Projektopgave forår 2016
02362 Projekt i software-udvikling
02327 Indledende databaser og database programmering

Gruppe nr: 4
Afleveringsfrist: Fredag d. 13/5-2016 Kl. 23:59

Denne rapport indeholder 44 sider inkl. denne side.

s144951, Harder, Nikolaj, _____

s145091, Javaid, Arbab, _____

s153959, Larsen, Mathias Rosenlund, _____

s153850, Leth, Adam, _____

	Analyse	Design	Implementation	Test	Rapport	Timer i alt
Nikolaj						
	10	15	15	4	11	55
Arbab						
	10	15	15	4	11	55
Adam						
	10	15	15	4	11	55
Mathias						
	4	6	40	4	7	61
Sum	34	51	85	16	36	226

Indhold

1 Forord og indledning

I dette projekt har vi fået til opgave, at færdiggøre det matadorspil, som vi tidligere har arbejdet med på 1. semester. Kravene bliver denne gang endnu større, og vi skal nu have alle regler og felter med. Desuden skal spillet denne gang også kunne gemmes og hentes, således at man ikke mister noget, hvis man lukker spillet undervejs. Hele projektet bygger på CDIO princippet og følger de principper som vi tidligere har fulgt, herunder blandt andet GRASP. Projektet er, ligesom de tidligere projekter, også testet igennem for eventuelle fejlkilder for, at skabe et stykke software der virker efter hensigterne.

2 Problemstilling og baggrund

2.1 Kravspecifikation

Visionen fra kunden er ikke ligeså kortfattet som den har været i tidligere projekter, men er tilgængeld ret præcis denne gang. Der var derfor ikke særligt mange spørgsmål at stille, og valgene er derfor taget udfra de beslutninger som vi mener er bedst. Vi har fra de tidligere projekter allerede fået svar på visse spørgsmål, og har derfor brugt de samme svar som fra dengang. Blandt andet gav det i CDIO3 ikke mening, at felter af samme type kunne komme lige efter hinanden da vi dengang fik frie hænder til rækkefølgen af felter. Det har vi tilnærmelsesvist også fulgt denne gang, vi har dog valgt, at placere felterne, som de ville have ligget på en fysisk spilleplade.

2.1.1 Funktionelle krav

1. Generelle krav
   (a) Spillet skal kunne spilles af 3-6 personer.
   (b) Spillet skal gå på tur mellem spillerne.
   (c) Spilleren skal kunne lande på et felt og fortsætte derfra næste gang.
      i. Spilleren skal gå i ring på brættet.
      ii. Spilleren indkasserer 4000 når denne passerer start.
   (d) Spillet skal kunne kaste to terninger.
      i. En spiller får en ekstra tur ved at slå 2 ens.
   (e) Spillet skal kunne gemmes.
   (f) Spillet skal kunne hentes.
   (g) Spillet skal organisere spillernes pengebeholdning.
      i. En spiller er ude, når hans samlede realiserbare formue rammer 0.

ii. Spillerne starter med en kapital 30.000.
iii. En spiller vinder, når alle andre spilleres kapital er på 0.

2. Krav til spillepladens felter

(a) Territory:
i. Territory felter skal kunne købes af spillerne.
ii. Når en spiller lander på et Territory felt, som er ejet af en anden spiller, skal han betale en forudbestemt leje.
iii. Ejeren af et "sæt"af Territory felter skal kunne bygge huse og hoteller.
iv. Spillere skal kunne sælge huse og hoteller.
v. Lejen på Territory skal være afhængig af om der er bygget ejendomme på grunden.

(b) Laborcamp:
i. Laborcamp felter skal kunne købes af spillerne.
ii. Når en spiller lander på et Laborcamp felt, som er ejet af en anden, skal den aktive spiller slå med terningerne og gange summen med 100 og med antallet af Laborcamps ejet af modspilleren. Resultatet af dette skal betales til ejeren.

(c) Fleet:
i. Fleet felter skal kunne købes af spillerne.
ii. Når en spiller lander på et Fleet felt ejet af en modspiller, skal han betale leje. Denne stiger med antal Fleet felter, som modspilleren ejer.

(d) Tax:
i. Tax felter skal ikke kunne købes.
ii. Når en spiller lander på et Tax felt, skal han betale en afgift.
A. På det ene Tax felt skal spilleren kunne vælge imellem at betale 4000 eller 10 % af de samlede aktiver.

(e) Refuge:
i. Refuge felter skal ikke kunne købes.
ii. Når en spiller lander på et Refuge felt får han et forudbestemt beløb.

(f) Prøv Lykken:
i. Når en spiller lander på Drawcard skal det næste kort i bunken trækkes.
ii. Enkelte korttyper skal kunne gemmes til senere brug i spillet.
iii. Et gemt kort skal kunne bruges ved spillerens tur.

(g) En spiller skal kunne ende i Fængsel ved at

i. Lande på felt 11 (Gå i fængsel)
ii. Trække et gå-i-fængsel kort

(h) En spiller kan komme ud af fængslet ved at
i. Bruge et benådningskort fra Prøv lykken-bunken.
ii. Betale 1000 kr. inden spilleren kaster terningerne. Spilleren kan rykke ud af fængslet runden efter.
iii. En spiller kommer ud af fængslet ved at slå to ens. Dvs. to 1'ere, to 2'ere, to 3'ere, osv.
iv. En spiller kan maks sidde i fængsel i tre omgange. Efter at have siddet i fængsel i 3 omgange, trækkes kr. 1000, hvorefter spilleren rykker antal øjne på terningen.

2.1.2 Non-funktionelle krav

1. Softwaren skal kodes i Java
2. Spillet skal kunne hente nødvendige informationer fra databasen.
3. Eclipse skal bruges som udviklingsmiljø.
4. Softwaren skal kunne køres på DTU's databar maskiner.

3 Analyse

Analyse er det første skridt imod udviklingen af det færdige produkt. Der laves herunder en masse analyseringer ud fra den vision som vi har fået udleveret fra kunden. Ud fra disse analyseres der for krav, navneord samt udsagnsord. På denne måde kan vi komme frem til en kravspecifikation, aktører, og usecases. Dette gør os herefter i stand til at opstille en domænemodel, som vi kan bruge til, at videreudvikle softwaren.

3.1 Funktionel afhængighed

Begrebet funktionel afhængighed bruges hvis man ud fra en eller flere attributter kan bestemme en anden attribut i tabellen. Funktionel afhængighed kan skrives på følgende form.

A –> A dvs. A bestemmer B / B er funktionelt afhængig af A

A –> B dvs. A og B bestemmer C / C er funktionelt afhængig af A og B

3.2 Normaliseringsanalyse

Når man skal designe sin database er der flere faktorer man skal tage med i sine overvejelser. Det er f.eks. en dårlig idé at have attributter der indeholder mere end en enkelt værdi. Desuden bør man ikke gemme samme information flere forskellige og unødvendige steder i databasen. En måde man kan undgå disse negative egenskaber i sin database er ved, at anvende en metode som kaldes normalisering. Essensen ved benyttelse af normalisering er, at undgå gentagelse af information (redundans) samt, at det skal være nemt, at hente oplysninger fra tabellerne. Normalisering er en proces bestående af flere dele og indebærer, at man bringer sine tabeller på normalform. Der er udviklet mange varianter af normalformer, men de er dog hver især forudsætning for hinanden. Dvs. at hvis man vil bringe en tabel på 2. normalform skal den i forvejen være på 1. normalform. Ved implementering af 3. normalform kræves det at tabellen allerede er på 1. og 2. normalform. Vi vil i dette afsnit dække de tre første normalformer.

3.2.1 Første normalform

Det specielle ved en database på 1. normalform er, at der skal være defineret en primærnøgle bestående af en eller flere attributter. Hver attribut må kun indeholde atomare værdier. Dvs. at en attribut kun kan have en enkelt værdier. Derudover må der ikke være nogen gentagende grupper af kolonner - hver attribut skal have et unikt navn.

3.2.2 Anden normalform

For en database på 2. normalform er forudsætningen, at den allerede er på 1. normalform. Derudover må en attribut der ikke er en primærnøgle eller del af en sammensat primærnøgle ikke være delvist afhængig af selve primærnøglen.

