2.3.11 REST

Web Api'et arkitektur vil gøre brug af REST (Representaional State Transfer) principper, for ikke at gøre data overførelserne for komplekse og for at overholde SoC (Seperation of Concerns), ved at adskille repræsentationen og dataen fra hinanden, på den måde sikres det at dataen, kan blive repræsenteret på den ønskede måde, og ikke er tvunget til at blive gemt på en specifikt måde. Dette kommer til udtryk ved følgende 5 REST principper(Mangler ref) som søges opnået:

1. Alle de data objekter der arbejdes med tilhører en bestemt unique URI. Dataen hentes, sendes og manipuleres igennem standard HTTP metoder som (GET, POST, PUT, DELETE).
2. Der arbejdes ud fra en client/server arkitektur med data som resource.
3. Web Api'et er stateless, hvilket vil sige der gemmes ikke nogen tilstand omkring clienten på server siden.
4. Der arbejdes ud fra en lagdelt struktur, som betyder at hver component kun kan se de componenter som grænser op til den selv.
5. Der anvendes Cashing på server siden (Dette vil ikke blive implementeret).

2.3.12 Konlusion

Systemets backend består at Web Api, som bygges op omkring MVC mønstret, som gør det muligt at lave en logsik gruppering af den enkelte funktionalitet. Web Api'et vil ligeledes gøre brug af REST principper for at adskille data resourcerne fra deres repræsentation på clientens side.

3 Design

3.1 Overordnet System Design

Dette afsnit repræsenterer hvordan modulerne ønskes at kommunikere med hinanden samt give et overblik over hvor meget der kommunikeres mellem modulerne i de forskellige stadier. Der vil i dette afsnit indgå 4 User stories til at repræsentere systemets design. Disse User Stories er relevante for design, da de involverer alle moduler samtidigt og giver et indblik i kommunikationen på tværs af systemet.
Herunder ses et sekvensdiagram for User Story 7-10 - Load. Her ses hvordan det ønskes at de forskellige moduler kommunikerer på tværs af systemet når bruger loade et save fra databasen.

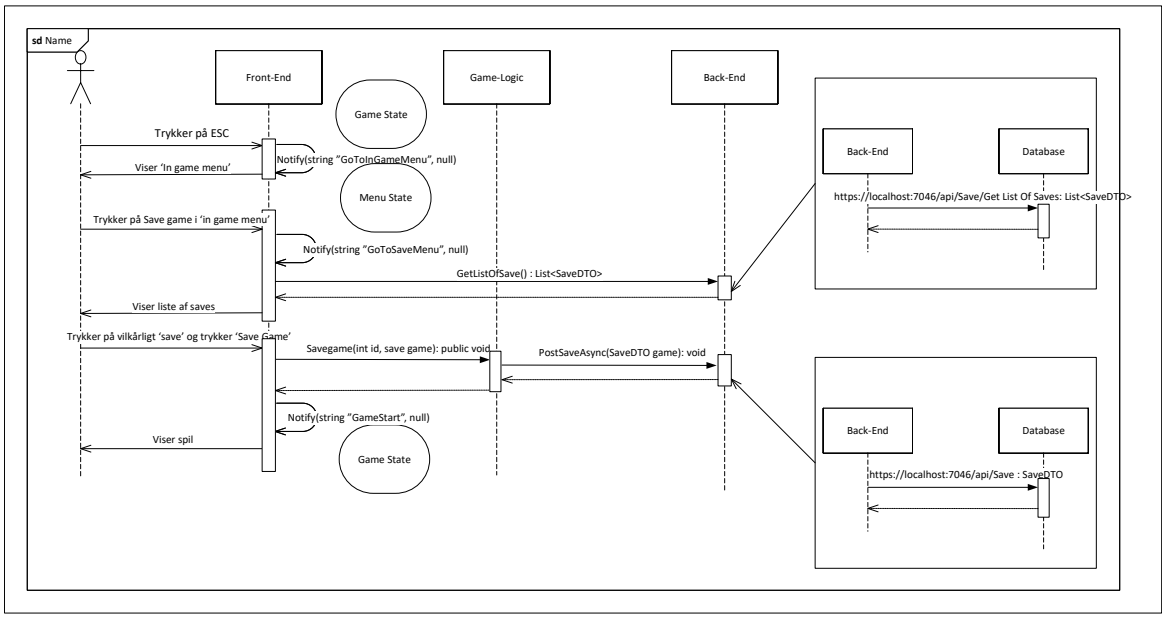

Figure 13: SD diagram for User Story 7-10. Diagrammet viser hvordan det ønskes at systemet overordnet skal kommunikere på tværs af hindanden når bruger skal gemme sit aktuelle save

Herunder ses et sekvensdiagram for User Story 17-18 - Load. Her ses hvordan det ønskes at de forskellige moduler kommunikerer på tværs af systemet når bruger gemme et save i databasen.
Herunder ses et sekvensdiagram for User Story 1 - Log in. Her ses hvordan det ønskes at de forskellige moduler kommunikerer på tværs af systemet når bruger skal logge ind.
Herunder ses et sekvensdiagram for User Story 2 - Opret Bruger. Her ses hvordan det ønskes at de forskellige moduler kommunikerer på tværs af systemet når bruger skal oprette en bruger i systemet.

3.2 Frontend Design

I frontenden ønskes det at hele spillet foregår i et vindue. Det er derfor nødvendigt at programmet kan skifte mellem forskellige views uden at skulle åbne et nyt vindue, hver gang der skiftes. Løsningen til denne udfordring beskrives nærmere i implementerings afsnittet, nemlig subsection 4.2. Spillet vil bestå af en række af vinduer, som giver spilleren den nødvændige information for at de kan spille spillet (så som at logge ind, gemme og hente spil, samt spille spillet).
Inden arbejdet på Frontend arkitekturen begyndte, er der blevet lavet en teknologiundersøgelse **(INDSÆT REFERENCE HER)**, om hvilket udviklingsværktøh Frontenden og derigennem spillet

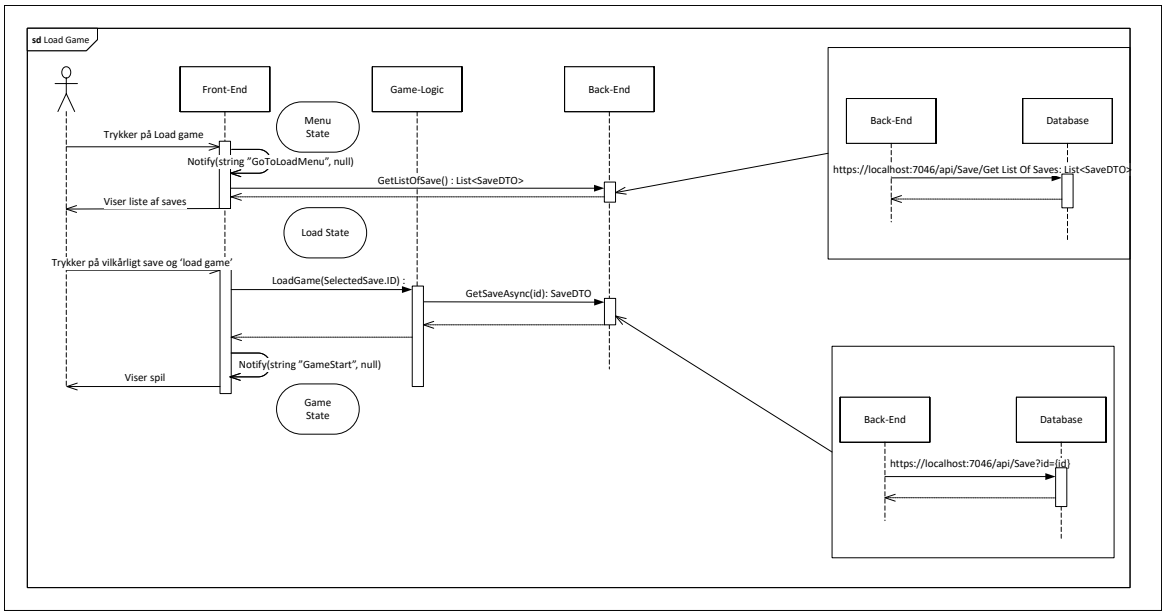

Figure 14: SD diagram for User Story 17-18. Diagrammet viser hvordan systemet overordnet skal kommunikere på tværs når bruger skal loade sit aktuelle save.

skulle udvikles i. Baseret på denne teknologiundersøgelse er der blevet valgt, at spillet vil blive udviklet i et .NET framework. Dette valg er blandt andet truffet da dette framework passer bedre med den opdeling af arbejde der er lavet i projektgruppen, altså opdelingen af Frontend og Backend. For andre grunde, se **(INDSÆT REFERENCE HER)**, hvor flere fordele og ulemper for både unity og .NET frameworket er sat op.
Her følger en række af mockups[1] af nogle af spillets views.

3.2.1 Room

Room view (Figure 17) er det primære spil-vindue. Her præsenteres spilleren for en beskrivelse af det rum de er i, samt hvilke elementer i rummet de kan interagere med. Der vises også et kort over banen. Kortet Viser kun de rum spilleren allerede har været i, mens resten holdes skjult. Når brugeren så besøger et nyt rum, kan dette ses selvom spilleren forlader rummet. Dette lader spilleren udforske og oplåse hele kortet.
En række knapper nederst i højre hjørne på skærmen giver spilleren mulighed for at interagere med spillet. Fire knapper ("Go North/West/South/East") lader spilleren gå fra et rum til et andet. Ikke alle rum har forbingelse til alle sider, så det er f.eks. ikke altid muligt at trykke på "Go North". Kortet og rum beskrivelsen fortæller hvilken vej det er muligt at bevæge sig i.
Udover de fire retningsknapper er der et antal andre knapper. Disse bruges til at gemme spillet, gå til menuer, samt interagere med elementerne i rummet. Det specifikke antal og deres funktion er afhængig af den præcise implementering.

3.2.2 Combat

Combat vinduet er baseret på room vinduet. Strukturen er den samme: der er et kort øverst til højre, en beskrivelse øverst til venstre og knapper neders til højre. Nederst til venstre er der information om hvordan kampen går, i stedet for en liste af elementer i rummet.

