Exam 2018

Software Development 2018

Department of Computer Science
University of Copenhagen

Emil Møller Hansen <ckb257@alumni.ku.dk>,
Casper Bresdahl <whs715@alumni.ku.dk>,
Torben Olai Milhøj <vrw704@alumni.ku.dk>

Version 3.8.9;
Due: Wednesday, June 6th

1 Indledning

Spacetaxi er et gammelt spil med formål at flyve passagere fra en destination til den anden i rummet. Man får points for hver passager man sætter korrekt af, hvis man er for langsom til at sætte passageren af formindskes antallet af points. Udfordringen i dette ligger i at man både skal flyve hurtigt for at sætte passageren af inden for den specificerede tid og at man samtidig skal undgå forskellige forhindringer undervejs.
Første aflevering har været med henblik på at hente ASCII-karakterer fra en txt-fil og dernæst kunne forbinde disse karakterer med tilhørende billeder, således af en grafisk implementation af første level blev mulig. Anden aflevering har implementeret spilleren, indført game states og implementeret collision detection med vægge og platforme. Trejde aflevering har omhandlet implementering af customers, timers og points således spillet nu har et mål og kan spilles.

Generelt for vores afleveringer har fokus specielt været design-processen. Hvordan og hvorledes spillet bør implementeres er blevet grundigt overvejet igennem opgavernes forløb, for ikke at skabe hindringer fremme i tiden og for at have en solid implementering, der overholder forskellige principper som eksempelvis "The expert Doer Principle", "The High Cohesion Principle" og "The low Coupling Principle". Implementeringen er yderligere visuelt repræsenteret igennem et UML-diagram og et sequence diagram.

2 Baggrund

Til dette projekt har vi benyttet DIKUArcade som game engine. Denne har mange funktioner som har hjulpet os med implementeringen i alle afleveringerne. Målet er at man skal kunne spille spillet, dette inkluderer at kunne hente og afsætte kunder i 2 levels, samt få points for dette, man skal også kunne støde ind i forhindringerne i banerne, og dette skal få spillet til at slutte. Programmet skal kunne køres med Mono (Version $\geq$ 5.10). Af andre dependencies gøres der også brug af DIKUArcade, som benytter sig af OpenTK 2.0.

3 Analyse

Vi har indelt problemet i forskellige concepter, der hver har deres eget ansvar, resultatet kan ses herunder.

Responsibility description	Concept name
Læser fil og returnerer Level objekt med info herfra	File reader (FR)
Class til at beskrive Level objekter	Level class (LCl)
Singleton der skal indeholde en dictionary af Levels	Levels keeper (LK)
Sørger for alle levels loades før start på spil	Level loader (LL)
Laver en entity container til level baseret på Level objekt	Level creater (LCr)
Returnerer en entity som repræsenterer blok i spillet	Entity creater (EC)
Sørger for at skifte mellem vores states	StateMachine
Vores game states	GameState
Sørger for at der kan subscribes til events	SpaceBus
Spawne customers med deres info samt udregne point	Customer
Ansvar for customers	Customers
Oversætte data til Customer	CustomerTranslator

Table 1: Dele af vores løsning inddelt i responsibilities og koncepter

Vi har opdelt koncepterne efter koncepttyperne Know og Do, så hvert koncept har netop én type. Vores Do koncepter har hver ansvar for én opgave, eksempelvis file reader der kun skal lave et level ud fra informationen fra en fil. Vi har 11 koncepter hvoraf 10 er Do- og 1 er know-koncepter.
Vores GameState koncept omhandler flere klasser som er ens i struktur men variere i funktionalitet således at vi for eksempel kan pause og tabe.
I tabel 2 herunder ses det hvordan de forskellige koncepter skal kommunikere med hinanden.

Concept pair	Association description	Association name
Level Loader og file reader	Giver filnavn og får level tilbage	Hent data
File reader og level class	Laver Level objekt	Genererer
Level loader og levels keeper	Levels keeper gemmer Level fra loader	Gem data
Level creator og levels keeper	Får Level objekt fra Keeper	Hent data
Level creator og entity creator	Får entity med givne parameter	Genererer
GameStates til StateMachine	Anmoder om at skifte til en ny state	request
Customer til GameStates	Customer beregner points som i GameRunning lægges til	gem data
StateMachine til GameStates	Giver events videre	Kommunikerer
SpaceBus og GameStates	GameStates registrerer events til SpaceBus	Kommunikerer
SpaceBus og StateMachine	SpaceBus kommunikerer input events	Kommunikerer
CustomerTranslator og Customer	CustomerTranslator parser data til Customer	Parser data

Table 2: Associationer mellem forskellige concepter

Det ses at vi har besluttet at Level loader skal kalde en funktion i file reader for at denne returnerer et Level som Level loader gemmer i levels keeper. I starten af planlægningsprocessen var det egentlig planen at file reader skulle sende Level-objektet til levels keeper, men da vi fik et bedre overblik over de forskellige ansvarsområder kunne vi se at det gav bedre mening at gøre det som beskrevet ovenfor. På denne måde får vi nemlig en klarere adskillelse mellem Level loaders og file readers ansvar, i og med at file readers eneste job

er at læse og parse filen, hvorefter level loader behandler Level-objektet.
I begyndelsen var det også meningen at vi ville have at vores Customer koncept bestod af instanser af Customers som selv havde ansvaret for at finde ud af hvornår de skulle spawne. Dette kom vi dog fra idet det krævede rigtig meget kommunikation mellem StateMachine, GameRunning og Customer, og vi besluttede derfor at lade GameRunning håndtere vores timedevents.
I tabel 3 ses det hvilke attributer vi tænker at hvert concept skal have.

Concept	**Attribute**	**Attribute description**
Level loader	Levels	En liste af filnavnene på level layouts
File reader	Ingen attributer	
Level class	Level array	2D-struktur der indeholder level design
	Name	Navn på level
	Costommer	Information om costommers i dette level
	Platforms	Info om hvad der er platforme
	Decoder	Gemmer sammenhængen mellem karakterer og billedenavne
Levels keeper	Levels	Contatiner til alle levels
	Index	Tæller der holder styr på hvor mange levels der er I levels
Level creator	Levels	Reference levels keepers
Entity creator	Ingen attributer	
StateMachine	ActiveState	Refference til den aktive state
GameState	Instance	Refference til sig selv (Singleton)
SpaceBus	Ingen attributer	
Customer	Ingen attributer	
Customer	SpawnPlatform	Hvilken platform som denne customer skal spawne på
	Destination	Hvilken platform denne customer skal af på
	Points	Hvor mange points denne customer giver
	Time	Hvor lang tid man har til at aflevere denne customer
CustomerTranslator	Ingen atributter	

Table 3: Attributer for hvert koncept

Det ses at det ikke er alle vores koncepter der har attributer, dette skyldes at det ikke er dem alle der skal gemme informationer. Eksempelvis skal file reader blot læse filen og generere objekter på baggrund af dette, den behøver altså ikke gemme informationen til senere brug. I modsætning til dette skal hvert Level objekt gemme informationen om den selv.