3.2.3 Tredje normalform

En database på 3. normalform skal naturligvis være på 2. normalform og må ikke indeholde transitive funktionelle afhængigheder. Dvs. at en attribut ikke må være funktionel afhængig af primærnøglen via en anden attribut der ikke er primærnøglen eller del af primærnøglen.

3.3 Usecases

Dette projekt er til dels udarbejdet efter metoderne i Unified Process. Efter at have udarbejdet kravspecifikationen, følger det at man laver usecases for at uddelegere ansvar til de forskellige dele af spillet. Opgaven er at videreudvikle spillet fra tidligere med de partielle dele som blev udviklet i sidste semester. Derfor er det også naturligt, at arbejde videre på usecases fra det

tidligere projekt. Samtidig er spillet blevet mere kompliceret og skal indeholde flere funktioner, hvilket har resulteret i ca. dobbelt så mange usecases som tidligere.
Vi skal bl.a. inkorporere flere avancerede funktioner til felterne såsom køb af aktiver og pantsætning. Dette har krævet flere usecases end tidligere. Opsætningen og sammenhængen mellem de forskellige usecases er vist i usecasediagrammet (10.1.). Her ses det, at usecase U1 er helt uden for spillet, hvilket skyldes, at den bruges til, at oprette et nyt spil med tilhørende database, sam tildeler spillerne navne og start balance.

Use case:	Start spillet
ID:	U1
Beskrivelse:	Bruger skal starte spillet for at kunne spille.
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Computer skal være tændt og spil gemt på den.
Main flow:	1. Bruger skal indtaste navn på spillere 2-6.
	2. Der besluttes hvem der starter.
	3. Alle spillere tildeles 30.000 kr. til deres konto.
	4. Der oprettes en database med spillets data
Postcondtions:	Der er oprettet databaser til spiller og spillets mainflow er startet
Alternative flows:	

Figur 3.1: Usecase U1

U2 Spil - Spillet beskriver hvorledes spillets main flow fungerer. Spillet bliver ved med, at skifte tur mellem spillerne. De alternative flows beskriver hvilke hændelsesmuligheder der er i løbet af den enkelte spillers tur. Når et alternativt flow er afsluttet, gives turen videre. Dette fortsætter indtil spillet afsluttes eller, at der er fundet en vinder ved, at alle undtagen én er gået bankerot.

Use case:	Spil spillet
ID:	U2
Beskrivelse:	Bruger skal spille spillet - Main flow i spillet
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spillet skal være startet
Main flow:	1. Så længe der ikke er fundet en vinder så. 1.1 Spillet viser hvem den aktive spiller er. 1.2 Spiller ruller terningerne. 1.3 Spillet summerer de to terninger. 1.4 Spiller rykker summen af terningerne i urets retning. 1.5 Spiller lander på felt 1.6 Næste spillers tur. 2. Er der en vinder. 2.1. Navnet på vinderen vises. 2.2 Bruger vælger om spillet skal genstartes eller lukkes. 2.3 Hvis bruger vælger omspil start forfra ved punkt 2
Postcondtions:	Spil skal være lukket eller være startet forfra.
Alternative flows:	1.1.1 U18 slipUdAfFængsel 1.2.1 U17 slåDobbeltSlag 1.5.1 U2: landPåTaxFelt. 1.5.2 U3: landPåLedigtFelt. 1.5.3 U4: landPåRefugeFelt. 1.5.4 U5: landPåFleetFelt. 1.5.5 U6: landetPåTeritorry 1.5.6 U7: landPåLaborFelt. 1.5.7 U16: landPåDrawCard 1.7 U9: gemSpil

Figur 3.2: Usecase U2

Usecase U3-U8 er nær identiske med de usecases vi havde i rapporten fra sidste semester. Dermed bidrager de ikke med noget nyt. Vi har derfor valgt, at udelade dem i denne rapport (de kan dog ses under afsnit 11)

Usecase 9 beskriver hvad der sker når et fængselskort bliver trukket fra bunken.

Use case:	Alternativt flow: TrækFængselsKort
ID:	U9
Beskrivelse:	Spilleren trækker et fængselskort
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Der skal være trukket et fængselskort
Main flow:	1. Spilleren rykker til fængselsfeltet 2. Spilleren er fængslet
Postcondtions:	Spilleren er fængslet
Alternative flows:	U17

Figur 3.3: Usecase U9

Hvis en spiller skal betale et beløb, der er større end spillerens samlede realiserbare formue, går spilleren bankerot. I sådan et tilfælde fjernes spilleren fra Gameboardet. Alle huse og hoteller fjernes fra spillerens grunde og disse er ikke længere ejet.

Usecase:	Extension usecase: Bankerot
ID:	U10
Beskrivelse:	Spilleren går bankerot.
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spilleren spilleren skal betale et beløb, han ikke har råd til.
Main flow:	1 Spillerens ejede grunde bliver frigjort. 2 Spilleren er ude af spillet.
Postcondtions:	Spilleren er ikke en del af GameBoardet mere
Alternative flows:	

Figur 3.4: Usecase U10

Spillet gemmes automatisk i databasen, når en runde i spillet er færdig eller der laves ændringer på Gameboardet.

Usecase:	Alternativt flow: gem spil
ID:	U11
Beskrivelse:	Spillet gemmes undervejs
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spillet er startet og navne er indtastet.
Main flow:	1. En spillers tur er færdig eller Gameboardet opdateres. 2 Spillet gemmer stadiet i databasen, så det kan indlæses til senere brug.
Postcondtions:	Spillet er gemt og fortsætter som tidligere.
Alternative flows:	

Figur 3.5: Usecase U11

Når spillet startes har spillerne mulighed for, at hente et gemt spil fra databasen. Hvis der vælges ja, hentes alle data om spillerne, Gameboardet, prøv lykken kort og så fortsætter spillet fra det stadie det var i da det sidst blev gemt.

Use case:	Alternativt flow: Hent spil
ID:	U12
Beskrivelse:	Spillet startes op og et tidligere spil genindlæses.
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spillet skal være startet.
Main flow:	1. Spillet åbnes. 2. Databasenavnet for det gemte spil vælges 3. Spillet indlæses og fortsætter som tidligere.
Postcondtions:	Spillet er hentet og fortsætter nu som hvis det aldrig var blevet lukket.
Alternative flows:	

Figur 3.6: Usecase U12

Når en spiller lander på et felt ejet af spilleren selv og hvorpå der er bygget enten huse eller hoteller, har spilleren mulighed for, at sælge disse aktiver til halv købspris. Hvis spilleren vælger, at sælge fjernes aktiverne så fra Gameboardet og den halve salgspris overføres herefter spillerens konto.

Use case:	Alternativt flow: Sælg aktiver
ID:	U13
Beskrivelse:	Spilleren vælger at sælge i forvejen købte aktiver
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spilleren skal være landet på et i forvejen ejet Ownable og have aktiver på dette
Main flow:	1 Spilleren vælger at sælge aktivet på feltet. 2 Aktivet sælges til banken og beløbet overføres fra bank til spiller.
Postcondtions:	Spilleren har solgt aktivet og spillet fortsætter.
Alternative flows:	

Figur 3.7: Usecase U13

Usecase 14 beskriver hvad der sker når man trækker et tilfældigt kort.

Usecase:	Alternativt flow: landPåLuckyCard
ID:	U14
Beskrivelse:	Spilleren lander på et træk kortet felt.
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spillet skal være startet.
Main flow:	1. Spilleren trækker et tilfældigt kort. 2. Hvis kortet ikke kan gemmes 2.1. Kortet bruges. 3. Hvis kortet kan gemmes 3.1. Kortet gemmes til senere brug.
Postcondtions:	Spillerens konto og getOutOfJailCards opdateres og spillet fortsætter
Alternative flows:	

Figur 3.8: Usecase U14

Usecase 15 beskriver hvad der er af valg for spilleren når han lander på et felt som han selv ejer.