[1] Lavet i Unity

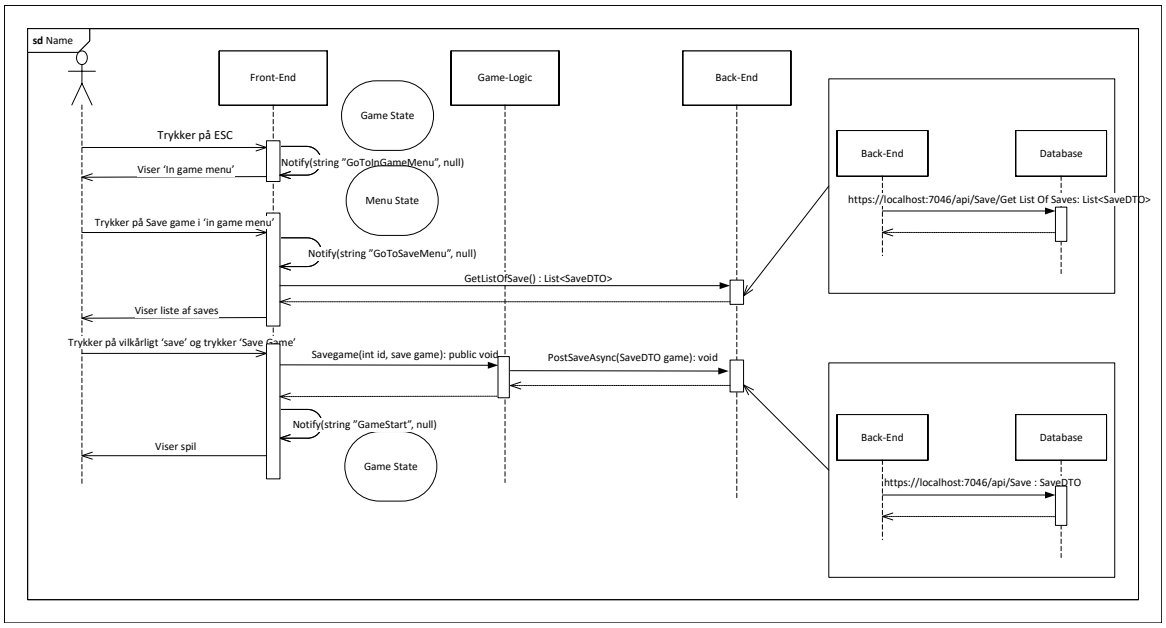

Figure 15: SD diagram for User Story 1. Diagrammet viser hvordan systemet overordnet skal kommunikere på tværs når bruger skal logge ind

Knapperne består af nogle "menu" knapper, som lader dig gå til spil menuer, samt en 'Fight' knap og en 'Flee' knap. 'Fight' knappen lader spilleren kæmpe mod fjenden, mens 'Flee' knappen lader spilleren flygte fra kampen og tilbage til rummet som spilleren kom fra.

3.2.3 Login

Når spillet startes bedes spilleren prompte at logge ind på deres profil. Dette sker i login vinduet. Spilleren kan indtaste sit brugernavn og kodeord i de to tekstfelter 'Username' og 'Password'. Knappen login fører dem videre til spillet, hvis det indtastede login er korrekt.
Knappen create user fører spilleren til et vindue som ligner login vinduet, og som lader spilleren oprætte en ny bruger.
Exit lukker spillet.

3.2.4 Settings

Settings viduet tillader at spilleren kan ændre indstillinger for spillet, og kan tilgås fra hovedmenuen, samt fra selve spillet.
Det er her muligt at ændre f.eks. skærmopløsning og lydstyrke. Det er muligt at gemme de indstillinger som er valgt, forlade menuen uden at gemme de valgte indstillinger, samt gendanne standard indstillingerne for spillet.

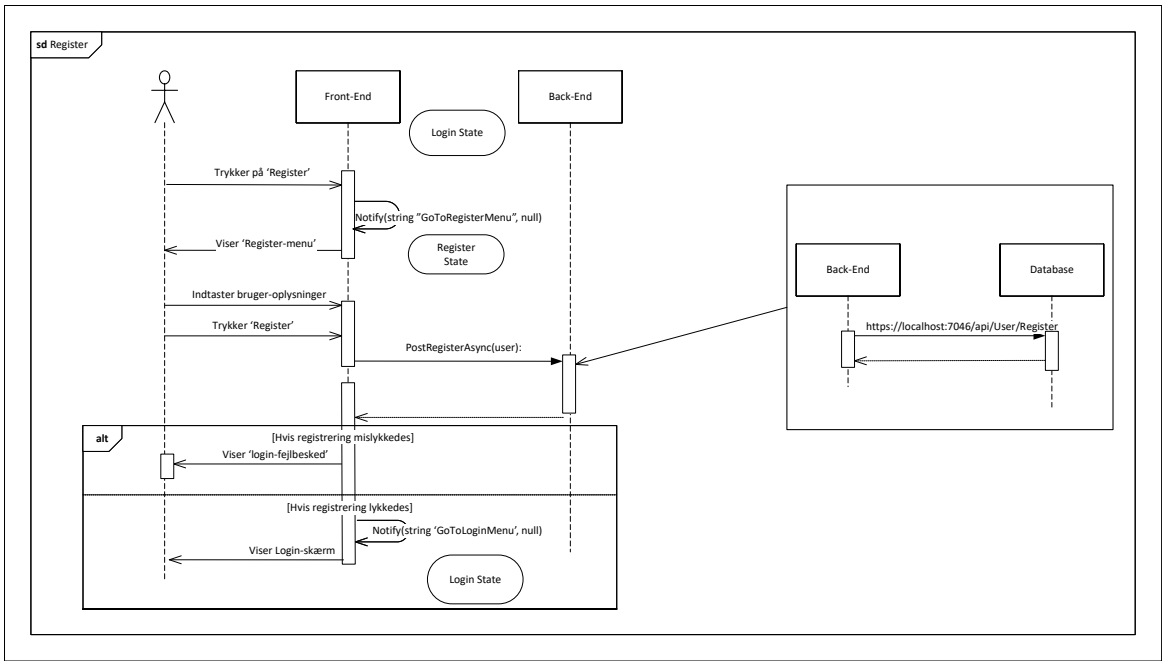

Figure 16: SD diagram for User Story 2. Diagrammet viser hvordan systemet overordnet skal kommunikere på tværs når bruger skal oprette en bruger i systemet

3.3 Database Design

For at kunne udarbejde et ER diagram til modellering af vores sql database skal vi start med at finde ud af hvilke krav vi har til og hvilke attributter vi ønsker at gemme i vores database. Først og fremmest ønskede grupper at vi kunne gemme beskrivelserne af de forskellige rum, i spillets layout, for at formindske antallet af filer i klienten, og samtidigt gøre eventuelle senere tilføjelser nemmere. Her benyttes rummets id som key, da vi ikke ønsker at man skal kunne oprette flere beskrivelser til samme rum.
Her efter kommer kravene til at kunne gemme et spil for en bruger. Her ønskede vi at man kunne stå et vilkårligt sted i spillet, med untagelse af en combat, og gemme spillet. Det skulle derefter være muligt for spilleren at loade spillet igen, hvorefter spillet er i samme stadie som man gemte det i.
Først og fremmest ønskede gruppen et bruger system, så eventuelle gemte spil kun tilhørte en bruger. Der gemmes derfor en bruger entitet med et unikt brugernavn og et tilhørende password. Sikkerhed på password og versalfølsomhed på brugernavnet håndteres spillets backend. En spiller skal derefter kunne gemme unikke 5 spil med forskellige oplysninger. Håndteringen af restriktionen med max 5 forskellige spil pr. Bruger, håndteres ved at oprette 5 standard gemte spil pr. bruger som vi overskriver, når vi gemmer. Der kan på denne måde ikke oprettes mere en 5 gemte spil pr. bruger.
I et gemt spil ønkede vi at gemme en række forskellige attributter for spilleren. Første og fremmest får hver spil et unikt id som vi benytter til at lave forhold mellem de forskellige tabeller. Et gemt spil får et navn, valgt af brugeren, som gør det nemmere for brugeren at differentiere mellem de forskellige spil. Dette navn skal forskelligt fra de 4 andre gemte spil som tilhører brugeren. En spiller Health gemmes også, da man kan have taget skade efter en kamp. Det gemmes også hvilket rum, spilleren står i når spillet gemmes, så vi loader korrekt tilbage. En spiller kan derudover også holde genstande, som armor og våben, i hånden eller i sit inventory.
Tabellen med inventory har 2 atributter, et ID, som svarer til en bestemt genstand, og en reference til set SaveID. Denne parring er unik, da man ikke kan holde 2 af den samme genstand. Tabellerne

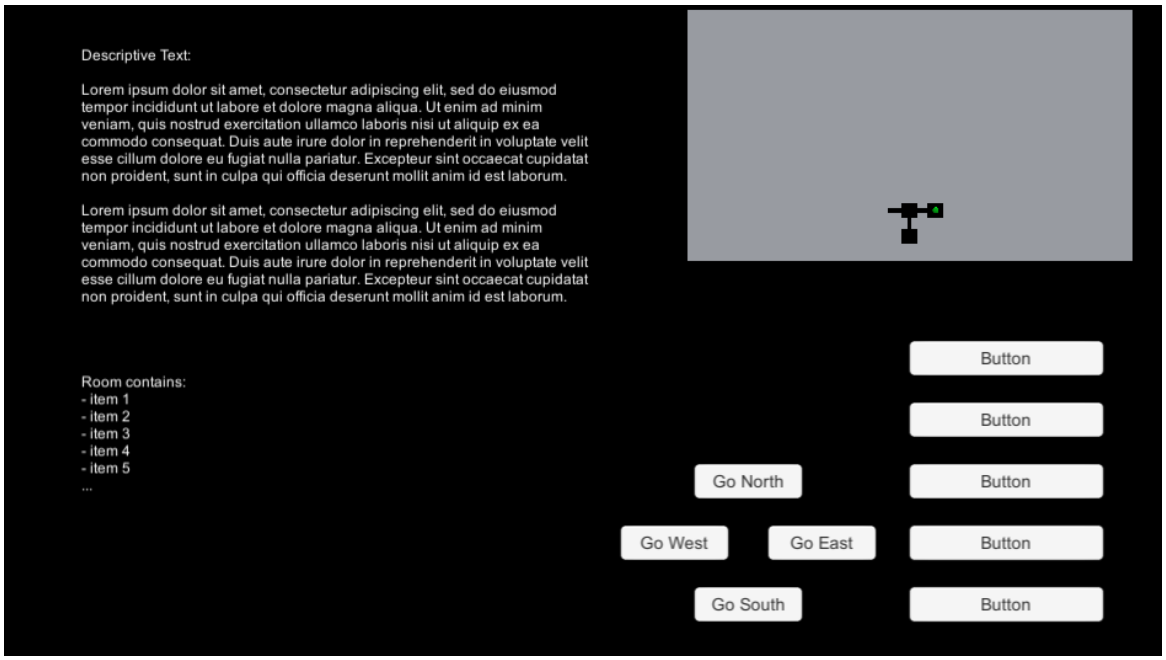

Figure 17: Et mockup af det primære spil vindue. Tekst øverst i venstre side af skærmen giver en beskrivelse af det rum spilleren er i, samt en liste af elementer i rummet som spilleren kan interagere med. Øverst til højre vises et billede af banen. Spilleren interagerer med spillet via knapper nederst i højre hjørne. Knapperne "Go North/West/South/East" fører spilleren ind i et andet rum, mens de resterende knapper (markeret "Button") bruges til andre funktionaliterer i spillet.

med Enemies og puzzles fungerer på samme måde. Her har hver enemy og puzzle i spillet et unikt id. Id'et gemmes i kombination med et saveId, som et unikt par, da man ikke kan vinde over samme enimy og løse samme puzzle flere gange. Til slut ønskede vi at kunne vise spilleren de rum som allerede er blevet besøgt. Derfor gemmes der i path tabellen, for hvert spil, en unik kombination af saveID og besøgt rum id. Denne parring er unik da man blot behøver at besøge et rum før det er synligt på kortet.
Der er i projektet oprettet klasser tilsvarende ER diagrammerne.

