Exercise 7

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 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.

2 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)

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 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 objeker.
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 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 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	

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 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 en 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 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	LCl	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 EntityContainer 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 EntityContaineren og Game kan derefter render den.

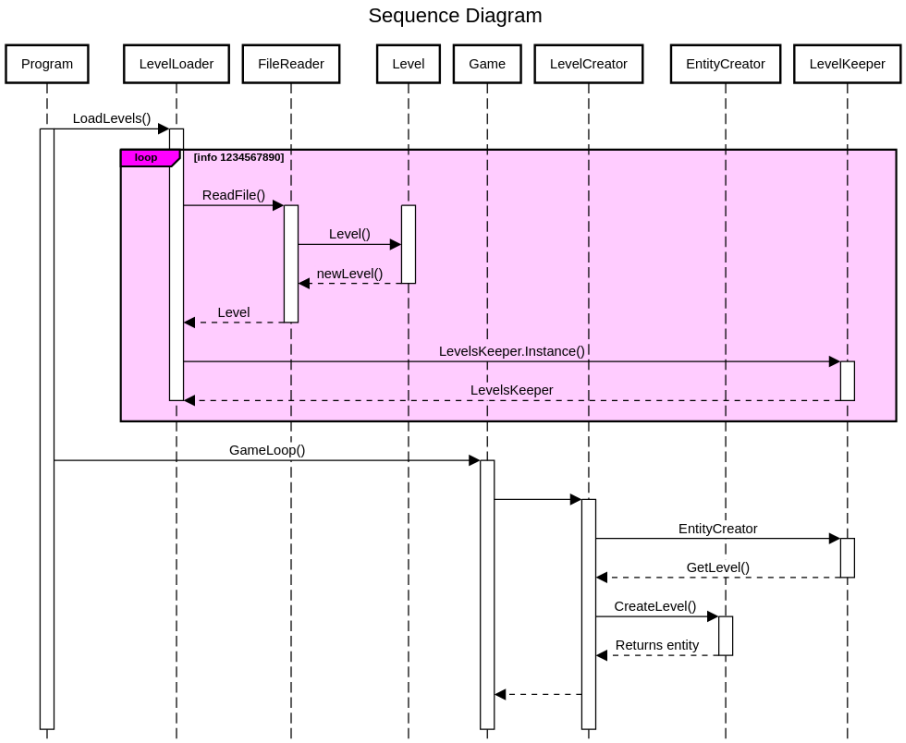
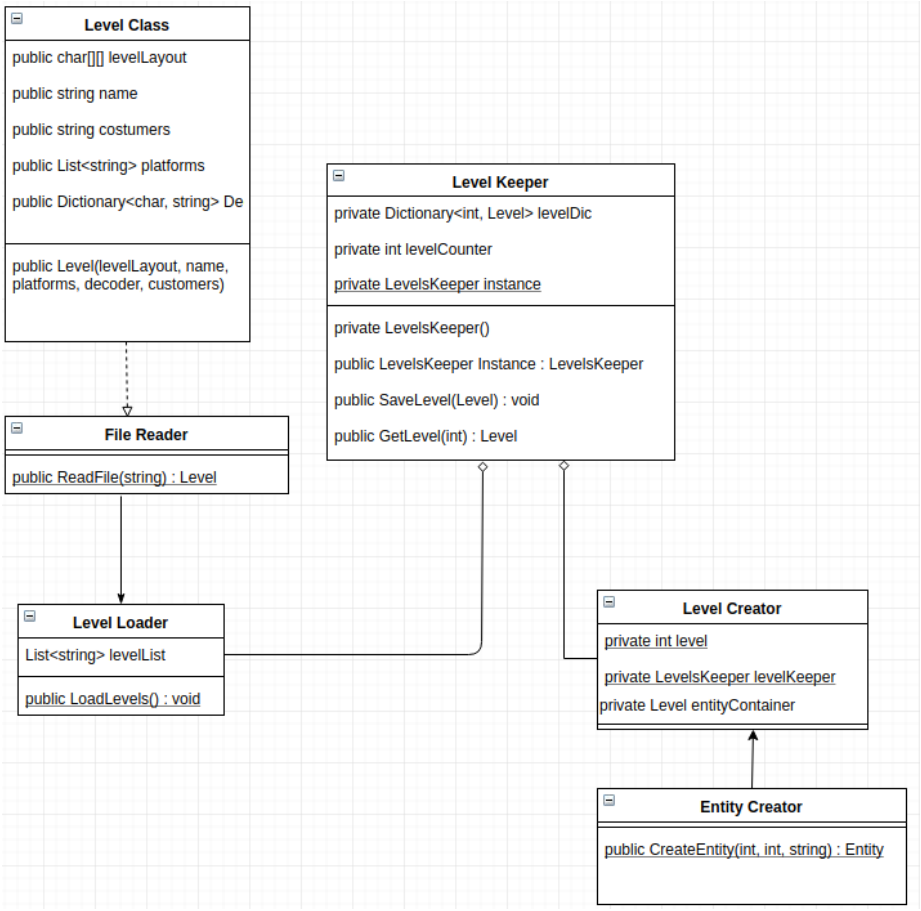


Figure 1: Sequence diagram

4 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.
Undtagelser på det ovenstående er vores Level class som instatierer et nyt Level objekt. 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 propertien har ingen setter for at følge princippet om at hver class udfører opgaverne for sit eget ansvar.

Som nævnt tidligere vil vi i denne uge antage at alle blokke som danner vores level skal være lige store og have de samme egenskaber. Dette sørger EntityCreator for. I fremtiden vil vi formodentlig skulle gøre forskel på blokke og platforme, og sørge for at de kan have forskellig facon. Dette vil vi nemt kunne indføre i vores design ved at implementere en superklasse hvorfra en platforms klasse og en blokklasse kan arve fra. Det vil da blive EntityCreators opgave at enten kalde en klasse som kan returnere en platform, eller en klasse som kan returnere en blok, afhængigt af oplysningerne fra LevelCreator. På denne måde vil vi nemt kunne udføre collission detection og afgøre om vi har ramt en blok eller en platform.

5 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 maintainablity hurtigt dette. Vi kan også konkludere at vores program nemt kan udvides i forhold til funktionalitet som beskrevet for blandt andet EntityCreator.