Usecase:	LandPåEjetOwnable
ID:	U15
Beskrivelse:	Spilleren lander på et Territory-felt som spilleren selv ejer.
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spilleren er landet på et Territory-felt, som spilleren ejer.
Main flow:	1.1 U13 - SælgAktiver 1.2 U17 - KøbAktiver 1.3 Spilleren gør intet
Postcondtions:	Spillerens aktiver og balance opdateres
Alternative flows:	

Figur 3.9: Usecase U15

Usecase 16 beskriver de valg man har som spiller når man ender i fængsel.

Usecase	Fængsel
ID:	U16
Beskrivelse:	Spilleren er i fængsel
Primære aktøre	Bruger
Sekundære akt	Ingen
Preconditions:	Det er spillerens tur og spilleren er i fængsel
Main flow:	1.1 Spilleren slår med terningerne
	1.2 Spilleren betaler 1.000 kr.
	1.3 Spilleren bruger getOutOfJailCard
	1.4 Spilleren har været i fængsel 3 ture og tvinges nu til, at betale 1.000 kr.
Postcondtions:	Spilleren kommer ud af fængslet, det er spillerens tur
Alternative flow	1.1.1 Spilleren slår ikke dobbelt, spilleren bliver i fængsel
	1.4.1 U10 Bankerot

Figur 3.10: Usecase U16

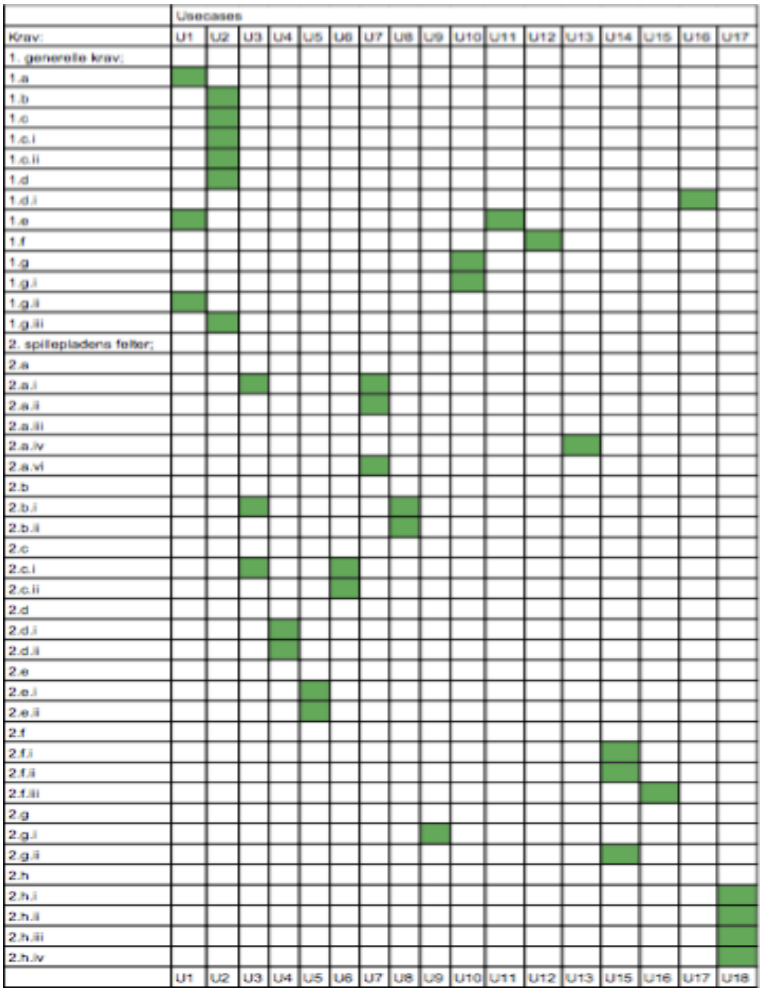
Usecase 17 beskriver hvordan flowet for hvordan man køber aktiver.

Usecase:	Alternativt flow: Køb aktiver
ID:	U17
Beskrivelse:	Spilleren vælger at købe aktiver
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spilleren skal være landet på et i forvejen ejet Ownable
Main flow:	1 Spilleren vælger at købe aktiver
	2: Beløbet trækkes fra spillerens konto og aktivet placeres på feltet
Postcondtions:	Spilleren har købt aktiver og spillet fortsætter
Alternative flows:	2.1: Spilleren har ikke råd til at købe aktiver, spillet fortsætter som ellers

Figur 3.11: Usecase U17

3.4 Kravsporing

Figur 3.14 giver et overblik over hvilke usecases der opfylder kravene til vores program. Som man kan se har vi så vidt muligt forsøgt at få uddelegeret ansvaret for opfyldelse af de forskellige krav på de forskellige use cases. Der er arbejdet henimod, at opfylde målene i GRASP-principperne om fordeling af ansvar.

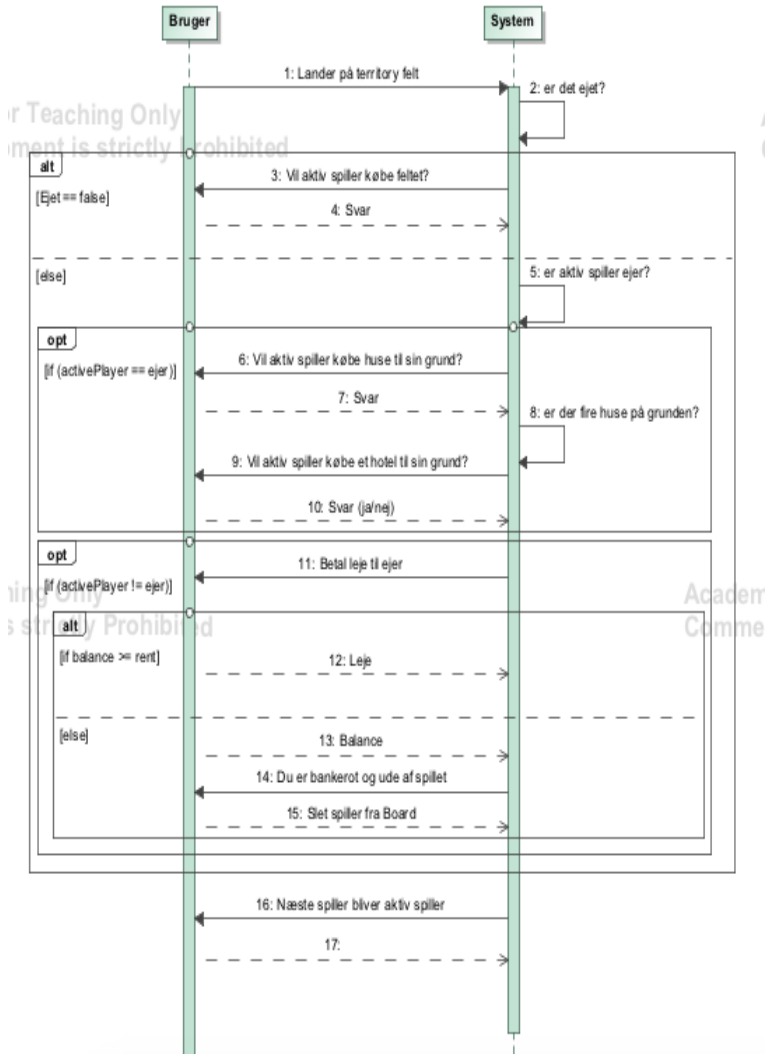


Figur 3.12: Kravsporing

3.5 Systemsekvens analyse

Med systemsekvens diagrammet vist i figur 3.13 er der vist hvordan systemet og brugeren interagerer ved, at den aktive spiller lander på et territory felt. Her bliver der spurgt til om det felt spilleren lander på er ejet. Hvis ikke feltet er ejet skal spilleren svare på om han vil købe feltet. Dette vil blive

vist som en valgmulighed i interfacet. Derefter vil systemet finde ud af om det er den aktive spiller der også ejer feltet. Hvis spilleren ejer feltet skal vedkommende nu svare på om han vil købe huse til sin grund. Hvis spilleren allerede har 4 huse på grunden får han valgmuligheden at placere et hotel på grunden.
Hvis ikke den aktive spiller ejer feltet skal der betales leje til ejeren af feltet. Her er det selvfølgelig vigtigt, at balancen på den aktive spillers konto er større end den leje der skal betales til ejeren. Hvis ikke spilleren har nok penge til at betale lejen er han gået bankerot og bliver slettet fra Gameboardet.
Herefter går turen videre til den næste spiller der nu skal slå med terningerne.