3.1 Målsætninger

Vores indfaldsvinkel på vores design vil være at sørge for at hver class i vores program kun vil have **et ansvar** (SRP), og at klassen selv kan udføre opgaverne som skal til for at klare ansvaret. Vi vil også skrive kode som **let kan udvides** (OCP), og som let kan **bygges videre på** (LSP), samtidigt med at det vil være nemt at finde bugs og maintaine i fremtiden. Da vi laver objekt-orienteret programmering bestræber vi os efter at lave så **specifikke abstraktioner** som muligt (ISP).
Vi har vurderet at projektet ikke er stort nok til at der behøves at tage højde for

dependency inversion princippet, da det vil være begrænset hvor mange interfaces vi skal lave, og det derfor ikke vil være så nødvendigt at fordele interfaces og implementationer i to projekter.

4 Design

I vores design har vi vægtet at hver class vi implementere har ét ansvar. Det vil sige at det ikke nødvendigvis er den class som indeholder informationen som bearbejder den. At følge dette princip har den svaghed at kommunikationskæden vil blive længere end den behøver hvilket ultimativt vil lede til at vores program vil køre langsommere. Tilgengæld vil vores kodebase være nemmere at maintain da det vil blive tydeligt hvor hvilken opgave bliver udført og det vil derfor også blive nemmere at identificere hvor bugs opstår. Vi har også fulgt princip om at hver klasse selv udfører sit arbejde og vi har derfor ikke klasser som er afhængige af andre klassers metoder, men vi har klasser som er afhængige af andre klasser.
Vi kan bekræfte det ovenstående ud fra vores traceability matrix som mapper hvert af vores koncepter med use cases. Vores traceability matrix indikerer at hver af vores klasser kun har ét ansvar idet at få koncepter matcher flere use cases. Mængden af klasser kan være en indikation på at kommunikationskæden er lang, mens det at der ikke er mange koncepter som mapper til mere end en use case fortæller os at arbejdsbyrden er jævnt fordelt.

	LL	FR	LCl	LCr	EC	LK	GameState	StateMachine	SpaceBus	Customer	CT
Load Levels	x										
Læs filer		x									
Level class			x								
Lav et level				x							
Lav en entity					x						
Hold Levels						x					
Pause Game							x				
Menu							x				
Tab spil							x				
Kør spil							x				
Skift state								x			
Inputs							x	x	x		
Tid							x				
Customer information										x	x

Table 4: Traceablity matrixm, hvor det ses hvordan use cases mapper til klasser.

Ud fra vores sequence diagram, som ses på figur 1, kan det ses at vores design arbejder af flere omgange. LevelLoader beder FileReader om at læse en levelfil hvorefter at FileReader benytter informationerne fra levelfilen til kalde konstruktoren i Level for at lave et nyt level objekt som bliver givet tilbage til LevelLoader. I LevelLoader hentes der en instans af LevelsKeeper som bruges til at gemme level objektet. Dette loop bliver gentaget for alle de levelfiler som er specificeret i LevelLoader hvilket udgør den første del af vores design. LevelLoader bliver brugt ude i Program hvilket sørger for at alle levels i spillet er loadet inden brugeren får givet nogen form for kontrol. Dette vil kunne give problemer med hastigheden af opstart af spillet hvis der skal loades en meget

stor mængde af levels, mens at det vil sikre en hurtig transition mellem levels. Da vi ikke forventer der skal loades store mængder af levels, så vil vi placere en lille ventetid i opstarten af programmet og dermed opnå et hurtigt skift mellem levels.
Dernæst bliver Game instantiatet hvilket resulterer i instantiationen af SpaceBus, StateMachine og MainMenu.
Herefter køres GameLoop i Game hvilket begynder spillet. Når der trykkes 'New Game' fra menuen så skifter StateMachine den aktive state til GameRunning ved hjælp af StateTransformer. Når GameRunning bliver kørt bedes LevelCreator om et array af EntityContainers for de blokke som skal udgøre det level som skal tegnes. LevelCreator indeholder også en instans af LevelsKeeper som nu benyttes til at hente et specifikt level som er blevet gemt. LevelCreator benytter oplysningerne i det hentede level objekt til at bede EntityCreator om at lave en ny entity som skal udgøre en blok i et level. LevelCreator gemmer denne entity i en EntityContainer, og når alle blokke er blevet bygget returneres arrayet af EntityContainere og GameRunning kan derefter render dem.
Da alle vores GameStates er meget ens, så er de samlet under det samme interface. Dette bevirker at StateMachine skal holde en instans af en 'IGameState'. Vores StateMachine er indført ved principperne om en statemachine, som kan beskrives som en form for Mediator pattern som sørger for kommunikationen mellem vores game states. I dette pattern gøres der brug af LSP da forskellige konkrete implementationer af IGameState udskiftes med hinanden.

Vi har valgt at lade spillet løbe i cirkler indtil spilleren selv lukker det. Dette har vi gjort da det gør det nemmere at spille spillet igennem og fordi der ikke er noget formelt krav om at implementere en måde at vinde på. Dette betyder dog også at hvis en customer skal fra et level og over i det næste, og man i stedet for at sætte customeren af i dette level, flyver tilbage til det oprindelige, så kan man ikke længere samle nye customers op, og man kan ikke længere sætte den gamle customer af. Vi mener ikke at denne bug er et reelt problem idet at hvis der stilles et formelt krav om at afslutte spillet, så ville man ikke længere kunne flyve tilbage til det oprindelige level, og man ville kunne implementere et tjek for om spilleren "glemmer" sin customer.

4.1 Design principper

Som tidligere beskrevet har vi lagt stor vægt på Single Responsibility Princippet og vi ser at dette princip i høj grad bliver opfyldt. Hver klasse har en enkelt opgave som den selv løser. Det kan diskuteres om StateMachine og StateTransformer bør være én klasse, men vi har valgt at se StateMachines ansvar som den klasse der skifter states og StateTransformers som værende at oversætte events. Vi har således to ansvar til to klasser men de arbejder sammen om at udføre et fælles mål.
Ud fra vores traceablity matrix ser vi at flere af vores koncepter håndterer input. Dette er muligvis en smule misledende idet at alle koncepterne får hjælp af SpaceBus til dette. Da SpaceBus fungerer som en observer så er det ikke underligt at de klasser som benytter SpaceBus alle vil håndtere inputs.

Ønsker vi at udvide med flere GameStates så kan det ikke lade sig gøre uden at skulle udvide både StateTransformer og StateMachine. Dette betyder at vi ikke følger Open/closed pricippet og det er altså ikke ligetil at skulle udvide vores spil. Dog ser vi at princippet kan følges i forhold til LevelCreator og EntityCreator idet at vi kan skabe en ny type af blokke til vores level ved kun at ændre LevelCreator.

Selvom Liskov Substitution pricippet ligger Open/closed princippet nært, så ser vi alligevel at dette princip er fulgt. Idet vores GameStates er samlet under samme interface, så kan vi nemlig nemt skifte states. Selvom det er besværligt at indføre nye GameStates til spillet, så er det altså ligetil at skifte GameState.

Desværre ser vi at IGameState interfacet er for bredt, idet flere af vores states ikke har en implementation for flere af de definerede metoder. Vi følger derfor ikke Interface segregation princippet. Da IGameState er defineret i DIKUArcade har vi valgt ikke at ændre interfacet selvom det ville være nemt og hurtigt derved at følge princippet.

