Exercise 8

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 1;
Due: Tuesday, May 1st

1 Indledning

Spacetaxi er et gammelt spil med formål at flyve passagere fra en destination til den anden i det ydre rum. 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. Spillet er endnu ikke spilbart, men vil i løbet af de kommende afleveringer blive en mere og mere fyldestgørende implementation af spillet.

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

2 Baggrund

Til dette projekt vil vi benytte DIKUArcade som game engine. Denne har mange funktioner som vil hjælpe os med implementationen, både nu og i fremtiden. Vores indfaldsvinkel på vores design vil være at sørge for at hver class i vores program kun vil have et ansvar, og at klassen selv kan udføre opgaverne som skal til for at klare ansvaret. Ved at skabe mange klasser som kun har et enkelt ansvar vil vi få kode som let kan udvides, og som let kan bygges videre på, samtidigt med at det vil være nemt at finde bugs og maintaine i fremtiden.

3 Analyse

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

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 9 koncepter hvoraf 8 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.
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

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

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

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
StateMachine til GameStates	Giver inputevents videre	Kommunikerer information
SpaceBus og GameStates	GameStates registrerer events til SpaceBus	Kommunikerer information
SpaceBus og StateMachine	SpaceBus kommunikerer input events	Kommunikerer information

Table 2: Associationer mellem forskellige concepter

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 tabel 3 ses det hvilke attributer vi tænker at hvert concept skal have.
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.

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	

Table 3: Attributer for hvert koncept

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.

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

	LL	FR	LCl	LCr	EC	LK	GameState	StateMachine	SpaceBus
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

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 en liste 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 listen 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, men da vi ikke har lørt om dette design pattern kan vi kun beskrive det som en form for implementation af et Mediator pattern som sørger for kommunikationen mellem vores game states.

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.

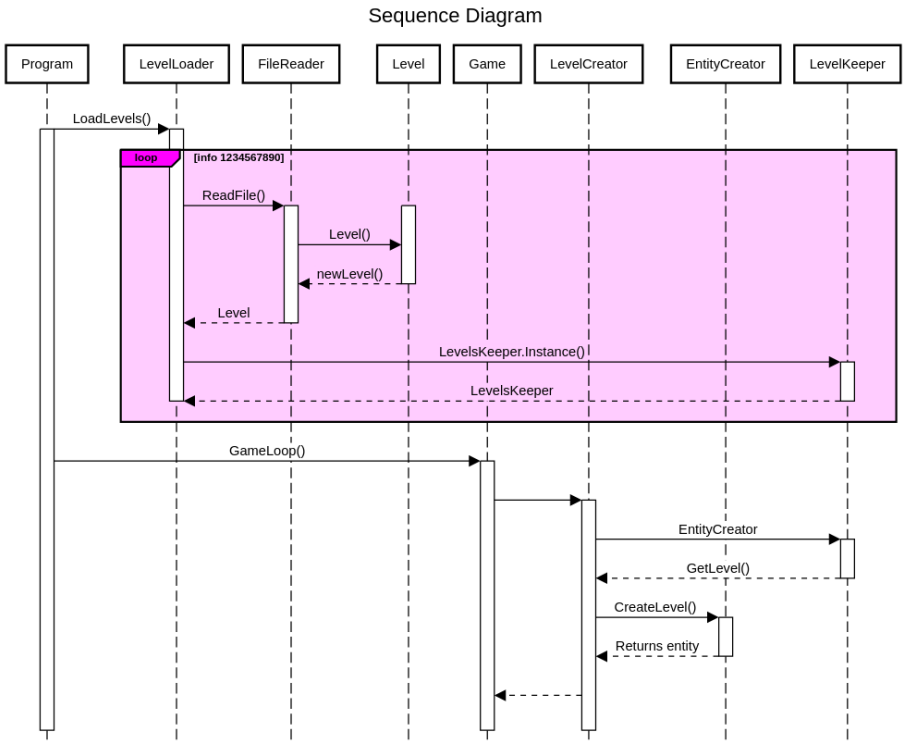
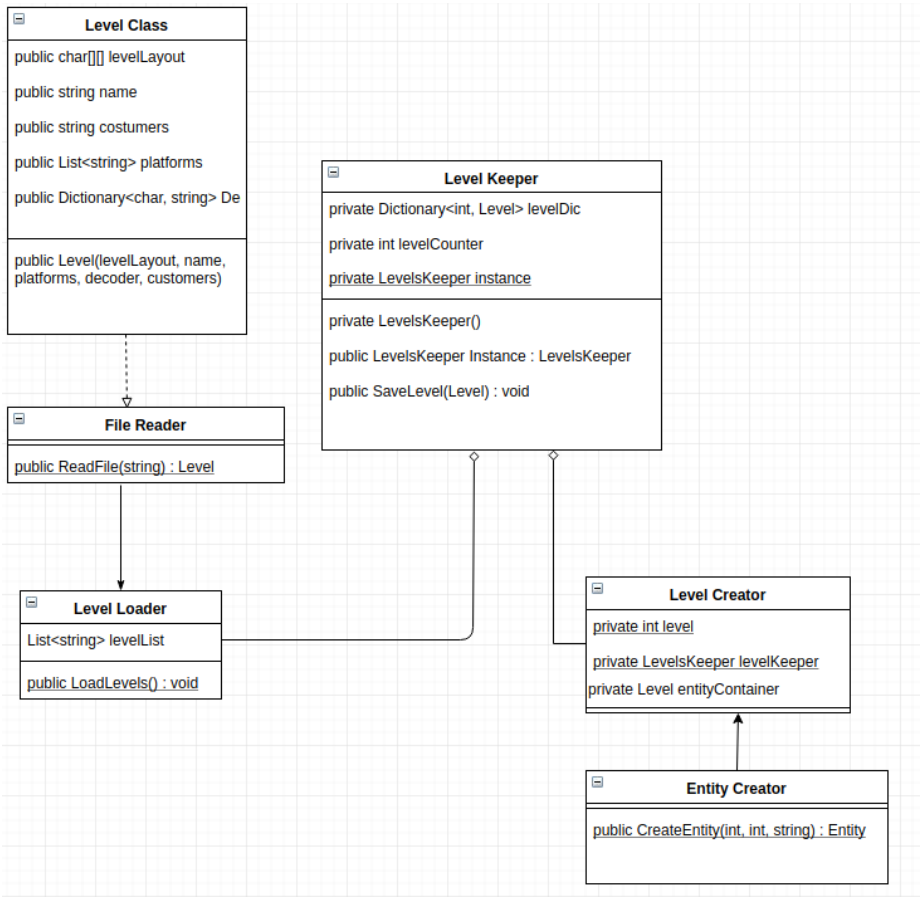


Figure 1: Sequence diagram

5 Implementation



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.

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 EntityContainers, 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].

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.

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

6 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 sidste aflevering 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, eksempelvis i Player-klassen. Denne loader en masse billeder som den kan vise, og denne kan derfor heller ikke instansieres i tests. Vi kunne godt lave Player-klassen om, så den tog et array af IBaseImage som argument, og da brugte disse, men dette ville kræve større ændringer i implementationen af både GameRunning og Player. Vi har derfor valgt ikke at gøre dette, men i stedet vente på vi får en officiel metode til at teste dele af DIKUArcade, hvilket vi har fået at vide der bliver arbejdet på på nuværende tidspunkt.

7 Konklusion

Vi kan konkludere at vores program kører som forventet og at vi 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.