Figur 3.13: Systemsekvens analyse for at lande på et territory felt.

4 Design og Implementering

Under implementering er der lagt særlig vægt på at overholde GRASP principperne fra tidligere CDIO-projekter og også på, at holde strukturen fra de tidligere projekter. Af hensyn til overblikket over projektet, har vi altså her valgt at bygge videre på en af gruppemedlemmernes projekt fra sidste år. Dette har gjort det muligt, at lave et i forvejen godt projekt endnu bedre, omend den i forvejen implementerede løsning var en smule abstrakt i forhold til hvad den burde være.
Alting er ligesom sidst delt op i pakker, der er kommet nye klasser og en enkelt ny pakke til. Den nye pakke 'game.entities.cards' holder styr på lykke kortene. Pakkerne er som følger:

- App
- game.boundaries
- game.controllers
- game.entities
- game.entities.cards
- game.entities.fields
- game.resources
- game.util
- test
- test.mockClasses

4.1 3-lags modellen

Tre-lags-arkitektur bruges til, at inddele et program i tre lag der så vidt muligt bliver holdt adskilt, så hele programmet bliver mere overskueligt. De tre forskellige lag er delt op således:

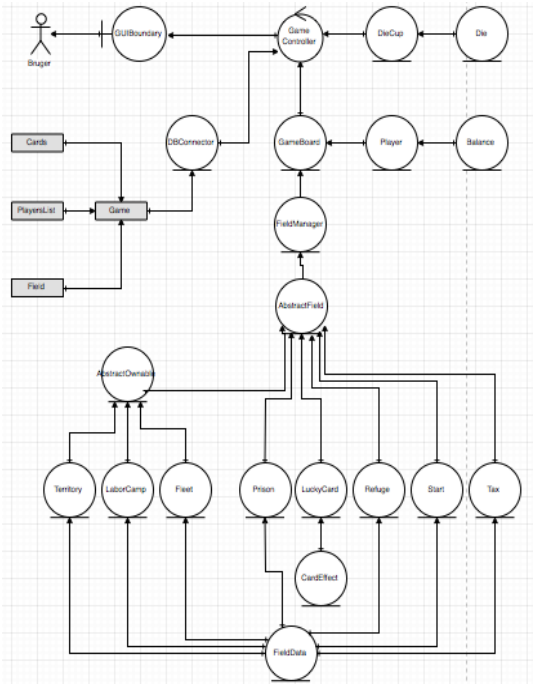
- Præsentationslag
- Logiklag
- Datalag

Præsentationslaget er det lag, der er kendt for at være meget tæt på brugeren. Det er det øverste lag der håndterer modtagelse og præsentation af data. Logiklaget er det midterste lag, der håndterer udveksling af data mellem præsentationslaget og datalaget. Datalaget er det nederste lag, der modsat det

øverste lag er kendt for at være tæt på computeren. Dette lag opbevarer og håndterer data.
Skal man kigge nærmere på vores projekt, overholder det faktisk 3-lags modellens principper. Vi har nemlig et datalag, som er vores resourcefiler - CardEffect og FieldData. Derefter har vi vores logiklag, som er de respektive logiklag for de enkelte felter og de enkelte kort. Alt denne logik samles igennem vores FieldManager, som bruger logikken, og igennem denne (som er en medierende klasse), kaldes så vores præsentationslag. Dette er vores GUIBoundary, der har til formål at kommunikere data ud til GUI. Man kunne egentlig godt argumentere for, at vores præsentationslag, burde være vores GUI, men da det er vores GUIBoundary, der kommunikerer med GUI, er det altså efter vores opfattelse dette, som er vores præsentationslag.

4.2 Boundary-Controller-Entity model

Spillet er denne gang en del anderledes, end det tidligere har været. Specielt fordi der er kommet en del flere elementer på. Disse elementer kan ses af vores BCE-model nedenunder i figur 4.1. I forhold til tidligere har vi nu fået lidt flere felter på, men også nogle lykkekort. Alle disse entities er helt essentielle for spillet, og er opbygget ud fra polymorfi.



Figur 4.1: BCE model

4.3 Implementering af tilstandsmaskine

Spilleren bevæger sig imellem en række mulige tilstande:

- 0. Starttilstand
- 1. Klar til terningkast
- 2. Valg
- 3. Venter på tur
- 4. Venter på tur imens spilleren er fængslet
- 5. Sluttilstand: Spilleren er bankerot eller har vundet
- 6. Fejltilstand

Hændelser: Spilleren udsættes for en lang række af hændelser, blandt andet udbetalinger, indbetalinger og dobbelt slag. For at beskrive tilstandsmaskinen har vi grupperet disse hændelser efter hvilken påvirkning de har på spillerens tilstand. Vi ignorerer hændelser der ikke har nogen direkte effekt på spillerens tilstand. Så hændelser som ind- og udbetalinger, lykkekort og passér start, indgår kun i det omfang de resulterer i fængsling, bankerot, valg eller at spilleren vinder. De mulige hændelser er som følger:

A: Almindelige hændelser: Gruppering af alle hændelser der får spillet til at fortsætte som ellers. Dette dækker blandt andet over at lande felt ejet af en modspiller, lande på et ikke bebyggeligt felt spilleren selv ejer eller lande på et draw card felt der ikke er ”gå i fængsel”.

B: Udbetalinger der resulter i at enten spilleren selv eller alle andre spillere går bankerot.
D: Dobbeltslag. Terningerne har samme antal øjne. Dette giver spilleren en ekstra tur, forudsat at der ikke samtdig foregår en hændelse der forhindrer dette.

F: Fængsling. Land på gå til fængsel, træk gå til fængsel kort eller en allerede fængslet spiller der ikke formår at komme ud af fængslet i en given runde.

S: Tidligere spillers tur er slut.

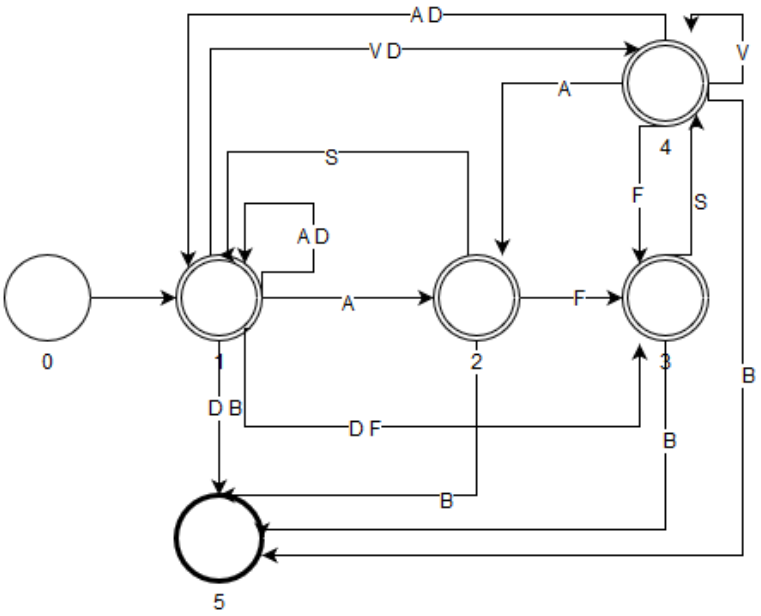
V: Valg. Alle hændelser der resulter i at spilleren stilles over for et valg. Dette kan være hvorvidt spilleren ønsker at købe eller sælge grunde eller aktiver, hvorledes en spiller ønsker at forsøge at slippe ud af fængslet eller hvilke grunde en spiller ønsker at pantsætte, i tilfælde af at spilleren

ikke har tilstrækkelig kapital til at dække en udgift.