Vi har valgt ikke at sætte det som en målsætning at skulle følge denpendency inversion, da vi vurderer at projektet ikke er stort nok til at have flere lag af abstraktioner. Det ses dog at vi med GameStates benytter os af abstraktioner, i og med vi bruger et interface IGameState, som en abstraktion og da implementerer dette i flere forskellige konkrete GameStates. Vi har altså ét lag af abstraktion, men denpendency inversion handler generelt om flere lag.

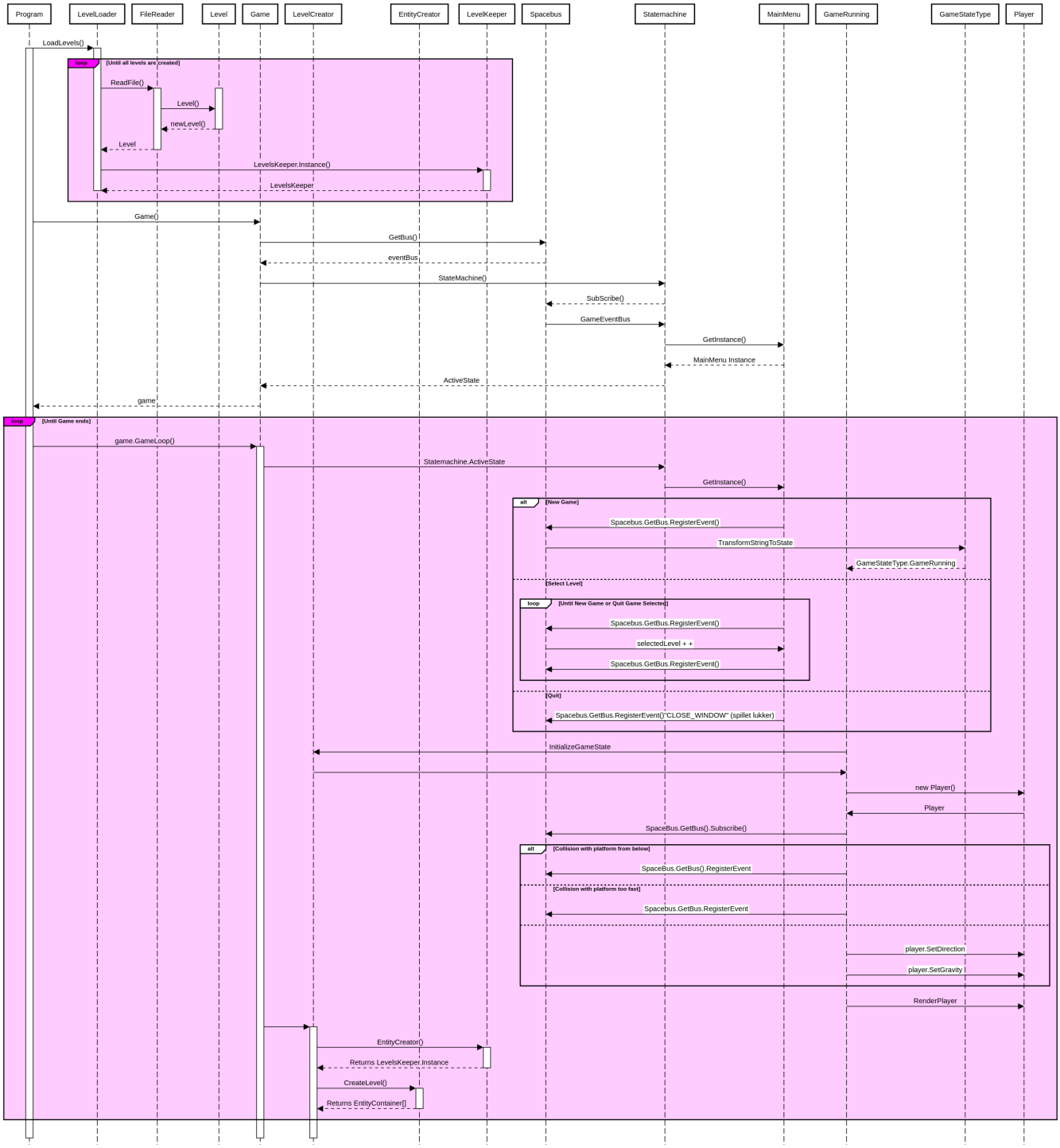
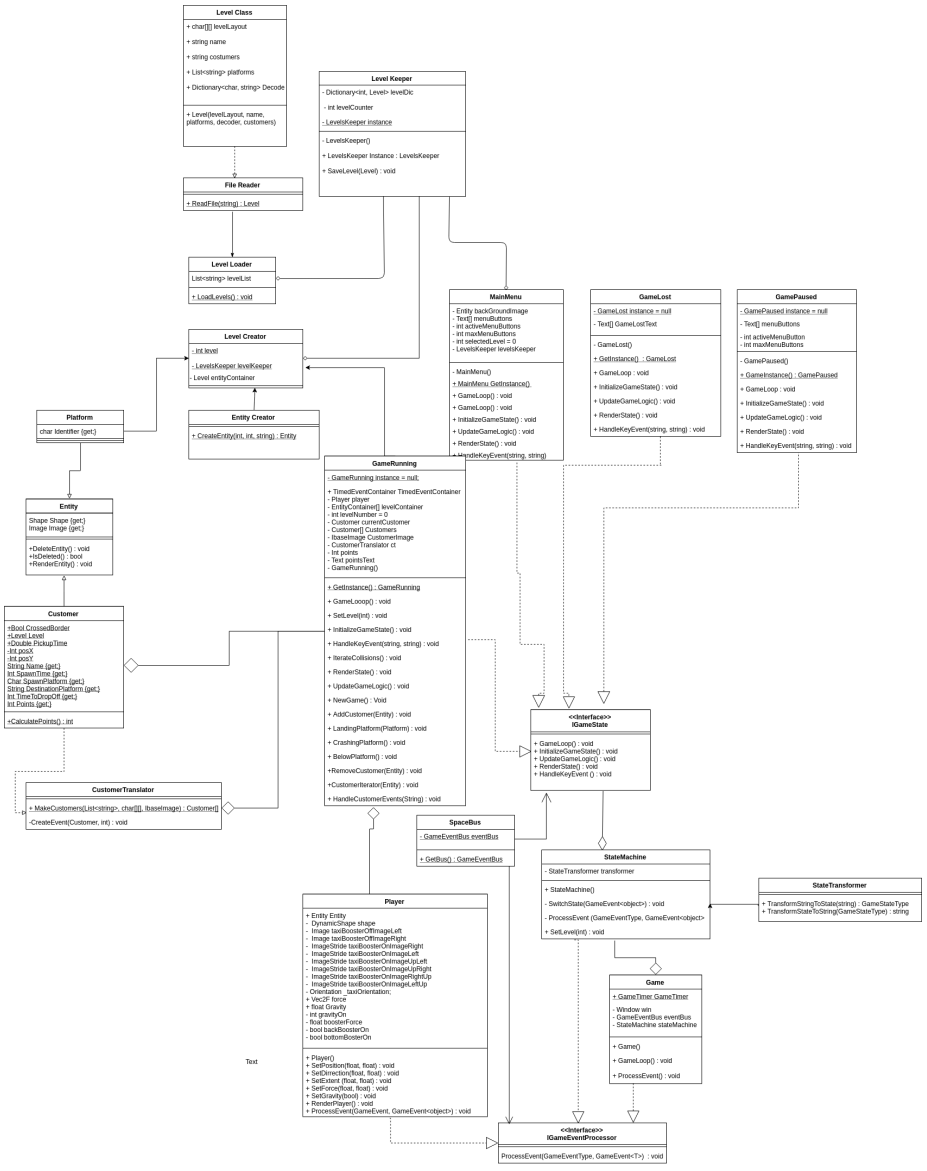