4 Implementering

4.1 System Implementering

Under implementeringen af nogle funktioner i samtlige moduler, er funktioner blevet lavet til funktioner der følger følgende opstilling:
public async Task <T> Foobar();
Funktionerne returnerer en Task af typen T og den kan gøres asynkront. Dette er specielt vigtigt når man kontakter databasen, da programmet ikke må køre videre før den asynkrone funktion er færdig med sit database kald. Når en funktion i f.eks. Game Controlleren kalder: public async Task <SaveDTO> GetSaveAsync(int id) i backend controlleren, i sin egen funktion SaveGame() skal SaveGame() også erklæres async og returnerer en Task af typen T og derfor er nærmest alle funktioner der kontakter databasen erklæret for async kald, som returnerer typen Task.

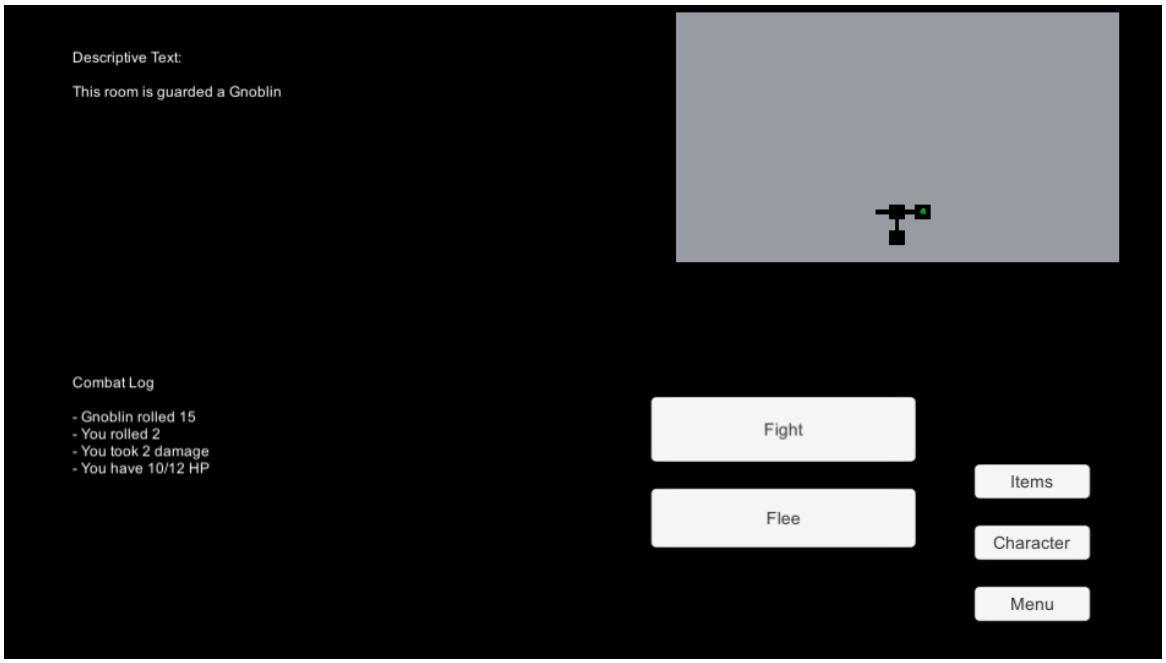

Figure 18: Combat view. Spilleren præsenteres for en fjende, og får information om hvordan kampen mod fjenden går. Der er knapper til at kæmpe og flygte, samt gå til menuer. øverst til højre er kortet over banen, ligesom i Room vinduet Figure 17.

4.2 Frontend Implementering

Det endelige design af vinderne følger tæt det oprindelige mockup design. Der benyttes MVVM (Model, View and View Model) designpatterns Det har undervejs i implementeringen vist sig et behov for væsentlig flere menuer end først antaget, og disse er implementeret efter samme overordnede design, som resten af systemet, så derved følges stilen og følelsen af spillet.
Det endelige design har følgende views: Login, Register, Main, Settings, Ingame, Load, Save, Inventory, Character, Victory, Defeat, Room og Combat.
Til at skifte views uden at oprette et nyt vindue bruges et mediator design, hvor hver knap som skifter view giver en besked til mediatoren. Dette virker fordi alle views er oprettet som WPF user controlls, og ikke views, hvilket tillader at et main view kan skifte mellem viewmodels, derved skifter vil alt indhold på vinduet, men uden at selve rammen skiftet. Dette resulterer i et mere flydende User Interface for brugeren og derved en alt i alt bedre oplevelse.[2]
Et eksempel på hvordan mediatoren kaldes ved et tryk på en knap kan ses i følgende kode eksempel:

```
private DelegateCommand _loadGame;

public DelegateCommand LoadGame => _loadGame ?? (
        _loadGame = new DelegateCommand(
        ExecuteLoadCommand, CanExecuteLoadCommand));

async void ExecuteLoadCommand()
{
    await GameController.Instance.LoadGame(SelectedSave.ID);
    Mediator.Notify("GameStart", "");
```

[2] **https://www.technical-recipes.com/2018/navigating-between-views-in-wpf-mvvm/**.

Figure 19: Login view. Bruger kan indtaste sit brugernavn og kodeord for at få adgang til spillet, eller trykke på create user for at komme til et vindue hvor man kan oprætte en ny bruger. Det er også muligt at forlade spillet igen.

```
}

bool CanExecuteLoadCommand()
{
    return SelectedSave != null;
}

void LoadCommand()
{
    LoadGame.RaiseCanExecuteChanged();
}
```

Der er implementeret at nogle af spillets funktioner kan tilgås via key-bindings, hvilket har nødsaget et brud på MVVM-designet. Når mediatoren skifter mellem views bilver fokus ikke sat til indholdet af det nye view, og keybindings virker derfor ikke. For at løse dette sættes top-elementet på den nye side som fokus. Dette kan dog ikke gøres i MVVM, så her er det pattern brudt, da det er nødvændigt at gå ind i code-behind filen for at sætte fokus.
De følgende afsnit viser et udvalg af view, samt en beskrivelse af hvordan de afviger fra det oprindelige design og en beskrivelse af interesante programerings-tekniske beslutninger.

4.2.1 Login view

De overordnede struktur af login (Figure 23) og register menuerne følger meget tæt det oprindelige design. Alle elementer er blevet stiliseret så de matcher det ønskede look. Dette er gjort ved brug af globale resourses og styles i app.xaml filen(**INDSÆT REFERENCE HER**). Dette gør det nemt at oprette nye views og ændre udseende af hele spillet. Selve login håndteres af backenden som kaldes

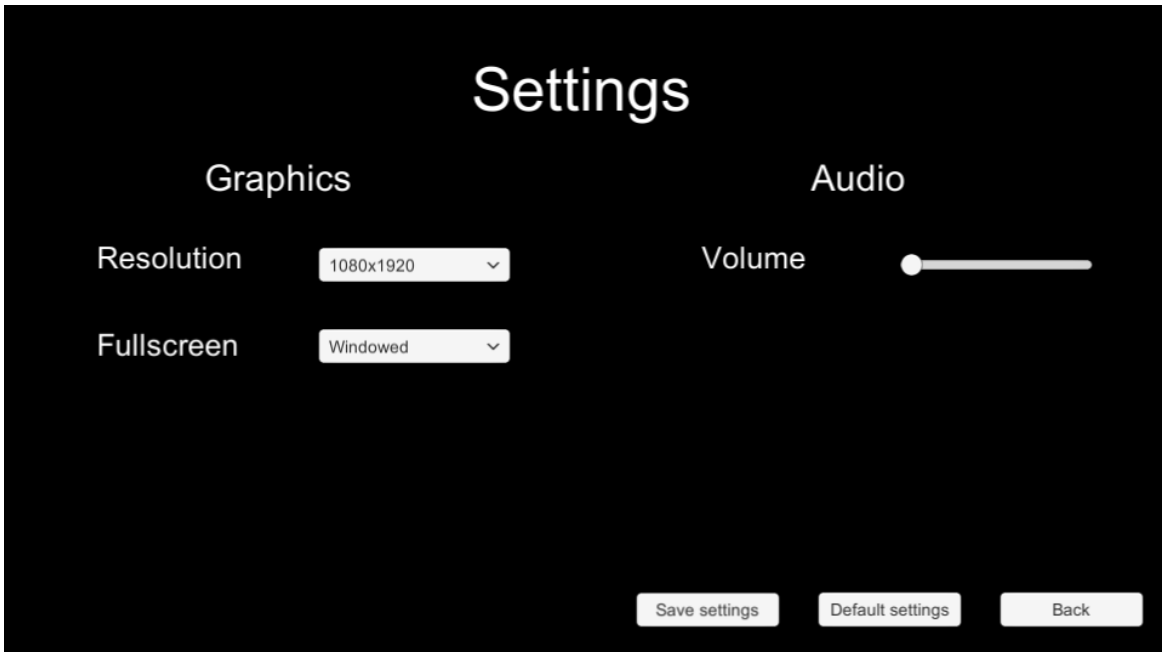

Figure 20: Settings view. Tillader at man kan ændre indstillinger for spillet, så som skærmopløsning og lydstyrke. Det er muligt at gemme indstillingerne, forlade skærmen uden at gemme indstillingerne, samt sætte spillet tilbage til standard indstilinger.

af login og register knapperne via command bindings.

4.2.2 Room View

Visuelt er room view (Figure 24) ikke ændret betydeligt fra det oprindelige design. Der er ændret lidt på placering og antal af knapper så det passer til antallet af interaktioner tilgængelig til brugeren. Kortet er lavet så det opdateres når spilleren går ind i et nyt rum, ved at ændre på synligheden af elementerne i kortet. Det er yderligere sat op så det kan skaleres til de skærmopløsninger som understøtes.
Ved at trykke på interact knappen kan spilleren flytte et valgt 'item' fra listennederst til venstre over i sit inventory (et seperat view), som kan tilgås veed at trykke på Inventory knappen.
Alt tekst er vist med data binding.

4.2.3 Combat View

Combat view (Figure 25) er bygget med room view som skabelon, så de fleste elementer er ens. Knappernes antal og funktion er ændret, og sammenlignet med det oprindelige design er knapperne for items og character fjernet, da det blev besluttet at det ikke skulle være muligt at tilgå sit inventory under en kamp. I stedet for en beskrivelse af rummet og en liste af items vises der nu en beskrivelse af hvordan kampen går nederst i venstre side af skærmen. Tal-værdierne hentes fra gameengine, og sættes ind i en tekststreng, som gør det nemt for brugeren at forstå hvordan det skal fortolkes. Der er yderligere tilføjet en healthbar, som giver en visuel indikation af, hvor tæt spilleren er på at tabe spillet og skifter frave fra grøn til gul til rød, som spilleren tager mere skade. Baren giver derfor lidt farve til et ellers meget gråtonet spil.