Tilstande \ Hændelser	A	B1	B2	D	F	S	V
0	6	5	5	6	6	1	6
1	3	5	5	1, 2, 4, 5	4	6	2
3	6	5	5	6	6	1	2
4	6	5	5	6	6	6	2
2	1	5	6	6	4	6	2
5							
6							

Figur 4.2: Tilstandstabel

Som det kan ses ud fra figur 4.2 og figur 4.3 kan hændelse D, resultere i mere end en hændelse - fængsling, klar til terningkast, bankerot og valg. Der er derfor tale om en non-deterministisk hændelsesmaskine.



Figur 4.3: Tilstands Diagram

4.4 Implementering af interfaces samt exceptions

Det har undervejs, været en rigtig god ting at implementere både interfaces, samt at benytte exceptions. I forbindelse med vores implementering af SQL

scripts, viste det sig i særdeleshed, at være en god ting at bruge try/catch, da vi på denne måde nemt kunne finde frem til de relevante fejl, der kunne være opstået i SQL-scriptet uden, at spillet crasher.
Vi har i dette tilfælde brugt de indbyggede exceptions, men det kan i visse tilfælde være en rigtig god ide, at lave sine egne exceptions-klasse, hvis man skal arbejde med større og mere avancerede systemer. Vi har dog ikke ment, at der har været behov for det i dette tilfælde, da vores system sagtens kunne udvikles, ved brug af de i forvejen eksisterende exceptionsklasser.
I rigtig mange tilfælde kan man benytte sig at Java's "throwable"klasse, som enten kaster en fejl, eller en exception. Simple programmer vil normalt ikke lave fejl eller kaste exceptions, men kommer man op i lidt mere avancerede systemer, som for eksempel Matadorspillet for dette semester, så kan de godt kaste fejl af sig, hvoraf exceptions kan være en god ide.
Hvis en fejl opstår, vil det oftest være noget logisk eller runtime mæssigt, som heraf følger, at programmet ofte crasher. Kastes derimod en exception, betyder det, at der ikke er tale om en alvorlig fejl, men at programmet sagtens kan køre videre. Et eksempel på en exception, kunne være at den database man forsøger at oprette forbindelse til igennem JDBC, allerede findes, og derfor ikke kan oprettes.
Det har været en formidabel ide med interfaces, da vi derved kan lave vores specifikke funktioner, og udskifte i dem, uden at ændre på det egentlig funktionsnavn, det har også vist sig at være en ret lækker ting, når man skal lave en masse funktioner, til senere brug, men ikke laver dem med det samme.

Et interface betegnes som regel som en klasse med en masse funktioner uden kroppe. Det giver også et rigtig godt overblik over de funktioner, som der ligger til grunde for en specifik klasse, uden at man bliver overvældet med kode.

4.5 Klassemodellering via GRASP

Ligesom i sidste projekt, er spillet hovedsageligt bygget op omkring polymorfi. Alle vores felter er opbygget som entity klasser, der beskriver de enkelte felter. Der er altså lagt en del vægt på GRASP principperne, specielt Creator, Controller, Information Expert, High Cohesion, Polymorphism samt Indirection.
Da vi har lagt en del vægt på netop GRASP, er koden blevet mere dynamisk og kan desuden igen genanvendes, hvis vi nu skulle vælge at tage Matador til et helt nyt niveau. Alt dette er gjort for at overholde GRASP principperne([Lar01] afsnit 17 og 25)., som har vist sig at være rigtig nyttige

4.5.1 Information Expert

Vores entity klasser, som var det første vi designede, var stort set allerede klar. Der har ikke været ret mange ændringer fra sidste spil. Labor-camp, Fleet samt Tax klasserne er dog genbrugt fra vores tidligere projekt sammen med balance klassen. Player klassen samt GameBoard klassen er blev udvidet lidt, men er ellers også genbrugt, da vi fra vejleders side fik, at vide at det var mest hensigtsmæssigt. Refuge klassen, har også fået lov at blive, af en enkelt årsag, som beskrives lidt senere. Af nye felter har vi:

1. cards.AbstractCard
2. cards.AbstractOwnable
3. cards.Refuge
4. cards.Tax
5. cards.Prison
6. cards.MoveActivePlayer
7. fields.Start
8. fields.Territory

Ideen bag information expert er, at placere ansvaret for klasserne, der hvor det passer bedst. Eksempeltvis nedarver vores Territory felt fra AbstractOwnable. Det giver altså her mening at placere ansvaret for om et felt er ejet i Territory klassen, da det er feltet selv det ved om der er en ejer, og givetvis hvem ejeren af det enkelte felt egentlig er.
Vi havde tidligere en Refuge klasse, som havde en del funktionalitet, denne er dog ikke særligt brugbar i dette spil, da vi ingen refuge felter har. Vi har dog valgt(fordi at det er sådan fra GUI'ens skaber), at parkeringsfeltet skal være et Refuge felt. Nogle gange spilles med at man skal lægge et bestemt beløb over til parkeringsfeltet, hver gang man gør en bestemt handling.
Dette er dog ikke et krav, eller en regel i dette spil. Vi har derfor valgt at der ved landing på parkeringsfeltet, indsættes kr. 200 på spillerens konto.
At bruge netop Refuge feltet, ved parkering, gør også GUI'en lidt pænere, da der herved fremkommet et fint billede. Vi så derfor ingen grund til ikke at benytte Refuge feltet her, fremfor at skrotte klassen, til trods for at det er en uhyre simpel klasse.

4.5.2 Creator

Creator mønstret bruges til at finde frem til, hvor det er bedst at instantiere et objekt. Hvem der har dataen, hvem der skal bruge objektet, samt om objektet gemmer på nogen, skal her tages i betragtning. Kigger man på disse ledetråde var det faktisk logisk, at vores luckycards ikke skulle instantieres af selve luckycard feltet, da det herved ikke ville være muligt at gemme korrekt på objektet. Dataen ligger i en seperat klasse, og derfor var hvem der havde dataen, egentlig ikke noget der spillede ret meget ind her.
Derfor er det vores FieldManager, som er meget tæt med vores felter, som har fået ansvaret, for at instantiere selve luckycard feltet, og samtidig også instantiere selve kortene. Dette gjorde det muligt, blandt andet at have en enkelt kortbunke, i stedet for at instantiere en ny for hver bunke.

4.5.3 Indirection og High Cohesion

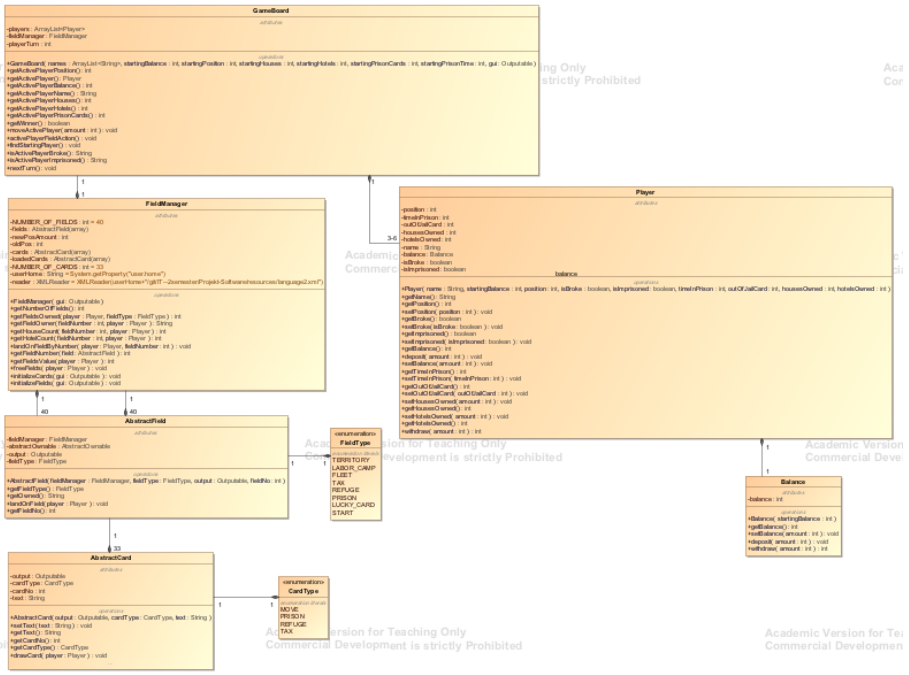
Vi benytter os af både Indirection samt High Cohesion ved brug af vores GUIBoundary i forbindelse med vores FieldManager. Ideen her er at man laver en klasse, hvis objekter nu skal administrere kommunikationen mellem andre objekter. FieldManager klassen medierer her mellem GameBoardet og Felterne. Grunden til at FieldManageren i sin tid blev til var, at vi skulle bruge info om hvor mange felter af en bestemt type, en bestemt spiller havde. Derfor oprettede vi en Fieldmanager, hvis eneste ansvar er, at holde styr på felterne. Præcis samme system skulle have været brugt ved vores CardManager, altså en medierende klasse, der skulle holde styr på de enkelte kort, deres placering, om de var ejet, hvis det var fængselskort, og i givetfald, hvem ejeren var, og om denne havde brugt kortet.
Dette blev droppet, dog ikke med god vilje, da dette ville have forstærket High Cohesion mønstret endnu mere, end det i forvejen er. FieldManager klassen er med nogle simple modificeringer i initializeFields() metoden genbrugt fra sidste gang, da den var en rimelig stor del af projektet dengang. Havde CardManager ideen dog fungeret, havde vi haft endnu en indirection-klasse. Grunden til at CardManager ikke kom med, er at vi ikke havde noget behov for klassen.