Figure 1: Sequence diagram

5 Implementation



5.1 Level Parsing

Selvom flere af vores klasser kun benyttes en gang i programmet har vi valgt at lade dem være konkrete klasser. Da Vi ikke ser nogen grund til at indføre et singleton pattern hos alle klasserne da der ikke sker noget ved at have flere objekter af for eksempel FileReader, så har vi valgt kun at benytte det på udvalgte klasser såsom LevelsKeeper.
Da vi ønsker at benytte vores Level objekter som en måde at lagre information

om vores levels, har vi valgt kun at implementere en konstruktor til dem og ellers gemme alle informationer i public properties uden setters. Dette garanterer at når Level først er blevet instantieret kan dens informationer ikke overskrives, men informationerne kan til enhver tid læses.
LevelsKeeper fungerer som en database for vores levels, og indeholder en dictionary af level numre som keys og Level objekter som values. LevelsKeeper er implementeret som en singleton. Dette er gjort da vi ikke ønsker at have flere 'halve' databaser med kun en delmængde af vores levels. Vi vil dog have et problem hvis der skrives og læses til det samme sted i hukommelsen samtidigt da vi arbejder med referencer. Men som det fremgik af vores sequence diagram vil dette ikke kunne lade sig gøre med den nuværende implementation, men skulle det blive nødvendigt at udvide vores model med endnu en klasse som skriver til LevelsKeeper samtidigt med at spillet kører, så så skal der implementeres en form for sikkerhed imod dette.
At LevelsKeeper er en singleton bevirker at både LevelLoader og LevelCreator skal holde en variabel for LevelsKeeper objektet, som kan kalde sin private metode til at gemme et Level objekt, eller kan benytte sin public property til at returnere et level hvis den er givet et indeks. Metoden er privat, og propertien har ingen setter for at følge princippet om at hver class udfører opgaverne for sit eget ansvar.

5.2 States og StateMachine

For at gøre det muligt at håndtere vidt forskellige tilstande i spillet, har vi implementeret en state machine. Denne har til opgave at sørge for der bliver skiftet mellem vores states korrekt. Vores implementerede states består af GameRunning som håndterer selve spillet, MainMenu som håndterer vores menu, GamePaused som sørger for at spillet kan pauses og GameLost som fortæller spilleren at spillet er slut. Alle vores states implementer 'IGameState' interfacet hvilket bevirker StateMachine kan holde en instans af en state. Dette bevirker at StateMachine da kan overholde LSP, da vi nemt kan erstatte et GameState med et andet, StateMachine overholder dog ikke OCP, da der skal ændres i StateMachine og StateTransformer for at kunne implementere nye states.
Der er forskellige krav til hvornår der der skal skiftes fra hvilket state til hvilken state, og derfor har StateMachine ansvaret for at kunne tage den rette beslutning om dette. For eksempel skal spillet pauses når spilleren klikker på escape, derfor skal vores EventBus fortælle StateMachine at der er klikket på escape, og StateMachine sørger derefter for at skifte tilstanden til GamePaused. Vi finder også at når spillet tabes, altså når spilleren for eksempel støder ind i en blok, så bedes vores EventBus om at udsende en besked om at skifte tilstanden til GameLost. Der kan altså være flere forskellige måder at skifte tilstanden på hvilket viser at StateMachine er fleksibel.

5.3 Kollision

Vi har lavet collision detection i denne opgave, her har det været nødvendigt at kende forskel på om man er stødt ind i en blok eller en platform. Dette har vi gjort ved at når LevelCreator kaldes returnerer den et array af Entity-

Containers, som GameRunning da kan rendere og lave collision detection på. Collision detection foregår først på index 0 i arrayet af EntityContainers som består af alle platformene, det vil sige at hvis der er kollision her ved vi at man har ramt en platform, og vi kan da fortsætte med de nødvendige efterfølgende checks. Derefter checker den næste index, som består af alle blokkene, hvis en af disse rammes slutter spillet. Hvis der ikke er kollision i nogle af de to EntityContainers tjekkes det om Taxien er over toppen af skærmen, hvis dette er tilfældet ved vi at man er fløjet gennem hullet i toppen af skærmen og næste level loades. I starten havde vi overvejet om vi skulle lave to klasser der arvede fra Entity, hvor den ene klasse kan beskrive en blok, og den anden en platform. Hvis vi havde gjort dette kunne vi nøjes med én EntityContainer der består af både bloks og platforme, vi ville da kunne tjekke typen af den Entity der er stødt ind i. Denne implementation ville dog kræve at vi tjekker typen af det objekt der er stødt ind i, dermed have if-statements der tjekker typen, dette ville være et brud på Open/Close princippet [1]. Til excercise 10 har vi alligevel valgt at indføre en platform class da vi er nødsaget til at kunne identificere hvilken platform vi lander på når vi flyver med en customer. Vores oprindelige design med de tre EntityContainers fungerer dog stadig, da vores platform class arver fra Entity, og vi derfor også kan komme disse ind i en EntityContainer. Det samme gælder for Customers, disse arver også fra Entity.

5.4 Fysik

En væsentlig del af SpaceTaxi er fysikken i spillet. Denne har vi valgt at integrere med spilleren da vi ved at der ikke vil være andre objekter som skal benytte den. Vi vælger altså at se fysikken i spillet som den måde spilleren skal bevæge sig på, og der er derfor stadig player objektets ansvar at kunne bevæge sig korrekt.
Vi har implementeret tyngdekraften således at vi kan tænde og slukke den. Dette skyldes at når spilleren lander på en platform, så ønsker vi ikke at den skal falde igennem platformen. Derfor slukker vi for tyngdekraften og spillerens fart sættes til nul (forudsat at den ikke dør på grund af en for hård landing eller dårlig vinkel). Tyngdekraften tændes dernæst først igen når rumskibet letter fra platformen. Vi oplever dog at rumskibet kan falde igennem platforme hvis den laver små hop. Vi mistænker dette skyldes at der ikke bliver foretaget collision detection hvis rumskibet bevæger sig meget langsomt, men det er ikke lykkedes os verificere dette.

5.5 Customers

Vores timedevents bliver processed i GameRunning hvilket vi har valgt for at gøre det enklere at spawne customers hvilket vil fremstå om lidt. En customer er en instans af Customer, og indeholder information om den enkelte customer. Den information i LevelsKeeper om hver enkel customer bliver sendt igennem CustomerTranslator som sørger for at vi kan instantiere en customer instans.