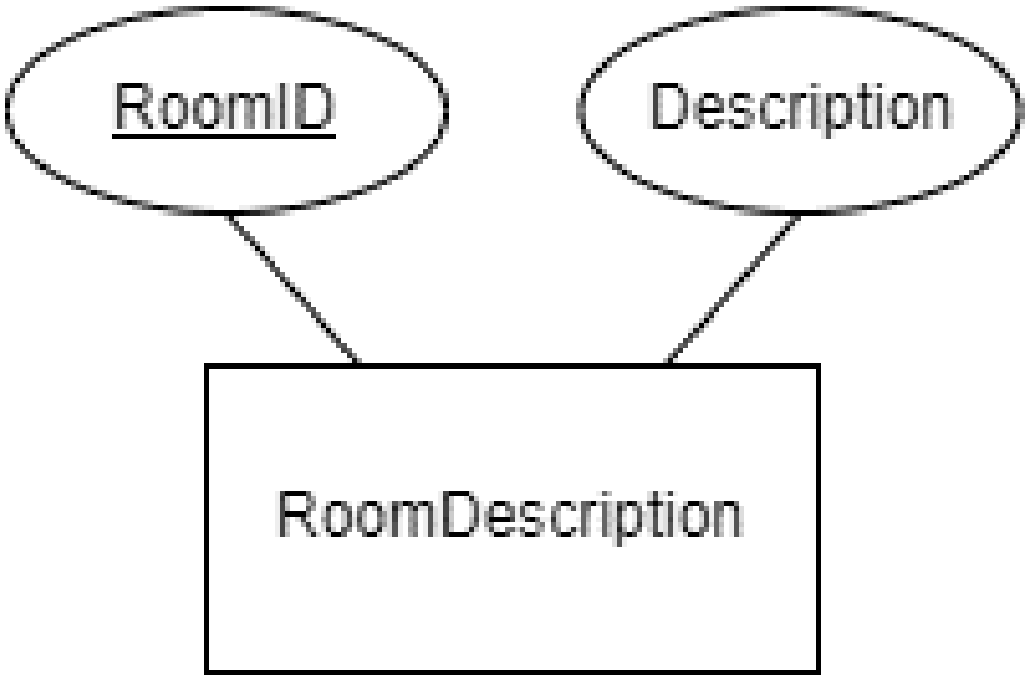

Figure 21: ER diagram for Roomdescription. En beskrivelse består blot af en beskrivende string samt det tilhørende unikke rumid.

4.2.4 Settings view

I settingsmenuen (Figure 26) er det muligt at vælge lydstyrke for musikken i spillet (samt slukke helt for musikken) og vælge mellem tre skærmopløsninger (1280x720, 1920x1080 og 2k). Det tre valgmuligheder er valgt til at teksten på skærmen stadig er læselige. Kortet i Room og Combat view skalerer med skærmopløsningen, men sætter en nedre grænse på 300x300 pixel. Tekststørelsen gør dog at den praktiske nedre grænse, hvor spillet ser ud som det skal, er 1280x720. Ved skærm opløsning større end 2k bliver teksten og kortet for småt til at kunne ses ordentligt, hvorfor 2k er en naturlig øvre grænse for skærmopløsningen. Alle opløsninger imellem de to grænser burde være i orden (så længe det bare nogenlunde følger en 4:3 eller 16:9 ratio).

For at sørge for at den valgte skærmopløsning er brugt i alle views er der oprettet et objekt til at holde informationer om blandt andet indstillinger, samt andre informationer, som det ønskes at kunne tilgå fra flere forskellige views uden at være nødsaget til at sende data med rundt, når der skiftes mellem views. Dette objekt følger et singleton design pattern, og den samme instance af objektet kan derfor tilgås fra alle de view som skal bruge informationer derfra, på den måde opnås det at der kan sættes glodbale indstillinger som kan bruges på alle views uden at informationen skal sendes med i et skærmskift.

Settings menuen kan tilgås fra både main menu og ingame menu, og det er derfor nødvendigt at spillet ved hvilken menu brugeren kommer fra, sådan at brugeren kan komme tilbage til den rigtige menu, når settings menuen forlades, ved at der trykkes på back-knappen. Til dette bruges singleton

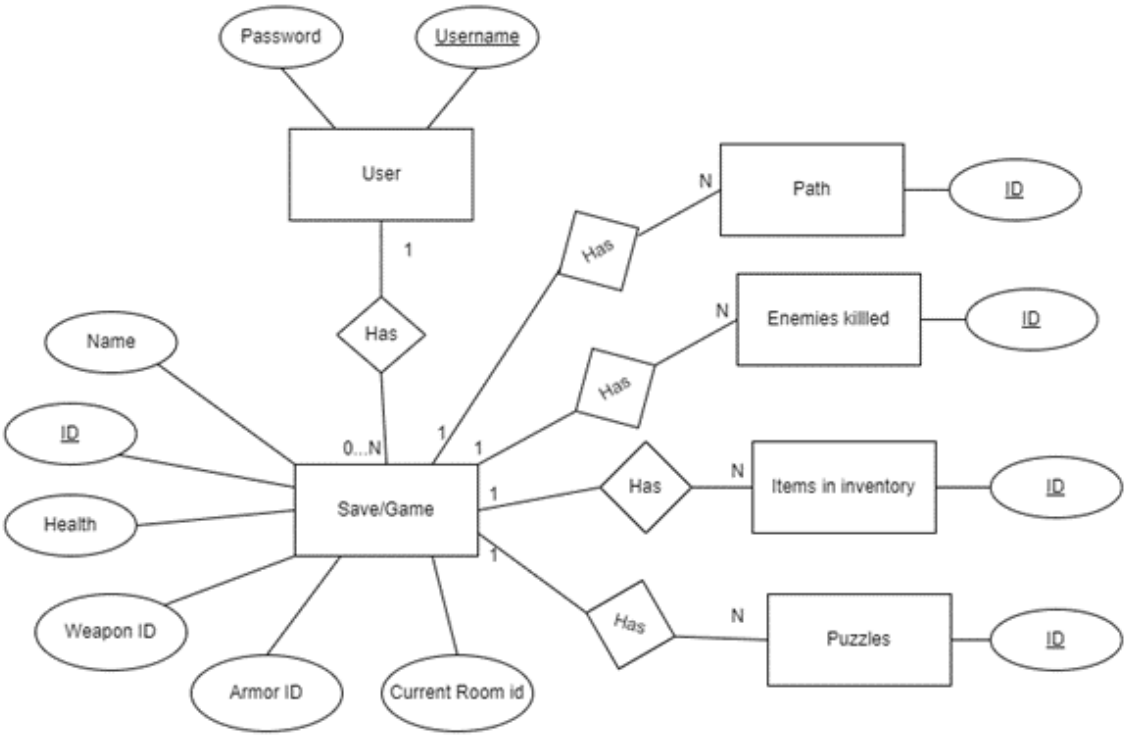

Figure 22: ER Diagram til at gemme et spil til en specifik bruger

objektet fra tidligere afsnit også, da det gør det nemt at bringe informationen mellem views. Dette bruges også i ingame menu, da denne kan tilgås fra både room view og combat view og skal kunne returnere brugeren til det rigtige view, når brugeren vælger at starte spillet igen.

4.2.5 Load view

Load (Figure 27) og Save menuerne præsenterer spilleren med en liste af gemte spil, som brugeren kan hente, eller gemme. Dette opnås ved at der hver gang spilleren åbner en af de to menuer, hentes en liste af tilgængelige 'save-games', som via databinding, præsenteres for brugeren. Når der trykkes på Save/Load sendes den fornødne kommando til backenden og backenden tager fat i databasen og udfører enten save eller load kommandoen.

4.2.6 Note om Baggrundsfarver

Da den generelle baggrundsfarve i spillet er sort er alle spillets interface elementer oprettet så baggrundsfarven matcher. Dette er for det meste nemt opnået ved brug af WPF styles, f.eks.:

```
<Style x:Key="textBoxStyle" TargetType="TextBox">
    <Setter Property="Background" Value="{StaticResource backgroundColor}"/>
    <Setter Property="Foreground" Value="{StaticResource textColor}"/>
    <Setter Property="FontSize" Value="25"/>
    <Setter Property="FontStyle" Value="Italic"/>
    <Setter Property="BorderThickness" Value="1"/>
</Style>
```

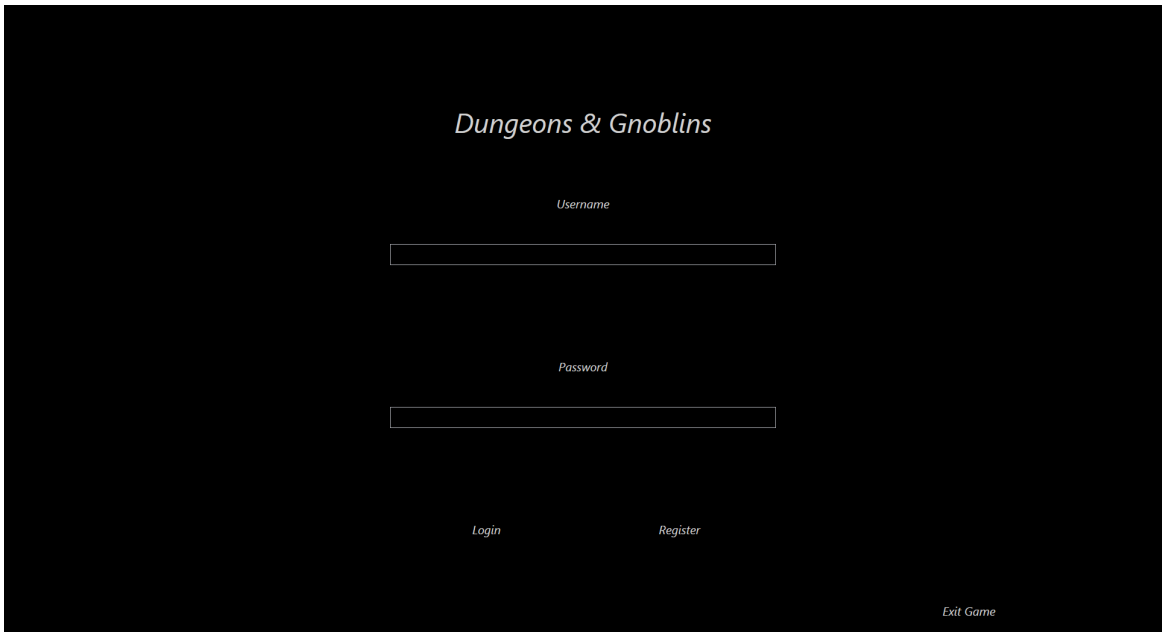

Figure 23: Endelig login skærm. Brugenavn og kodeord kan indtastes i de to felter. Register knappen fører til et nyt view, hvor man kan oprætte en bruger, mens login knappen fører spilleren til main menu, hvis deres login er korrekt.