Vi indså at det ansvar som CardManager, burde have haft, lige så godt kunne blive smidt ned i LuckyCard klassen, da det alligevel er denne, som opretter kortene. En fejl der dog er ved, at gøre på denne måde, er at det så i givetfald er en ny bunke kort, der oprettes for hvert lykkekort felt. Dette gør dog også kortene mere tilfældige. Alt i alt endte vi dog ud med at placere instantieringen af kortene i vores FieldManager klasse, frem for en CardManager. Vi indså, at en ekstra klasse, ville være uhensigtsmæssigt da dette ville skabe meget mere arbejde end nødvendigt, og da vores lykkekort jo er tilhørende en speciel felttype, så gav det rent faktisk god mening at

lade FieldManager lave kortene, og trække dem fra denne.

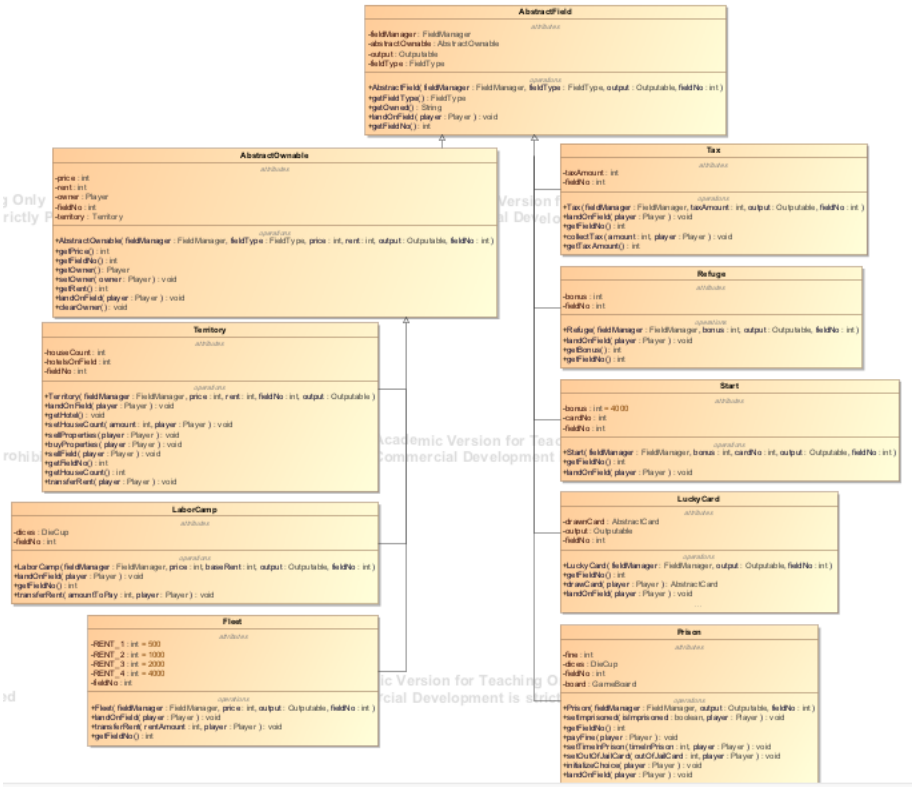
På denne måde omgik vi også de 4 kortbunker, og har derfor kun den ene kortbunke, som er nødvendigt for spillet. Igen er FieldManager her altså medierende klasse mellem spilleren og kortene, da det er denne som faktisk har kortene. LuckyCard feltet, trækker dog selv kortet fra bunken, og har derfor en smule logik i sig.

Figur 4.4 er diagrammet over de forskellige klasser vi har med i vores Matadorspil. Diagrammet giver et godt overblik over hvordan de forskellige klasser inteagerer med hinanden og hvilke forhold de har til hinanden.



Figur 4.4: Klassediagram

Figur 4.5 viser et diagram over hvilke klasser i vores Matadorspil der er superklasser og hvilke der arver fra disse. Hele vores spil er bygget op omkring principper bag polymorfi og derfor er figur 4.5 med til, at give et stort overblik over hvordan en af de vigtigste dele i vores spil er designet.



Figur 4.5: Diagram over polymorfi

4.5.4 Polymorphism: Abstract og Interfaces

Undervejs i implementeringen blev der forsøgt at overholde principperne fra sidste projekt, nemlig at benytte polymorfi til at skabe felterne. Herunder var idéen at have en AbstractCard klasse og en AbstractOwnable klasse, som resten af kortene så nedarvede fra. Ved at trække det næste kort i bunken, gennem LuckyCard klassen, returneres det korrekte kort så, ud fra hvilken type de enkelte kort er.

Den oprindelige idé var at mediere igennem en CardManager, men da denne blev droppet, udskriver LuckyCard klassen nu direkte til GUI. Det var til vores store forbavselse ikke muligt for os at kommunikere med GUI, igennem CardManager klassen. Grunden hertil, lå i at GUI simpelthen var så begrænset, at der forekom en fejl, når vi forsøgte at kalde det direkte på kort klasserne. Derfor blev dette nu lagt direkte ud i GUI, og skulle nu kaldes udfra kort nummer. Dette fungerede, dog løb vi nu ind i at der gik en hel runde før at GUI opdaterede og udførte handlingen fra kortet. Altså kunne

en spiller få besked på at rykke tre felter tilbage, men først runden efter. Derfor blev udfaldet at det nu er LuckyCard klassen selv, som skal kalde en ShowCardMessage på GUIBoundary, således at kortene bliver vist, men at metoden på kortet samtidig bliver udført.
Vi endte derfor med, at have 4 kort-typer, som alle nedarver fra deres Abstrakte klasse, som enten er AbstractCard eller AbstractOwnable. Alle felter pånær prison, nedarver fra AbstractCard, Prison klassen, som altså kan ejes af en spiller, nedarver fra AbstractOwnable.

4.5.5 Controller og hjælpeklasser

Som tidligere har vi benyttet vores XML-parser til, at indlæse det korrekte sprog. Det er ikke et egentligt krav at spillet skal kunne dette, men vi kunne se logikken i det, da det i tidligere CDIO opgave blev stillet som krav, og vi har derfor beholdt denne. Dette har også gjort det nemt for os at rette i specifikke strings, således at den tekst der udskrives til GUI passer. Alt tekst skrives som tidligere også til GUI via GUIBoundary klassen, som denne gang indeholder en del flere metoder, grundet udvidelsen af spillet. Vores XML-parser er en hjælpeklasse, præcist som vores terninger, og raflebæger også er.
Vores DBConnector er også en hjælpeklasse, da denne sørger for, at oprette forbindelse til den korrekte database, og vi igennem denne også kan udføre QUERYS, på vores SQL.
Som tidligere har vi også en GameController, denne gang modificeret lidt, altså er denne ikke genbrugt helt, da spillet skulle ændres lidt på, for blandt andet at kunne gemme og hente et spil. Samtidig skulle GameControlleren nu ikke kun holde styr på om en spiller var bankerot, men også på om spilleren var i fængsel. Selve XML-Parser klassen, die og dieCup klasserne er dog genanvendt fra tidigere projekt. Vi mener at man kan retfærdiggøre for dette, da en terning ikke kan beskrives særligt forskellige, et raflebæger kan næsten også kun laves på en måde, og vores XML-Parser var utroligt pålidelig tidligere, hvorfor denne også er genanvendt.