[1] Kilde: SU18-B4-03-Design_Patterns, undervisningsmateriale

Vores customers bliver ikke spawnet med det samme spillet startes, og de gemmes derfor i et field i GameRunning. Når der bliver oprettet en customer instans, så bliver der også lavet en timedEvent således at denne customer på et tidspunkt bliver spawnet. Når Denne event triggers, så hives den specifikke customer ned i EntityContainer, som bliver renderet, og på denne måde sørger vi for at vores customers først bliver renderet når de spawnes.
Gennem collisiondetection kan vi finde ud af om en customer bliver ramt af taxien, og hvis det er tilfældet, så sættes et field i denne customer til tidspunktet for kollisionen, og den fjernes fra den EntityContainer som render customers. Når taxien senere kollidere med den korrekte platform så udregner den hentede customer hvor mange points spilleren skal have baseret på hvor lang tid spilleren har været om at afsætte denne customer. Disse points bliver lagt til de allerede eksisterende points som er gemt i GameRunning, og GameRunning kan blot render noget tekst for at indikere pointsummen.

5.6 Point

Der ønskes et point-system, som belønner spilleren hvis vedkommende formår at sætte sin passager af til tiden og samtidigt straffer spilleren hvis vedkommende ikke formår at opfylde dette mål. Til dette er der blevet implementeret et pointsystem, hvor antallet af point som spilleren modtager bliver beregnet i en public method kaldet "CaluclatePoints()", som befinde sig i customerclassen. Denne method er essentielt set et matematisk udtryk, som beregner den procentmæssige forskel mellem den tid, det tager for spilleren at transportere kunden og den mængde tid, spilleren har for at få maksimalt point. Denne procentmæssige forskel fratrækkes så fra de antal point, der er at vinde ved at sætte kunden af i tide. Dette antal point, hvad end positivt eller negativt bliver så af metoden ”CalculatePoints()” returneret og field’et ”points” indeni GameRunning bliver dermed inkrimeret med denne værdi.
<<<<< HEAD
=======

5.7 GamePaused

Vi har lavet en funktion der gør man kan pause spillet, men det har ikke været muligt at pause den timer der holder øje med hvornår Customers skal spawne og hvor længe de har været i taxien. Dette skyldes at den timer der er er statisk, og den bruges både til at holde styr på timed events og til at holde øje med hvornår der skal tegnes en ny frame. Det vil sige at man ikke kan sætte timed events på pause uden at stoppe hele flowet i programmet,og det vil derfor ikke være muligt at starte spillet igen. Så hvis man pauser spillet vil tiden stadig gå i spillet og kunderne kan derfor stadig blive ”utålmodige” mens spillet er pauset. Vi processerer dog kun timed events så længe at spillet kører, det betyder at hvis man har spillet pauset på det tidspunkt hvor kunden burde spawnes, vil denne blot spawne idet man starter spillet igen. >>>>> e59f612e85d148144ea7d3bdc3ab391caa60d0f6

6 Vejledning

Når vores Space-Taxi spil startes er den første skærm menuen. Herfra kan et nyt spil startes, et level vælges eller programmet afsluttes. Valgmulighederne kan vælges ved brug af piletasterne og det valgte menupunkt er fremhævet med rød skrift. For at vælge et menu punkt klikker man på 'enter'. Ønsker man at vælge sit start level, så markerer man level menupunktet og klikker 'enter', dernæst vil det valgte level blive ændret og man kan starte spillet. Når spillet er startet styrer man taxien med piletasterne. Målet med spillet er at hente og aflevere passagerer indenfor en given tid. Er man længere tid om at aflevere passageren vil man miste points proportionelt med den tid længere man er om at aflevere dem end tilladt. Spillet tabes hvis man støder ind i en blok eller lander for hårdt på en platform. Fra 'Game Lost' skærmen skal man klikke enter for at komme tilbage menuen. Spillet kan pauses undervejs ved at klikke 'escape'. Spillet kan endnu ikke vindes som beskrevet tidligere, og det vil bare loope til det afsluttes.

7 Evaluering

Vi har til dette projekt lavet tests løbende for at sikre os at koden har den intenderede funktion. Disse tests består af unit tests og integration tests. Vores unit test består i at teste at File Reader kan læse en fil og konvertere denne til et objekt af typen Level. Dette testes på to forskellige filer der har forskelligt antal platforme, kunder og forhindringer, i og med denne passer uden fejl regner vi med at fil til Level objekt fungerer som den skal.
Vores integration test består i at teste hele kommunikationskæden fra LevelLoader til LevelsKeeper. Testen starter med at kalde LevelLoader og få denne til at loade alle levels og derefter tjekker vi LevelsKeeper for at se om denne indeholder de forventede Level-objekter. Testen indeholder på den måde både LevelLoader, FileReader, Level klassen, og LevelsKeeper. Da denne test også passer kan vi regne med at hele denne kæde fungerer som den skal.
I første aflevering i blok 4 havde vi et problem som gjorde det umuligt at teste vores implementationer hvori der indgik Images fra DIKU-arcade. I denne aflevering har vi fundet en løsning til nogle af disse problemer, vi har nemlig lavet en Mock up af IBaseImage. Det der gjorde vi ikke kunne teste implementationen sidst var at Image havde en Render-method som krævede at der var et Game-window åbent, hvilket vi gerne vil undgå til vores tests. Vi har derfor lavet metoden CreateEntity om så denne tager et objekt der implementerer IBaseImage og da bruger den dette til at sætte på den entity den returnerer. Det betyder at vi i vores tests kan give den et MockUpImage, som implementerer IBaseImage, som argument, mens vi i selve spillet giver den et rigtigt Image. Dette betyder også at klassen EntityCreator ikke længere er dirrekte afhængig af Image. Vi har på den måde lavet vores kode mere abstrakt. Det betyder at hvis vi eksempelvis ville have nogle blokke der brugte ImageStrides i stedet for blot images, kunne vi gøre dette uden at skulle ændre i klassen EntityCreator.
Det har dog ikke været ikke været muligt at bruge denne strategi alle steder ek-

sempelvis i Player-klassen. Inden Exercise 9 fik vi at vide at der ville blive udarbejdet en måde at lave et vindue der kan bruges til tests. Dette har vi også fået at vide er lavet, men vi har ikke kunnet få dette til at virke. Når vi kalder metoden "DIKUArcade.Window.CreateOpenGLContext" får vi en NullRefrenceException inde i OpenGL-frameworket. Det har derfor ikke været multigt at lave udtømmene tests af Player, states og CustomerTranslator. Vi har valgt at ændre lidt i Player i forhold til sidste aflevering for at kunne teste blot nogle af dennes funktioner. Det vi har gjort er at sørge for at billederne der er funbundne Player ikke loades når contructoren kaldes, de loades derimod i en seperat metode i Player kaldet "SetImages". Det vil sige at vi i tests kan instansiere en Player uden at loade images i tests, hvorimod vi i implementationen af spillet kan kalde contructoren og da SetImages. Det er selvfølgelig suboptimalt at der skal kaldes 2 metoder for at instansiere Player, men til gengæld kan klassen testses.

7.1 Testplan

Til de tests af de klasser der instansiere tekst og billeder skal man oprette en OpenGL kontekst, ellers vil testene fejle. Til dette er der en metode i DIKUArcade som også laver et vindue, som gør det muligt at teste disse klasser. Det har dog kun været muligt for os at køre disse tests i MonoDevelop, og ikke JetBrains Rider, som resten af vores kode er skrevet i.