men enkelte elementer (så som resolution dropdown menu, brugt i settings menuen) viste sig at være betydeligt mere omfattende. Det viser sig at det valgte element (WPF combobox) ikke tillader at baggrundsfarven for dropdown elementerne ændres i Windows 8 eller senere. Løsningen er at lave en kopi af hele templaten (ca. 300 linjer kode) for combobox og ændre 3-4 linjer.[3]

[3] **https://social.technet.microsoft.com/wiki/contents/articles/24240.changing-the-background-color-of-a-combobox-in-wp**

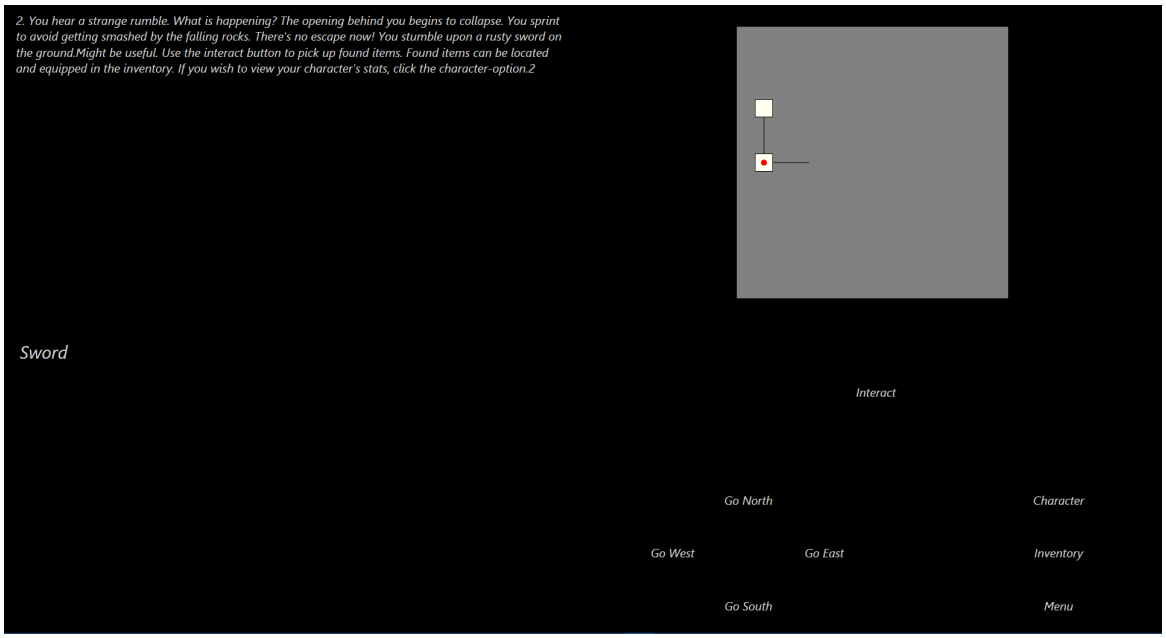

Figure 24: Endeligt udseende af room view. Generelt er der ikke ændret meget i forhold til det oprindelige design. Kortet er lavet så det skalerer med skærmopløsningen.

4.3 Game Engine

Game Enginen er skrevet i C#, et objectorienteret programmering sprog med stærkt library support der tillader udvikleren at fokusere på udvikling af applikationer istedet for udvikling af libraries til at støtte projektet.

Nedstående præstenteres en diagram over de mest kritiske komponenter Game Enginens interne spil logik og deres relationer til hinanden. Der er ikke medtaget interfaces, eller klasser som er map, items, BackendController og logs som er mere specifikke til spillet og ikke til den interne logik.

4.3.1 Gamecontroller Implementering

GameControlleren er det centrale komponent i game enginen, denne er ansvarlig for kommunikation til frontend del af applikationen. I denne implementering af GameControlleren har den adgang til alle spillet funktionaliteter gennem dennes association til Combatcontroller, se Figure 28, og BackendController , som ikke er vist på Figure 28.
Fra et implementering perspektiv er dette en nem løsning for en lille applikationen som denne men fra et design perspektiv er dette en dårlig løsning. GameControlleren har alt for mange grunde til at ændre sig og følger næppe SOLID principperne.

GameControlleren saver og loader spillet gennem sin relation til BackendController. Desværre er load funktionen ret kompliceret og burde være delt op i mindre functioner. Source code for load kan ses på Figure 29
Koden bliver kompliceret idet den forsøger at gennemløbe alle Room objects i spillet for på korrektvis at fjerne enemies og items, som allerede er blevet samlet op tidligere. Her gennemløbes alle Room objects og i hvert room gennemløbes alle items i Room chest objectet for at krydsreferer alle items med inventory listen for at se om de skal fjernes som en del af save gamet.

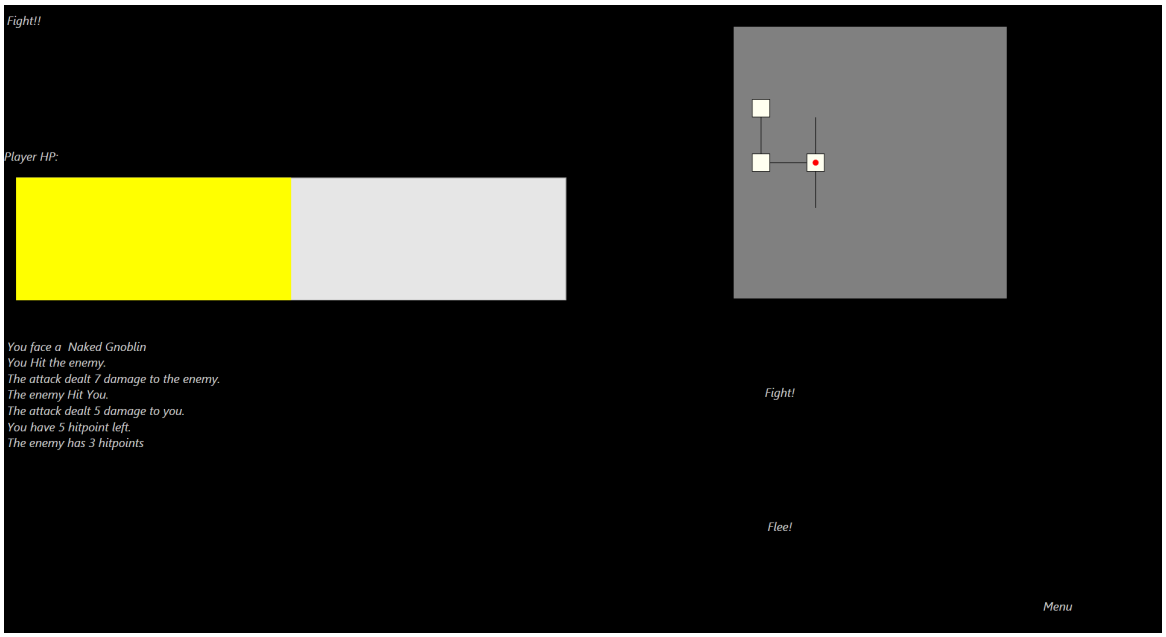

Figure 25: Combat view er baseret på room view. Her præsenters spilleren for info om en fjende og hvordan kampen mod fjenden går.

Ligeledes håndtere den enemies og krydsreferer alle enemies med slainEnemies listen for at se hvilke enemies spilleren allerede har besejret. Disse enemies fjerne herefter fra spillet.

4.3.2 Generering af Map

Map klassen generer et map til spillet. Denne består af en simple liste af lister over, hvilket rooms hvert room er forbundet til.
Denne Process er kompliceret og kræver en uddybende forklaring. Problemet består i at afgøre hvordan man sikre at det samme map bliver genereret hver gang spillet loades.

Gem en map layout fil

Denne løsning viste sig at være den bedste løsning for at sikre sig, at map layoutet kun skulle eksistere i en enkel folder i projektet. Men det viste sig vanskelig at læse en sådan fil uafhængigt af pc'en programmet blev kørt på.

Lav en klasse som generer map layout filen

Ultimativt blev dette løsningen, som benyttes i projektet. En MapCreator klasse danner en map layout fil når dens konstruktor bliver kald.
En map klasse som store map layoutet kan nu læse layout filen og danne et map udfra denne fil. Filen består af linjer med formen *"leftRoomId, TopRoomId, RightRoomId, BottomRoomId"*. Den først linje dækker room 1, næste linje dækker room 2 osv.
Hver linje i map layout filen mappes til en liste af roomId'er, der kan insertes i Map klassens mapLayout listea. Denne Liste danner grundlaget for spillets map og diktere hvordan spilleren kan navigere i spillet. Denne mapping mellem strings og int array er vist i Figure 30.

Items og Enemies

Som det sker for rooms, ligeledes sker der for enemies og items. MapCreator klassen generer

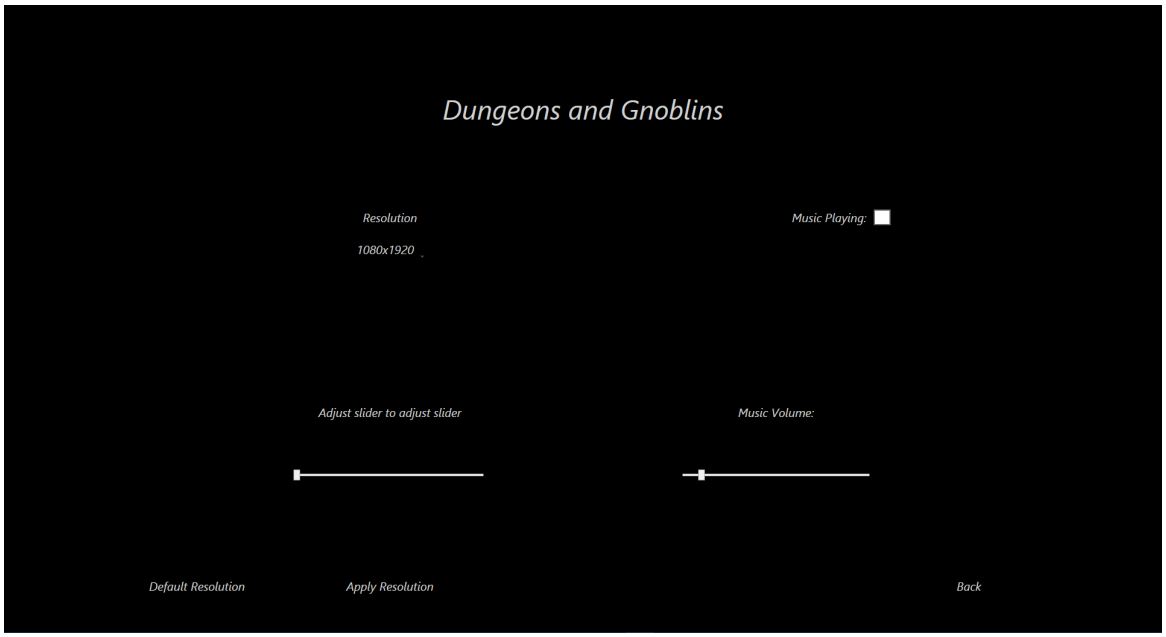

Figure 26: Menu til at ændre spillets indstillinger. Det er her muligt at vælge skærmopløsning og lydstyrke, samt tænde og slukke for musikken. Back knappen fører tilbage til enten main menu eller ingame menu, afhængig af hvilke menu man tilgik settings menuen fra.

seperate filer for enemy positions og item lokationer. Map klassen kan herefter læse filen og mappe hver linje i filen til, en enemy eller item og placere det i det korrekte room.