4.6 Implementering af LuckyCards

Det er som tidligere omtalt en kæmpe fordel med Abstrakte klasser og polymorfi, når man skal designe flere instanser af det samme. Derfor er vores felter blever til en masse klasser, beskrevet tidligere, som udfra polymorfi instantieres af vores FieldManager. Herfra kan spilleren så trække det første kort i bunken, når man lander på et LuckyCard felt.
Koden der trækker første kort, og smider det om bag i bunken, se herunder.

```
//Draw a card like in a queue
public AbstractCard drawCard(Player player){

    this.drawnCard=FieldManager.cards[0];
    this.drawnCard.drawCard(player);
    for(int i = 0; i < (FieldManager.cards.length-1); i++){
        FieldManager.cards[i]=FieldManager.cards[i+1];
    }
    FieldManager.cards[FieldManager.cards.length-1]=drawnCard;
    output.showCard(drawnCard.getText());
    return drawnCard;
}
public void landOnField(Player player){
    drawCard(player);
}
```

Figur 4.6: DrawCard metoden

Reelt set, så instantierer vores FieldManager kortene som en array af AbstractCard. Herfra ligges de så i den korrekte rækkefølge, altså efter kortnummer. De blandes herefter af en anden kode, før de ligger klar i vores FieldManager. Når først kortene ligger klar, og en spiller lander på feltet, kaldes metoden, som vi fik besked på at bruge sidst, nemlig landOnField. Denne metode kalder drawCard metoden, som tager en spiller som argument. Herefter taget der første kort i bunken af kort i FieldManager klassen, og dennes drawCard metode kaldes, som udfører den pågældende handling som kortet har. Til sidst lægges kortet om bag i bunken igen, og teksten på kortet kaldes, og vises for spilleren på GUI.

Reelt set skulle der her have været en ekstra metode, som fortalte at hvis kortet var et fængselskort, så skulle spilleren gemme dette(hvilket også sket), og kortet skulle fjernes fra bunken - sidstenævnte sker dog i midlertid ikke med denne kode, selvom vi nemt kunne have lavet en midlertidig bunke, som indeholdte de kort, som var ude hos spillerne, og så lægge dem tilbage når kortene var brugt. Vi har dog valgt, at det ikke skulle indgå i vores spil, derfor lægges kortet om bag i bunken igen. Omvendt er chancen for, at hele bunken er kørt igennem, inden de andre kort er brugt heller ikke særligt stor, hvorfor vi har nedprioriteret det.

5 Implementering af SQL

Til dette projekt har det været nødvendigt, at få implementeret SQL da dette indgik i vores krav som en del af projektets overordnede indhold. Vi har fået koblet vores Matadorspil sammen med en MySQL-server så funktioner som gem/hent spil kunne blive en realitet.
Design og kommunikationen mellem spillet og databasen vil i de følgende afsnit blive afklaret.

5.1 Tabeller

En tabel er noget man har i sin database. Disse tabeller indeholder bestemte informationer da de er udstyret med bestemte nøgler. Tabellerne gemmer på informationer som de er givet, så man senere kan tilgå og bruge dem (Section 2.2, Page 42) [SFS11].
Vi har til dette projekt valgt, at oprette fire forskellige tabeller i vores database. Disse tabeller er nok til at kunne gemme og senere tilgå et gemt spil. Vores database er sat op til at gemme spillet automatisk for hver runde og felterne gemmes for hver handling.
Følgende tabeller indgår i vores database (kan ses på figur 9.1):

- PlayersList
- Game
- Field
- Cards

- PlayersList gemmer først og fremmest navnet på spillerne i det aktive spil. Positionen på spilleren samt antal af ejede huse/hoteller bliver også gemt heri. Antallet af PrisonCards samt PlayerBalance hører også til under denne tabel.

- Game er den primære tabel. Det er denne tabel der gemmer spillet undervejs og gemmer på hvilet spillers tur det er.

- Field holder styr på felterne og hvilke spillere der ejer disse samt hvad og hvormeget spilleren har fået bygget på de forskellige felter

-Cards-tabellen gemmer de forskellige ejere af kortene samt de forskellige typer af kort og deres numre.

5.2 Nøgler

Når der er tale om en database og tabellerne til denne, så bliver man nødt til, at have sammenhæng mellem de forskellige tabeller. De forskellige nøgler vi igennem dette forløb har stiftet bekendskab med (Section 2.3, page 45) [SFS11] er som følger:

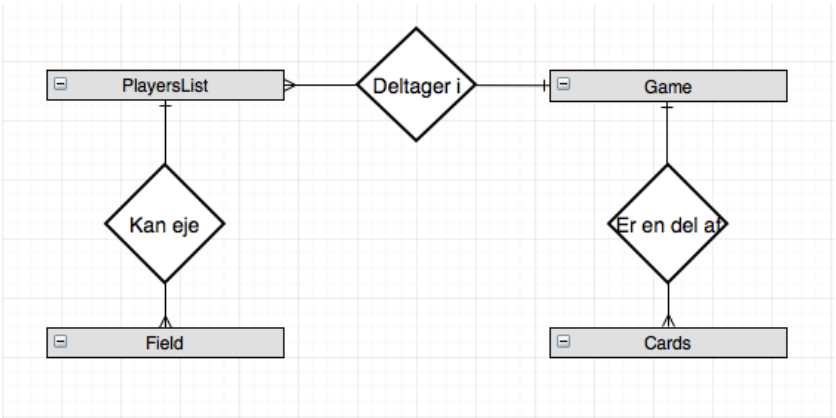
- Primarykey
- Foreignkey
- Superkey

Hver af disse forskellige nøgler har en bestemt rolle og betydning for hvordan de forskellige tabeller hører sammen.

5.3 Design af E/R diagram

"The entity-relationship (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database"(Section 7.2, page 262) [SFS11]

Som tidligere nævnt bliver der blot oprettet 4 tabeller når spillet startes. Nedenfor er det illustreret hvordan de forskellige tabeller interagerer.



Figur 5.1: E/R-diagram

Som man også kan se af E/R-diagrammet er SQL implementeringen meget enkel. Der er blot de fire tabeller hvor Game tabellen er den helt centrale. Spillet er opbygget sådan, at man kun kan have et spil kørende af gangen, men at man kan gemme og komme tilbage til et tidligere spil. Heraf kan der udledes at man således kun have et spil kørende med flere spillere. En spiller

kan eje flere felter hvor han kan vælge at bygge huse eller hoteller. Desuden findes der kort som spilleren kan eje. Det er de såkaldte benådningskort som der findes to af i 'Prøv Lykken'-stakken. En spiller kan således eje begge kort på samme tid.

Undervejs i udviklingen af E/R-diagrammet har der været forskellige ideer til hvordan databasen skulle designes, og der har bl.a. været helt op mod 6 forskellige tabeller i spil. Spillet er endt ud med kun at have fire forskellige da det har været mest hensigtsmæssigt, i forhold til hvordan spillet var designet fra sidste semester. Desuden er programmet nu designet således, at der bliver gemt undervejs gennem Game tabellen som er den primære tabel. De endelige metoder som er blevet anvendt kan ses i EER-diagrammet (Section 7.8, Page 295) [SFS11] under bilag som figur 9.1. Her er også indtegnet primær nøgler for hver tabel. De SQL-scripts der er blevet benyttet, er beskrevet under punkt 13.1.