7.1.1 Player

1. Test at tyngdekraften får Player til at skifte hastighed. For at kunne gøre dette skal der instansieres et player objekt hvorefter der skal berenges én frame, og da testes det om hastigheden i y-retningen er mindre end 0, og at hastigheden i x-retningen er lig med 0.

2. Test at kraften på Player er 0 i begge retninger ved instansiering.

3. Test at Players boostere starter ved korekte events. Denne skal oprette en PlayerEvent der siger at Player skal tænde en bestemt booster, hvorefter Players ProcessEvent bliver kaldt med dette event som argument. Derefter tjekkes det at Players force er som forventet i begge retninger. Så sendes et event til Player om at slukke for den pågældende motor, og det tjekkes at force nu er 0 i begge retninger.

4. Tjek at Player bevæger sig når force ikke er 0. Her sættes Players force til noget bestemt hvorefter det testses om spilleren har rykket sig i den forventede retning.

7.1.2 EntityCreator

1. Test at position er som forventet for den returnerede Entity. Data til denne test er at man giver EntityCreator et linjenummer og kolonnenummer og tjekker at den returnerede Entity har position som forventet. Det er som nævnt lykkes os at lave denne ved at give den et MockUpImage.

7.1.3 FileReader

1. Test at et kald til FileReader returnerer et level-objekt med de forventede attributer. Hver attribut testes i en test for sig, for at det er nemt at se hvilken del der fejler.

7.1.4 LevelCreator

1. En test der givet et level-nummer tjekker at de EntityContainers der returnerer indeholder de rigtige elementer.

7.1.5 StateMachine

1. Test at det aktive state er MainMenu, ved instansiering af StateMachine.

7.1.6 MainMenu

1. Test at det valgte level skifter når der trykkes ENTER mens knappen "Selected Level" er aktiv.

7.1.7 Customer

1. Test at man får færre end maksimum points hvis man afleverer en kunde for sent.
2. Test at man får maksimum points, hvis man afleverer en kunde til tiden.

7.1.8 Integrationstests

1. Test kommunikationskæden fra LevelLoader til LevelsKeeper, ved at få LevelLoader til at indlæse levels, og tjek derefter om der er de korrekte levels i LevelsKeeper.

Vi har valgt ikke at teste de ting der har med kollision at gøre da vi regner med at dette er testet i DIKUArcade. Det vil sige vi ikke tester det at man samler en kunde op, eller at spillet slutter når man kolliderer med blokke. Vi har heller ikke testet skiftet mellem states da vi mener at dette mere vil være en test af DIKUArcades event handeling. Alle disse tests passer.

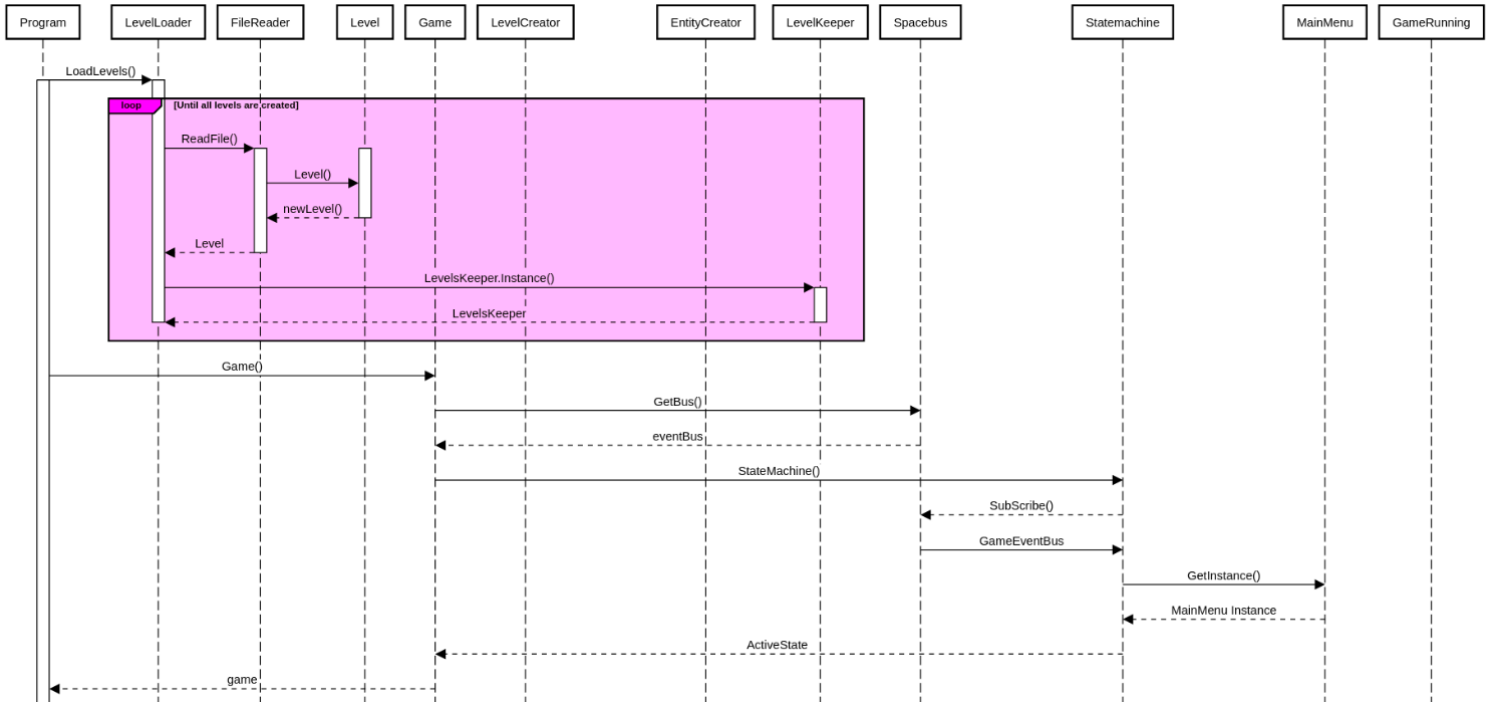
8 Konklusion

Vi kan konkludere at vores program kører som forventet og at vi følger design principperne 'SRP', 'LSP' mens 'OCP' og 'ISP' ikke bliver fulgt. Både de unit tests som vi kan udføre og den integrationstest som vi har viser også at vores program er robust og formår at videregive informationer korrekt. Selvom vores kommunikationskæde kunne ønskes kortere, så opvejer den nemmere maintainablity hurtigt dette. Vi kan også konkludere at vores program på nogle punkter kan udvides og integreres let, mens på andre er det mere vanskeligt.