4.3.3 Log

Komunikkation med front end fra GameControlleren og CombatControlleren gør brug af en log. Logen består af et dictionary datastruktur som GameControlleren og CombatControlleren benytter til at logge vigtige infomation, som frontend kunne have brug for at vise til spilleren.

Dette kan være infomation om spillerens bevægelse fra et Room til et andet. Det kan også være infomation omkring spillerens eller enemies tilbageværende livsmængde osv. Dette isolere frontend fuldt fra Game Engine, da frontend ikke har andre accesspunkter til infomation om hvad der sker i spillet.
Derved opnår frontend seperation of respondsibility da frontend nu kun er ansvarlig for at display infomation som det er givet af game enginen.

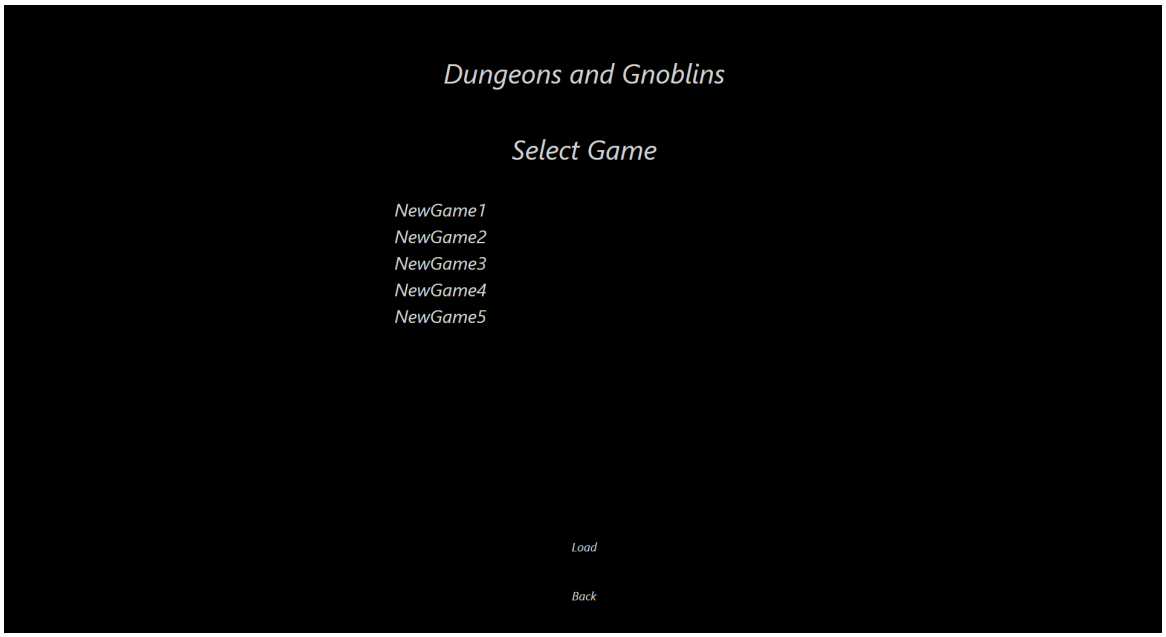

Figure 27: Load menu view. Her præsenteres spilleren for alle gemte spil. Disse hentes fra databasen hver gang spilleren går ind i load menuen.

4.4 Kommentar til implementeringen

Gennem udviklingen af Game Enginen er der brugt interfaces eller abstrakte klasser til implementering af alle funktionaliteter. Dette gør det både nemt at teste implementeringerne sam at udvide spillet. Men for scopet af opgaven er dette nok "Overkill". Det er godt at bruge men for en applikation så lille som denne, der ikke kommer til at udvide sig er det nok for meget boilerplate kode.

5 Test

Testing er en grundsten i ethvert succesfuldt softwaresystem. Uden testing kan der ikke stilles garanti for et systems opførsel. Nedenstående afsnit dækker alle test foretaget af "Dungeons and Gnoblins" spillet, for at sikre den korrekte opførsel i henhold til kravspecifikationerne.
Her Dækkes test af alle de største komponenter, Frontend, Game Engine, Backend og database. Testene inkluderer automatiseret unittests, integrationstest og accepttest.

5.1 Modultest Frontend

Modultesten af Frontenden er lavet i meget tæt samarbejde med Game Engine holdet. Dette er valgt sådan, at når Game Engine lavede en ny funktion eller funktionalitet, gik frontend holdet igang med at implementere en visuel repræsentation af denne nye funktion/funktionalitet. Således var det ikke kun en visuel test af at views så godt ud eller at man kunne trykke på en knap, men derimod kunne både frontenden og Game Enginen testes i samarbejde, hvor den reelle funktionalitet for systemet blev testet.

5.1.1 Test metoder

Hver gang der er blevet lavet små eller større ændringer i frontenden, er der blevet lavet både en funktionel og visuel test af de nye ændringer. Dette blev gjort for æstetiske ændringer ved at kigge i preview vinduet i WPF for det view der blev ændret. Var det derimod en ændring der inkluderede databinding til Game Enginen, blev det nødvendigt at teste hele programmet ved brug af compileren og derved lave en runtest, som inkluderede at det rigtige data blev hentet fra Game Enginen eller at den rigtige funktionalitet blev kaldt ved et tryk på en knap.

5.1.2 Eksempel på frontend test i forbindelse med Game Engine

Et godt eksempel på hvordan frontend testene blev kørt er ved implementeringen af Room-viewets map element og mere specifikt bevægelsen fra et rum til et andet, altså de implementerede "Go North,East,South,West" knapper, som krævede både en visuel og funktionel del.
Den visuelle del bestod i at kortet skulle opdateres, spilleren flyttes og rum-beskrivelsen opdateres.
Den funktionelle del bestod i at Game Enginen skulle kontaktes på korrekt vis med rigtige parametre, de rigtige databindings skulle opdateres, således at den visuelle del blev opdateret korrekt og derefter skulle informationen om det rum gemmes korrekt i Game Enginen.
For at være sikker på at alt data blev opdateret korrekt blev programmet kørt i debug mode, hvor der kunne sættes break-points i koden og værdierne på diverse variabler, såsom roomdescription og currentroom kunne undersøges. Samtidig blev der kigget på den visuelle del af selve viewet, hvor der blev tjekket om beskrivelsen blev korrekt displayet, mappen opdateret korrekt og at spilleren flyttes korrekt.

5.1.3 Eksempel på frontend test

Ikke alle moduler krævede at Game Enginen blev kontaktet. Eksempelvis Mediatoren, som er beskrevet i Frontend implementings afsnittet som kan findes her: subsection 4.2. Med dette modul blev der lavet mocks, hvor vi satte en dummy knap op til at kunne kalde notify() funktionen i Mediatoren og der blev tjekket visuelt om viewet blev skiftet uden at hele applikationsvinduet blev skiftet og at det korrekte view blev vist.

5.2 Modultest Game Engine

Game Engine styrer spillets indre logik. Dette dækker over alt fra spillerens bevægelse gennem spillet til "save" og "load" af nye og gamle spil. Det er derfor yderst nødvendigt at denne er fuld testet.
For at sikre Høj kvalitet er Game Engine skrevet med Robert C. Martins "Three Laws of TDD" [**CleanCode**] i baghoved hvilket førger til at Game Engine er eksklusivt test ved hjælp af black-box testing. Alle testene tester kun det offentlige interface, som er gjort tilgængelig.

5.2.1 GodkendelsesTabel Game Engine

Nedestående præsenteres en fuld tabel for alle classes testet som led i Game Engine. De vigtigste er GameController, CombatController, Player/Room og DiceRoller. Dice Danner "Core Mechanics" for spillet.
Interessant nok ses her at GameController fejler sine test grundet at der ikke er skrevet test til mange af GameControllerens ansvars punkter.

Table 1: Her Ses en komplet liste over alle test foretages på Game Engine komponenter, med kommentar til deres resultater og en endelig vurdering af test resultaterne.

Game Engine GodkendelsesTabel			
Komponent under test	**Forventet Adfærd**	**Kommentar**	**Test Resultat**
Game Controller	1. Kan skifte Player til Nyt Room 2. Kan samle Item op fra Room 3. Kan save Game 4. Kan loaded Games 5. Kan eliminere Enemy fra Game 6. Kan reset Game 7. Kan anskaffe Room description	Game Controller Har kun test for at skifte til nyt Room, dette betyder at der ikke kan stilles garanti for at resterende implementeringer af load- og save game osv. fungere som ønsket. Disse ting er svagt testet gennem visuelt trial and error test. Men fordi der ikke er skrevet nogen specifikke test til dem Fejler GameControlleren sin komponent test.	FAIL
Combat Controller	1. Kan Håndtere Combat Rounds 2. Kan håndtere at Player løber væk fra Combat	Combat controller kan håndtere at spilleren løber fra combat og at Player engager i combat. CombatController can stille garanti for at combat sker i den rigtige orden og at hverken spiller eller enemy kan angribe hvis denne er død. Ydmere stiller den garanti for at både enemy og spiller kan lave "critial hits" hvis dice rolleren slår 20.	OK

DiceRoller	1. Kan emulerer et kast med en N siddet terning 2. Kan emulerer N kast med en M siddet terning	DiceRoller Kan emulere et eller flere terninge kast med samme antal sidder. Denne kan ydmere stille krav for at fordellingen af disse terningekast har en normal distribution og dermed er alle udfald lige sandsynlige.	OK
BaseMapCreator	1. Kan generere et map layout file	BaseMapCreator can på korrekt vis generere et map layout file, den kan ydmere generer item layout files og Enemy layout files, der hjælper Map klassen med at genererer spillet Map. Der ikke skrevet test for eksistensen af item og enemy layout Map og derfor kan der ikke stille garanti for at disse bliver genereret på korrektvis. Testen Fejler derfor.	FAIL
BaseMap	1. Kan anskaffe Rooms ud fra en Direction	Map kan anskaffe Rooms uf fra en given Direction. Map kan på korrektvis generer et map udfra mappets layout file. Den kan også på korrektvis finde udaf hvilke Rooms har adgang til hinanden. Der er ikke skrevet test for at bekræfte at enemy og items er i de korrekte lokationer baseret på enemy og item layout filerne. Derfor fejler denne sin Test.	FAIL

