

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. 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 DIKU Arcade 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.

Responsibility description	Concept name
Læser fil og returnerer Level objekt med info herfra	File reader (FR)
Class til at beskrive Level objekter	Level class (LCI)
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 creator (LCr)
Returnerer en entity som repræsenterer blok i spillet	Entity creator (EC)

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

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 6 koncepter hvoraf 5 er Do- og 1 er know-koncepter.

Vores entity creator laver for nu kun firkanter der alle er lige store, og som har et billede, da vi antager vi ikke skal implementere collision detection i denne uge. På længere sigt skal der være forskel på størrelsen og formen på de forskellige objekter.

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

Table 2: Associationer mellem forskellige koncepter

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 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	Container 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	

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.

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. Tilgængæld 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 netop mapper hvert af vores koncepter med en implementeret class, hvilket betyder at hver af vores klasser kun har ét ansvar. Mængden af klasser kan være en indikation på at kommunikationskæden er lang, mens det at der ikke er nogen koncepter som mapper til mere end en klasse fortæller os at arbejdsbyrden er jævnt fordelt.

	Program	LL	FR	LCI	Game	LCr	EC	LK
Program	x							
LevelLoader		x						
FileReader			x					
Level class				x				
Game					x			
LevelCreator						x		
EntityCreator							x	
LevelKeeper								x

Ud fra vores sequence diagram, som ses på figur 1, kan det ses at vores design arbejder af to 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.

Den anden del bliver udgjort af Game som beder LevelCreator om en Entity-Container 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 Entity-Container, og når alle blokke er blevet bygget returneres EntityContaineren og Game kan derefter render den.

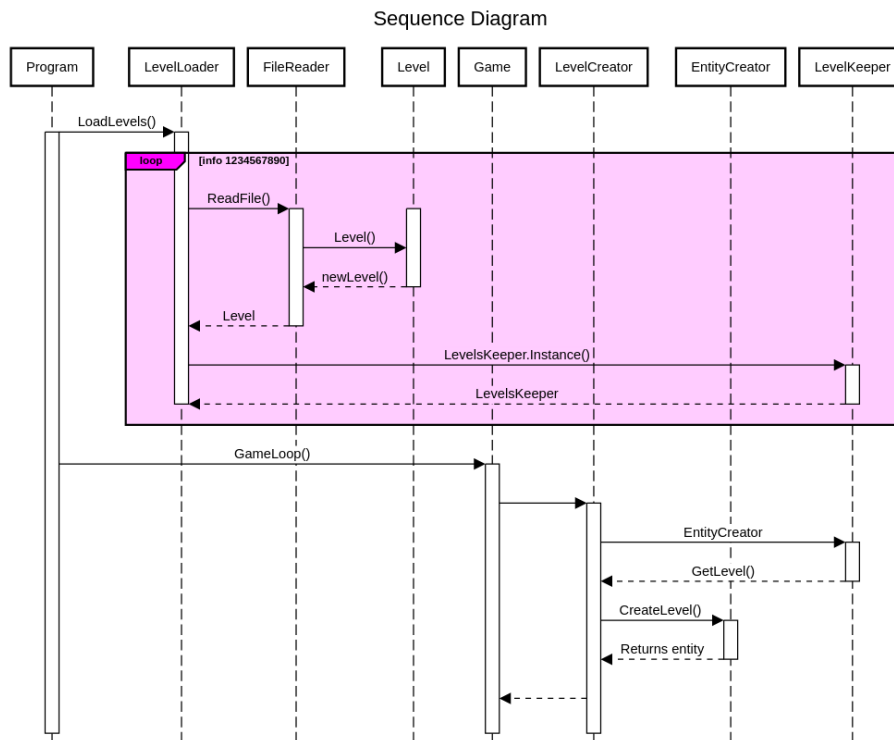
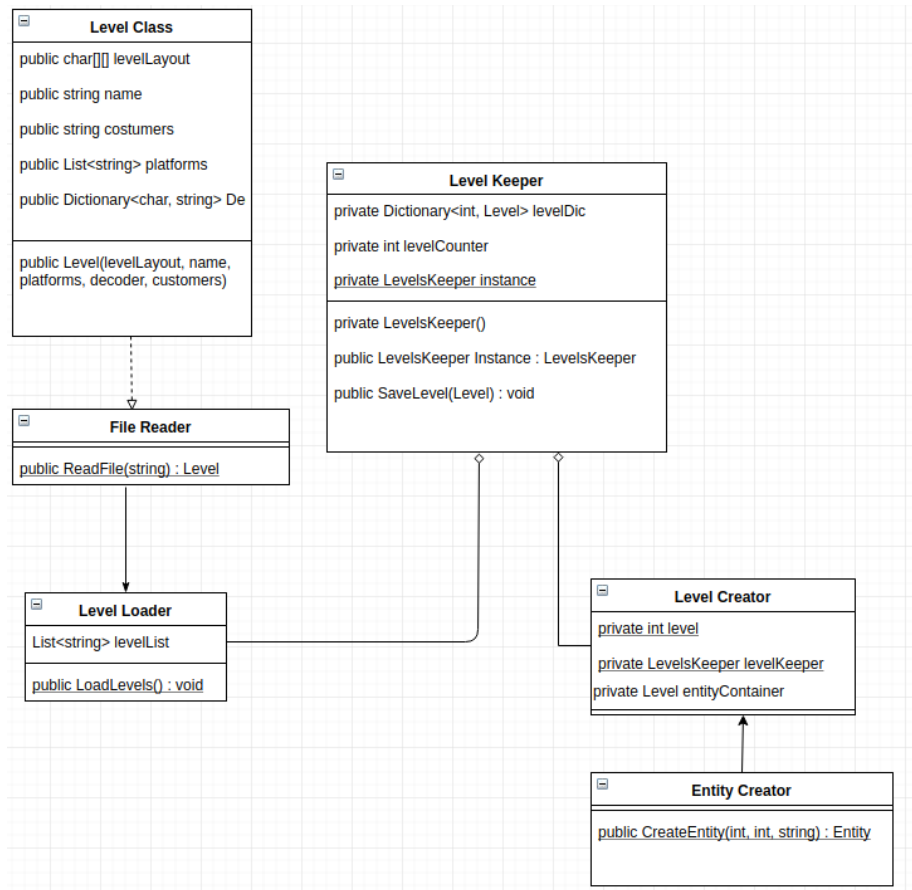