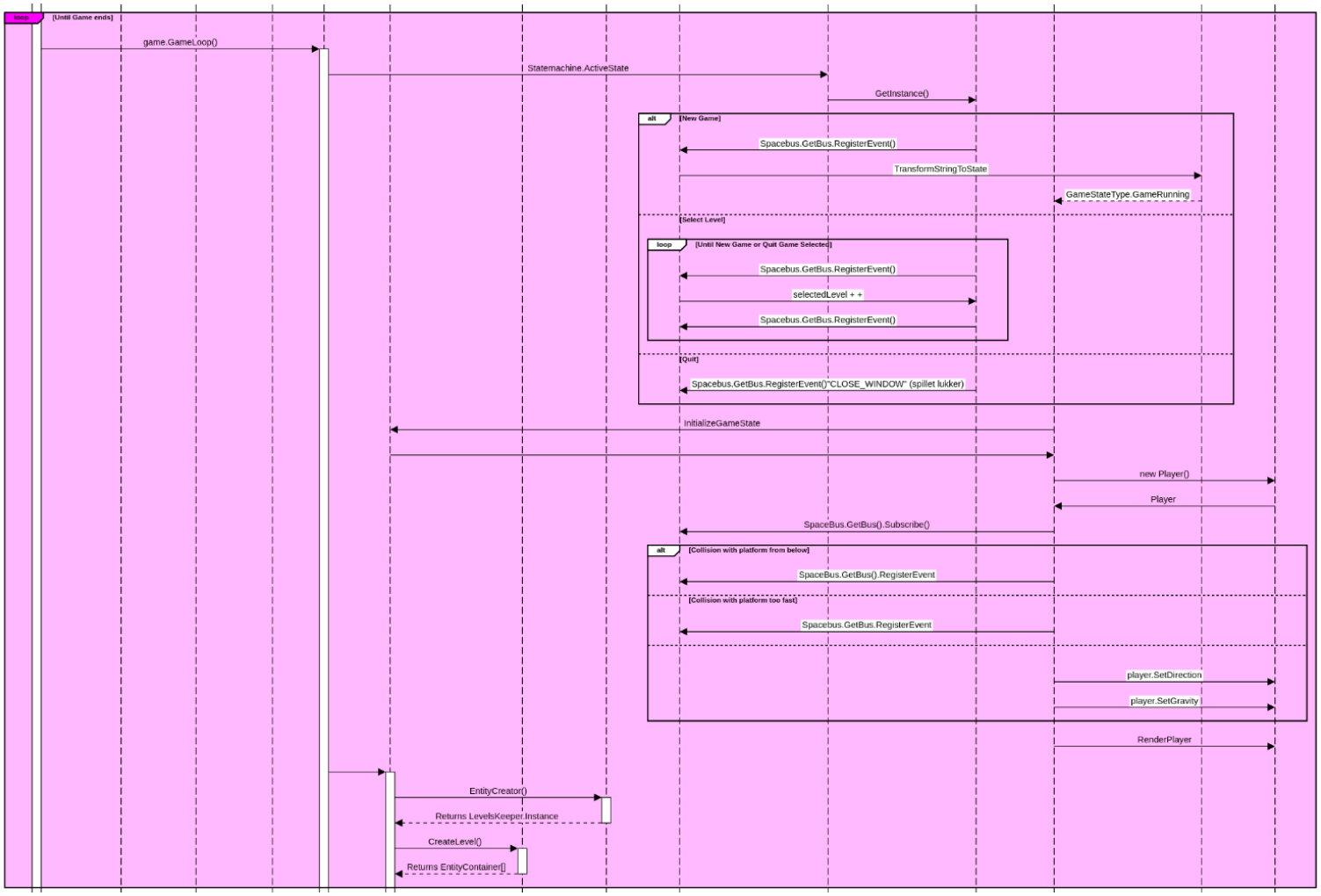
Vi kan også konkludere på vores testafsnit, at de tests vi har implementeret ikke er tilstrækkelige til at få et helhedsoverblik, men at vi med vores testplan har et overblik over hvilke tests der burde laves af vores program.