Player	1. Kan angribe Enemy 2. Kan tage skade 3. Kan skade Enemy (Ikke Kristisk) 4. Kan skade Enemy (Kritisk)	Player kan angribe, skade og tage skade fra enemy.	OK
Enemy	1. Kan angribe Player 2. Kan tage skade 3. Kan skade Player (Ikke Kristisk) 4. Kan skade Player (Kritisk)	Enemy kan angribe, skade og tage skade fra Player.	OK
Log	1. Kan log et event 2. Kan anskaffe et event 3. Kan merge to logs	Log kan logge et event, anskaffe eventet og merge to forskellige logs.	OK
Room	1. Kan tilføje Player 2. Kan tilføje Enemy 3. Kan fjerne Player 4. kan fjerne Enemy	Room kan fjerne/tilføje Player og Enemy som forventet	OK
Items		Items så som sword, shield, axe osv. indeholder kun data og har derfor ikke nogen testbar adfærd andet end deres constructor. De kan alle construeres på korrekt vis.	OK

5.2.2 Mock testing

I nogle tilfælde kan det være umuligt er lave troværdige tests, når en klasse f.eks. Player er dyb afhængig af DiceRoller klassen. Dette skyldes at DiceRoller danner pseudo-random outputs og derfor er test med den ikke altid troværdige.
Her benyttes mock tests og dependency injections til at sikre at DiceRoller klassen danner de samme outputs hver gang en test køres. Derved kan alle senarier for Player klassen testes, hvor man kan regne med at DiceRoller giver et bestemt output.

5.2.3 Test Resultater for Game Engine

Nedestående vises kort resultatet for Game Engine test, når de alle køres via visuel studio. Som det kan ses så er alle testene succesfulde, hvilket giver hvis garanti for at game engine opfører sig som specificeret i kravspecifikationerne og i de implementerede interfaces.

5.3 Discussion af Test Result

Projektets implementering og kravspecifikationerne har produceret adskillige tests, som projektet i overvejende grad har bestået. De fleste features er blevet testet i henhold til kravspecifikationerne se **(REF)** og har produceret resultater, der indikerer at hver feature fungerer som ønsket.

Alle funktionelle tests er beskrevet med user-stories og er evalueret med et "Godkendt" eller "Fejl" (**REF Here**) og evt. en foklarende kommentar. Ikke funktionelle test er har fået en evaluering "OK" eller "FEJL" og kan ses mere detaljeret i (**REF HERE**). Langt størstedelen af alle tests, for produktet, er bestået med få tests som fejler. Nogle tests fejler, da implementeringen ikke blev som forventet, mens andre ikke er blevet implementeret på grund af tidspres.

To eksempler er "puzzles" og "Delete Save Game", der skulle have været implementeret, som en del af det færdige spil. Grundet tidspres er disse features ikke blevet implementeret, men i stedet for, er de blevet nedprioteret til fordel for andre features, såsom "Combat" der er blevet vurderet mere essentiel. "Delete Save Game" er ikke blevet implementeret som specificeret i kravspecifikationerne men eksisterer i stedet for, som evnen til at overskrive allerede eksisterende save games. Her er kravende ikke blevet opdateret til at reflektere den anderlede implementering
Den stores success rate skyldes en insistens på tidlige integration mellem alle store komponenter for at sikre, at alt kommunikation mellem frontend, game engine, backend og databasen har fungeret fra et tidligt tidspunkt i projektet. I stedet for at integrere og teste alting samtidigt, er hvert komponent blevet gradvist integreret i projektet. Det har vist sig nemmere at integrerer mange små ændringer end at lave en stor integration.
Lad der dog ikke herske tvivl om, at der er store huller i projektets test suit, det betyder at store features ikke er testet i tilstrækkeligt omfang. Features som load- og save game er implementeret og testet ved visuel bekræftelse, men der er en betydelig mangel på modultest til yderligere at bekræfte, at disse feature fungerer som ønsket (**REF HERE**).

Den stores success rate skyldes en insistens på tidlige integration mellem alle store komponenter for at sikre at alt kommunikation mellem frontend, game engine, backend og databasen har fungeret fra et tidligt tidspunkt i projektet. I stedet for at integrere og teste alting samtidigt, er hvert komponent blevet gradvist integreret i projektet. Det har vist sig nemmere at integrerer mange små ændringer, end at lave en stor integration til sidst i udviklingsfasen.

5.4 Database modultest

Der er som gruppe taget en beslutning om at hoste databasen lokalt i en docker container. Dette blev valgt på baggrund af en samtale med vejleder, hvor vi blandt andet, grundet sikkerhedsforanstaltning på au's netværk, kunne løbe ind i nogle problemer. Den lokale hosting medførte at vi kunne holde øje med databasens udformningen, samt den gemte data gennem Microsoft azure datastudie.
For at teste DAL funktioner og dens forbindelse til databasen, oprettes først et DAL objekt med en context som indeholder en korrekt connectionstring.
Til test hvor der skal indsættes data i databasen oprettes der objekter af den korrekte type hvorefter DAL funktioner til indsættes kaldes. Korrektheden af de indsatte data kan herefter tjekkes med datastudie.
Et eksempel på dette kan ses på figur xx herunder. Der oprettes et DAL objekt, datahelper, og et GameDTO objekt, gamesave. Her opsættes gamesave til "Gamer1", med navnet "My First Run", hvorefter der tilføjes yderligere data. Det overskrevne save er "NewGame1", som her har id 1.

```
var DataHelper = new DAL(context);

var save = DataHelper.GetSaveByID(1);

var gamesave = new SaveDTO();
gamesave.ID = save.ID;
gamesave.Username = "Gamer1";
gamesave.SaveName = "My First Run";
gamesave.RoomID = 13;
gamesave.Health = 69;
gamesave.ItemsID.Add(13);
gamesave.Enemies_killed.Add(2);
gamesave.PuzzleID.Add(3);
gamesave.VisitedRooms.Add(1);
gamesave.VisitedRooms.Add(2);

DataHelper.SaveGame(gamesave);
```

Figure 34: Kode til test af SaveGame

På figur xxx herunder ses et screenshot fra datastudie hvor de 5 saves til "Gamer1" kan ses. Det noteres at alle saves er "blanke" og starter i rum 1.

Results Messages

	ID	RoomID	Armour_ID	Health	Weapon_ID	Username	SaveName
1	1	0	NULL	10	NULL	gamer1	NewGame1
2	2	0	NULL	10	NULL	gamer1	NewGame2
3	3	0	NULL	10	NULL	gamer1	NewGame3
4	4	0	NULL	10	NULL	gamer1	NewGame4
5	5	0	NULL	10	NULL	gamer1	NewGame5

Figure 35

Herefter køres programmet fra figur xx(kode), og vi kan nu se ændringerne i databasen. Det noteres at SaveName og de andre attributter nu er opdateret korrekt efter koden.

Results Messages

	ID	RoomID	Armour_ID	Health	Weapon_ID	Username	SaveName
1	1	13	NULL	69	NULL	gamer1	My First Run
2	2	0	NULL	10	NULL	gamer1	NewGame2
3	3	0	NULL	10	NULL	gamer1	NewGame3
4	4	0	NULL	10	NULL	gamer1	NewGame4
5	5	0	NULL	10	NULL	gamer1	NewGame5

Figure 36

Øvrige lister er også opdateret korrekt, som set på figur xx herunder.

ItemID	SaveID
13	1

EnemyID	SaveID
2	1

VistedRoomId	SaveId
1	1

Puzzles_ID	Save_ID
3	1

Figure 37

Funktioner hvor der skal læses fra Db kaldes DAL funktionen hvorefter den fundne information udskrives til konsolen ved hjælp af console writeline. Korrektheden af data i konsolen dobbelttjekkes med datastudie. Et eksempel på dette ses på figur xx herunder. Her henter vi de gemte save fra tidligere fra "Gamer1" ved navn "My First Run", hvorefter data fra dette save udskrives i konsolen.

```
var save = DataHelper.GetSaveByID(1);

Console.WriteLine("Username: " + save.Username);
Console.WriteLine("Current Room: " + save.RoomID);
Console.WriteLine("SaveName: " + save.SaveName);
Console.WriteLine("Health: " + save.Health);
Console.WriteLine("Armour id: " + save.Armour_ID);
Console.WriteLine("Weapon id: " + save.Weapon_ID);
Console.WriteLine("Items in inventory: ");
foreach (var i in save.ItemsID)
{
    Console.WriteLine(i+" ");
}
Console.WriteLine("Enemies killed: ");
foreach (var i in save.Enemies_killed)
{
    Console.WriteLine(i + " ");
}
Console.WriteLine("Puzzles solved: ");
foreach (var i in save.PuzzleID)
{
    Console.WriteLine(i + " ");
}
Console.WriteLine("Visited rooms: ");
foreach (var i in save.VisitedRooms)
{
    Console.WriteLine(i + " ");
}
```

Figure 38

Når kodestykket ovenfor køres får vi resultatet i konsol som set på figur xx herunder. Det noteres at det udskrevne data stemmeroverens med det indsatte data fra den tidligere funktionstest.

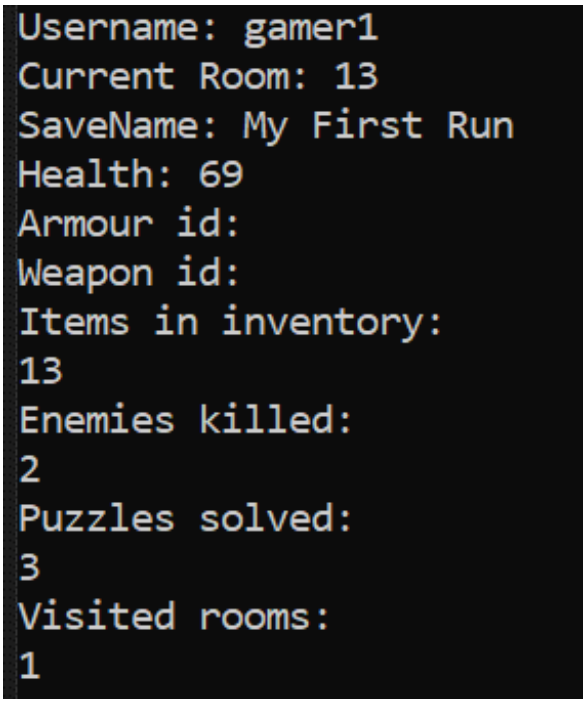

Figure 39: Konsoleudskrift

Test	**Funktion**	**Forventet resultat**	**Observering**	**Vurdering (OK/Fail)**
1	GetRoomDescription(id)	Her forventes det at funktionen henter beskrivelsen fra den valgte rumId.	Funktionen henterbeskrivelsen korrekt og vi kan udskrive denne på konsolen.	OK
2	GetAllSaves	Det forventes at alle spillets saves, med tilhørende info, hentes fra databasen	Alle saves hentes fra databasen og information om disse kan udskrives på konsolen.	OK
3	GetSaveById(id)	Det forventes at der hentes alle oplysninger om et enkelt save fra databasen med tilsvarende id, som den medsendte parameter	Det korrekte save samt tilhørende info hentes og kan udskrive til konsolen	OK
4	SaveGame(Game)	Det forventes at det valgte save med samme id som det nye, overskrives, og at tilhørende info opdateres	Det observeres at det gamle save med samme id, nu er ændret og har korrekte nye værdier	OK