Figure 1: Sequence diagram

5 Implementation



Som det ses på vores class diagram så er de fleste af vores klasser statiske. Da vi ikke skal bruge flere instanser af for eksempel **LevelLoader** så vi det mere intuitivt at holde **Program** og **Game** fri for objekter som kun skal benyttes en eller få gange, og i stedet bare lade dem kalde statiske metoder i de statiske klasser. Vi vil dog gerne ændre formindske antallet af statiske klasser til inden deadline for eksamen.

Undtagelser på det ovenstående er vores **Level** class som instatierer **Level** objekter. 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.

En anden undtagelse er **LevelsKeeper**. **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 for at arbejde med værdier frem for referencer hvilket bevirker at vi ikke skal bekymre os om hvad der sker hvis der læses og skrives til den samme hukommelse samtidigt. 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 kan dette sikkert implementeres. Alternativt ville vi kunne have lavet LevelsKeeper static, men så ville vi skulle være sikre på at der aldrig ville skulle skrives og læses det samme sted samtidigt.

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 propertyen har ingen setter for at følge princippet om at hver class udfører opgaverne for sit eget ansvar.

5.1 Collision detection

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 princippet¹.

6 Evaluering

Vi har til dette projekt lavet tests løbende for at sikre os at koden har den intendede 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 load alle levels og derefter tjekker vi LevelsKeeper for at se om denne indeholder de forventede Level-objekter. Testen indeholder på den måde både

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

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 direkte 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.

7 Konklusion

Vi kan konkludere at vores program kører som forventet og at vores design mål om at hver klasse kun har et ansvar er opnået. 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 maintainability hurtigt dette. Vi kan også konkludere at vores program nemt kan udvides i forhold til funktionalitet som beskrevet for blandt andet EntityCreator.