9 Billag

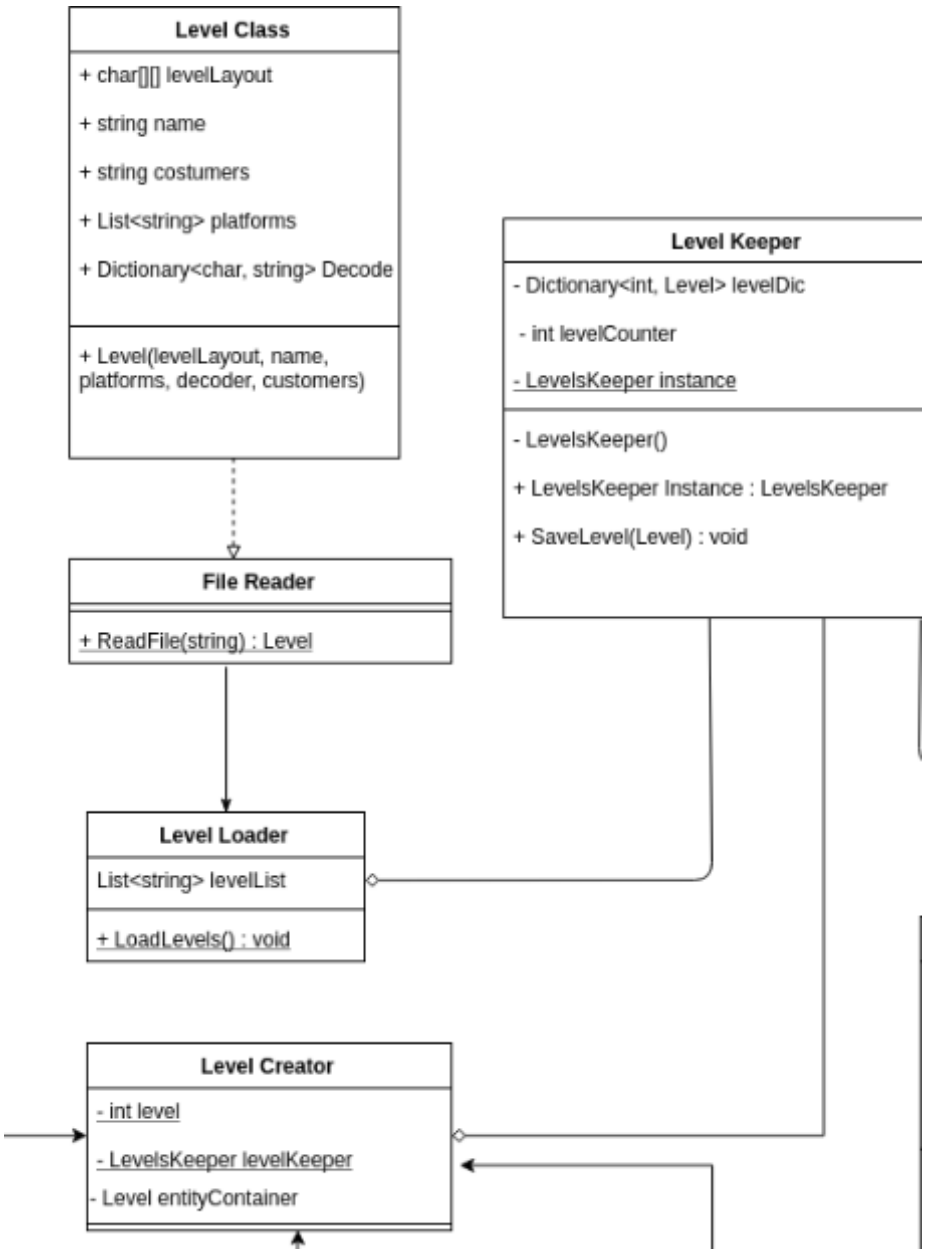
9.1 Billag 1 - Sequence Diagram 1



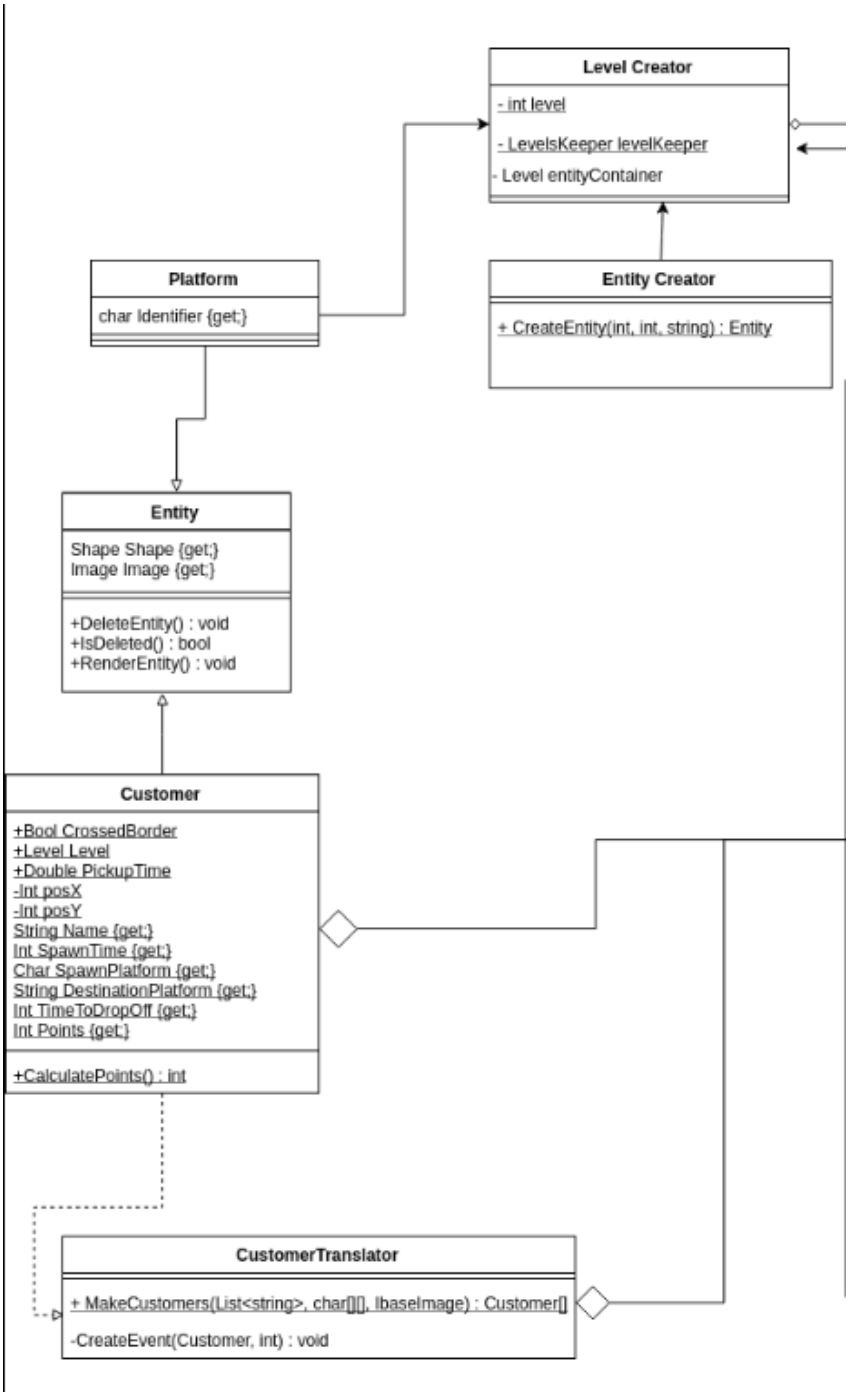
9.2 Billag 2 - Sequence Diagram 2



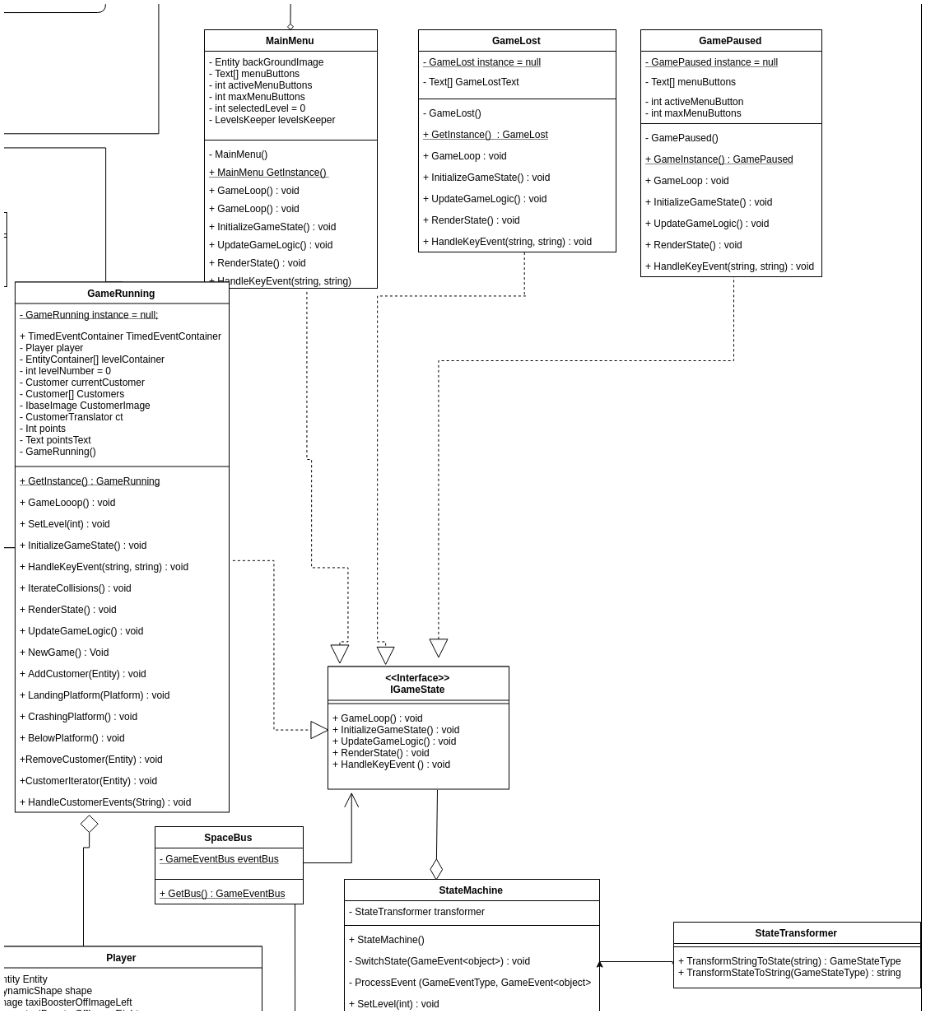
9.3 Billag 3 - Level Parsing



9.4 Billag 4 - Customer



9.5 Billag 5 - States



9.6 Billag 5 - Game