5.5 Funktionsbeskrivelse

5.5.1 Game Controller

public async Task Reset()
Return Type: Task
Parameter: void
Comment: Resetter game state til default tilstand.
public async Task GetRoomDescription()
Return Type: Task
Parameter: void
Comment: Initializere alle room descriptions fra databasen til de korrekte Rooms.
public async Tast SaveGame()
Return Type: Tast
Parameter: int id, string Savename
Comment: Gemmer game state og sender de via backend til databasen hvor det bliver gemt med det parameter id og navn.
public async Task LoadGame()
Return Type: Task
Parameter: int id
Comment: Resetter game state til default, skaffer save game med det givet id og initializer game state til at matche Save game statet.
public ILog Move()
Return Type: ILog
Parameter: Enum Direction
Comment: flytter spilleren i den angivet retning hvis og kun hvis en forbindelse eksistere mellem det nuværende room og den ønskede bevægelses retning.
public void EliminateEnemy()
Return Type: void
Parameter: void
Comment: Fjerner en enemy fra game state.
public void PickUpItem()
Return Type: void
Parameter: Item
Comment: Tilføjer et item til spilleren inventory og fjerne denne fra room.

5.5.2 Log

public void RecordEvent()
Return Type: void
Parameter: string key, string value
Comment: Gemmer et vent i dictionarien
public string GetRecord()
Return Type: string
Parameter: string key
Comment: Udskriver et event ud fra den givene key string.
public static ILog operator +()
Return Type: ILog
Parameter: ILog a, Log b
Comment: Tilføjer alle events fra en log til en anden log.

5.5.3 DiceRoller

public uint RollDice()
Return Type: uint
Parameter: uint numOfSides
Comment: Simulere et terninge kast og returnere et tal mellem 1 and numOfSides (Incl.)
public uint RollDice()
Return Type: uint
Parameter: (uint NumOfSides, uint NumOfDice)
Comment: Simulatere flere terningeslag og returnere summen af disse kast.

5.5.4 Room

public void AddPlayer()
Return Type: void
Parameter: Player player
Comment: Tilføjer en spiller til et room.
public void RemovePlayer()
Return Type: void
Parameter: void
Comment: fjerner en spiller fra et room.
public void AddEnemy()
Return Type: void
Parameter: Enemy enemy
Comment: tilføjer en enemy til et room.
public void RemoveEnemy()
Return Type: void
Parameter: void
Comment: fjerner en enemy fra et room.

5.5.5 Combat Controller

public EngageCombat ILog Emulere en rundte i combat, hvor spilleren slår først og hvis enemy dør, stopper combat ellers slår enemy på player. ()
Return Type: ILog
Parameter: ref Player player, ref Enemy enemy
Comment:
public void Flee()
Return Type: void
Parameter: void
Comment: stopper combat.

Figu nden.
Der

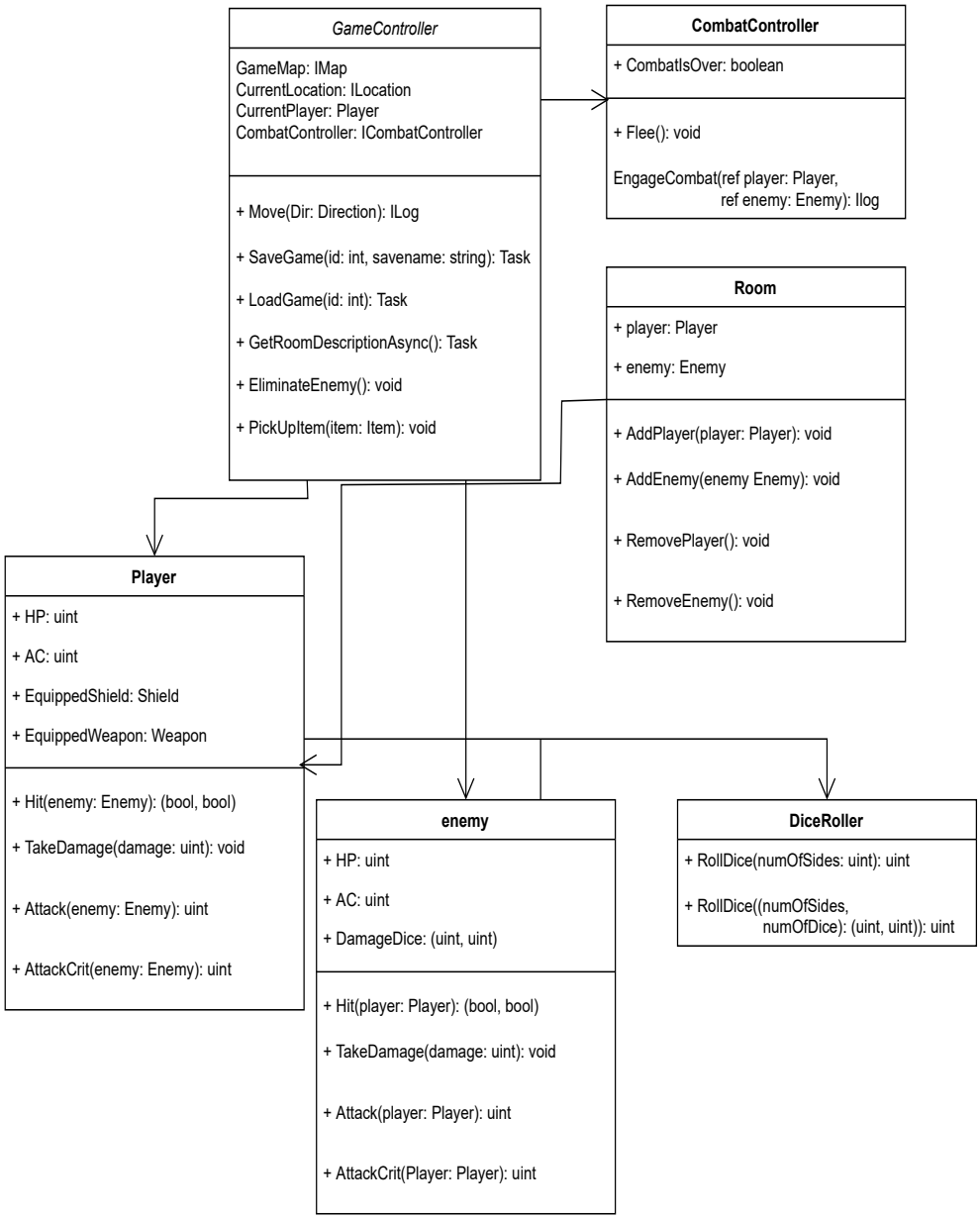

Figure 29: Load funktionen, resetter game, setter de rigtige states og gennemløber alle Room instances og sætter deres state til den korrekte state.

```
public async Task LoadGame(int id)
{
    await Reset();
    SaveDTO Game = await backEndController.GetSaveAsync(id);
    CurrentLocation.RemovePlayer();
    CurrentLocation = GameMap.Rooms[Game.RoomId];
    VisitedRooms = Game.VisitedRooms;
    SlainEnemies= Game.SlainEnemies;
    Inventory = Game.Inventory;
    CurrentPlayer.HP = Game.Health;
    GameMap.Rooms[CurrentLocation.Id].AddPlayer(CurrentPlayer);
    List<(uint,Item)> temparray = new List<(uint,Item)>();

    foreach (ILocation room in GameMap.Rooms)
    {
        if (room.Chest != null)
        {
            foreach (Item item in room.Chest)
            {
                if (Inventory.Contains(item.Id))
                {
                    if (Game.WeaponId == item.Id)
                    {
                        CurrentPlayer.EquippedWeapon = (Weapon)item;
                    }
                    if (Game.ShieldId == item.Id)
                    {
                        CurrentPlayer.EquippedShield = (Shield)item;
                    }
                    CurrentPlayer.Inventory.Add(item);
                    temparray.Add((room.Id, item));
                }
            }
        }

        foreach ((uint TempRoom,Item TempItem) tempval in temparray)
        {
            GameMap.Rooms[tempval.TempRoom].Chest.Remove(tempval.TempItem);
        }

        if (room.Enemy != null)
        {
            if (SlainEnemies.Contains(room.Enemy.Id))
            {
                room.RemoveEnemy();
            }
        }
    }

}
```

Figure 30: viser illustrer hvordan en linje fra map layout filen omdannes til en liste af room id'er som rummet er forbundet med. Dette er kerne mekanismen for hvordan en spiller kan navigere rund i spillet, da en spiller ikke kan bevæge sig fra et Room til et anden, der ikke er i denne liste af forbindelser mellem det nuværende room og den ønskede bevægelses retning.

```
1 reference
private LinkedList<int> CreateRoomLink(string linkingString)
{
    int[] link = linkingString.Split(",").Select(int.Parse).ToArray();
    return new LinkedList<int>(link);
}
```

Figure 31

```
[TestFixture]
0 references
public class PlayerTest
{
    private Player uut;
    private Enemy enemy;
    private DiceRoller basicDiceRoller;
    private Weapon weapon;

    [SetUp]
    0 references
    public void SetUp()
    {
        basicDiceRoller = Substitute.For<DiceRoller>();
        weapon = Substitute.For<Sword>(((uint) 8, (uint) 1), (uint) 2, (uint) 2);
        uut = new Player(10, 16, weapon, basicDiceRoller);
        enemy = new Enemy(10, 10, (10, 1), 2, "test");
    }
```

Figure 32: Mocks benyttes her til at sikre at dependencien, her DiceRoller, returner den ønskede værdi for situationen, der er under test. Alle test tildeles informative navne, for at sikre læsbarhed i forhold til testenes formål.

```
[Test]
0 references
public void Hit_IfPlayerEquippedWeaponNull_DefaultHitIsFalse()
{
    basicDiceRoller.RollDice(20).Returns((uint)10000);
    uut.EquippedWeapon = null;
    bool expectedHitResult = false;
    (bool Hit, bool) actualHitResult = uut.Hit(enemy);
    Assert.That(actualHitResult.Hit, Is.EqualTo(expectedHitResult));
}
```

Figure 33: Alle skrevene test til Game Engine passer, hvilket hjælper med at give vished om at Game Engine udfører dens funktionalitet, som det er beskrevet i kravene. Dette siges da alle testene er skrevet på baggrund af kravene som black-box tests og ikke som white-box test efter implementeringen.