5.4 Persistens mellem program og JDBC

"The application program typically communicates with a database server, through ODBC, JDBC, or other protocols, in order to get or store data"(Section 9.2.3, Page 380) [SFS11]

5.5 Optimering af database kald

Det blev undervejs besluttet at vi ikke ville begrænse vores SQL kald, mere end højest nødvendigt. Det er et krav, at vi skal bruge den udleverede GUI, men dette sætter hvertfald også en enkelt begrænsning. Vi kan nemlig ikke kalde en specifik metode, når spillet lukkes, hvorfor vores eneste to muligheder er:

1. Gemme under hele forløbet
2. Gemme ved tryk på en knap

Vi valgte at vi ville gemme under vejs i hele spillet, da vi mener at dette er mest sikkert. I tilfælde af, at man faktisk skal foretage en aktiv handling, er det ikke just sikkert, at de nødvendige ting gemmes, før man kommer til at lukke spillet. Omvendt, så er software aldrig perfekt, og hvis spillet skulle crashe, så gemmer det altså ikke, hvis man skal foretage en aktiv handling. Derfor gemmer spillet selv alt undervejs.
På denne måde sikrer vi os, at nødvendig data gemmes, uden at genere brugeren. Vi har under vejs ikke oplevet at skulle vente på database kald, andet end når man starter et nyt spil, men dette kan ikke umiddelbart optimeres mere, da databaserne jo skal oprettes. Omvendt fandt vi hurtigt ud af, at vi sløvede databasen en hel del ned, jo flere databaser den faktisk skulle lagre. Derfor slettes et afsluttet spil, når der er fundet en vinder.

Der er undervejs forsøgt, at holde kaldene så små som muligt, ligesom det er forsøgt kun, at opdatere, når der er behov for det, ofte efter en spillers tur.

6 Tests

Vi har valgt at inkludere nogle tests, af de klasser som vi lavede i forbindelse med vores første ide med chancekortene, nemlig, at bruge polymorfi. Herfra er der testet for nogle givne cases, for at se om vores enkelte chancekort egentlig virkede. Testene viser, at kortene faktisk virkede, men at der altså var et andet problem. Vores CardManager kunne også godt oprette kortene, efter samme princip, som vores FieldManager opretter felterne, men vi kunne altså ikke kalde DrawCard() metoden, når vi samtidig også skulle kommunikere med GUI. Ansvaret for kortene, blev derfor tildelt til FieldManager klassen, i stedet for CardManager. Alternativt skulle LuckyCard klassen selv have haft ansvaret for at instantiere, men så havde vi 4 kortbunker.

6.1 Testcases

Vi mener, at det ikke har været nødvendigt, at teste for selve spillets logik men kun for de nye implementeringer vi har i spillet i forhold til sidste semester. Dette gælder for LuckyCard- og databaseimplementeringen.

6.1.1 Database: ConnectionTest

Preconditions

- Have adgang til spil.
- Have adgang til en server.

Test

- Oprette forbindelse til database.

Postconditions

- Forbindelse til databasen oprettet.

6.1.2 Database: createTableTest

Preconditions

- Forbindelse til database skal være etableret.

Test

- Oprette en ny tabel i databasen.

Postconditions

- Den nye tabel er oprettet.

6.1.3 Database: insertToTable

Preconditions

- Forbindelse til database skal være etableret.

Test

- Sende nye data ind i tabellen.

Postconditions

- De nye værdier er oprettet i tabellen.

6.1.4 LuckyCard: MoveActivePlayerTest

Preconditions

- Spilleren står på felt 5.

Test

- Spilleren skal rykke et enkelt felt frem.

Postconditions

- Spilleren ender på felt 6.

6.1.5 LuckyCard: RefugeCardTest

Preconditions

- Spilleren har allerede 1.000 i balance.

Test

- Spilleren skal have 1.000 mere.

Postconditions

- Spilleren har 2.000 sammenlagt.

6.1.6 LuckyCard: usePrisonCardTest

Preconditions

- Spiller skal ikke have et PrisonCard.

Test

- Spiller bruger sit PrisonCard til, at komme ud af fængsel.

Postconditions

- Spilleren er nu ude af fængsel.

6.1.7 LuckyCard: LuckyCardQueueTest

Preconditions

- Spilleren har allerede et PrisonCard.

Test

- Spilleren skal ikke have endnu et PrisonCard.

Postconditions

- De to kort skal ikke være de samme da kortene ligger i kø.

6.1.8 LuckyCard: TaxCardTest

Preconditions

- Spilleren har en balance på 25.000 og ejer 4 huse og 2 hoteller.
- Tax-kortet har kortnummer 13.

Test

- Spilleren trækker et tax-kort.

Postconditions

- Spillerens balance er 17.200 jf. resources -> 'Matador - cards.csv' i javakoden.

6.2 Testrapport

6.2.1 Database: ConnectionTest

ConnectionTest	Dato	Commit Link	Succes
Preconditions			
Have adgang til spil	12/5-16	http://bit.ly/27j1fNj	Ja
Have adgang til en server	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Oprette forbindelse til database	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Der er oprettet forbindelse til databasen	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.1: Testrapport for ConnectionTest

6.2.2 Database: createTableTest

createTableTest	Dato	Commit Link	Succes
Preconditions			
Forbindelse til database skal være etableret	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Oprette en ny tabel i databasen	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Den nye tabel er oprettet	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.2: Testrapport for createTableTest

6.2.3 Database: insertToTable

insertToTable	Dato	Commit Link	Succes
Preconditions			
Forbindelse til database skal være etableret	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Sende nye data ind i tabellen	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
De nye værdier er oprettet tabellen	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.3: Testrapport for insertToTable

6.2.4 LuckyCard: MoveActivePlayerTest

MoveActivePlayerTest	Dato	Commit Link	Succes
Preconditions			
Spilleren står på felt 5	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Spilleren skal rykke et enkelt felt frem	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Spilleren står på felt 6	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.4: Testrapport for MoveActivePlayerTest

6.2.5 LuckyCard: RefugeCardTest

RefugeCardTest	Dato	Commit Link	Succes
Preconditions			
Spilleren har allerede 1.000 i balance	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Spilleren skal nu have 1.000 mere	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Spilleren har 2.000 i balance	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.5: Testrapport for RefugeCardTest

6.2.6 LuckyCard: usePrisonCardTest

usePrisonCardTest	Dato	Commit Link	Succes
Preconditions			
Spiller skal have et PrisonCard	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Spiller bruger sit PrisonCard til, at komme ud af fængsel	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Spilleren er ude af fængslet	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.6: Testrapport for usePrisonCardTest

6.2.7 LuckyCard: LuckyCardQueueTest

LuckyCardQueueTest	Dato	Commit Link	Succes
Preconditions			
Spilleren er ikke fængslet	12/5-16	http://bit.ly/27j1fNj	Ja
Spiller har ikke et PrisonCard	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Spiller bruger sit PrisonCard til, at komme ud af fængsel	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Spilleren er ude af fængslet	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.7: Testrapport for usePrisonCardTest

6.2.8 LuckyCard: TaxCardTest

TaxCardTest	Dato	Commit Link	Succes
Preconditions			
Spilleren har en balance på 25.000 og ejer 4 huse og 2 hoteller	12/5-16	http://bit.ly/27j1fNj	Ja
Tax-kortet har kortnummer 13	12/5-16	http://bit.ly/27j1fNj	Ja
Test			
Spilleren trækker et tax-kort	12/5-16	http://bit.ly/27j1fNj	Ja
Postconditions			
Spillerens balance er 17.200 jf. resources -> 'Matador - cards.csv' i javakoden	12/5-16	http://bit.ly/27j1fNj	Ja

Tabel 6.8: Testrapport for usePrisonCardTest

7 Brugervejledning

Efter at installationen er afklaret (kan ses under afsnit 13.2), kan man endelig begynde, at spille spillet. Dette sker ved, at man enten trykker 'nyt spil' eller 'load spil' på GUI. Ved tryk på 'nyt spil' vælger man antal spillere og spillernes navn og så er man godt kørende. Ved tryk på 'load game' kan man få lov til, at loade et ældre og igangværende spil.

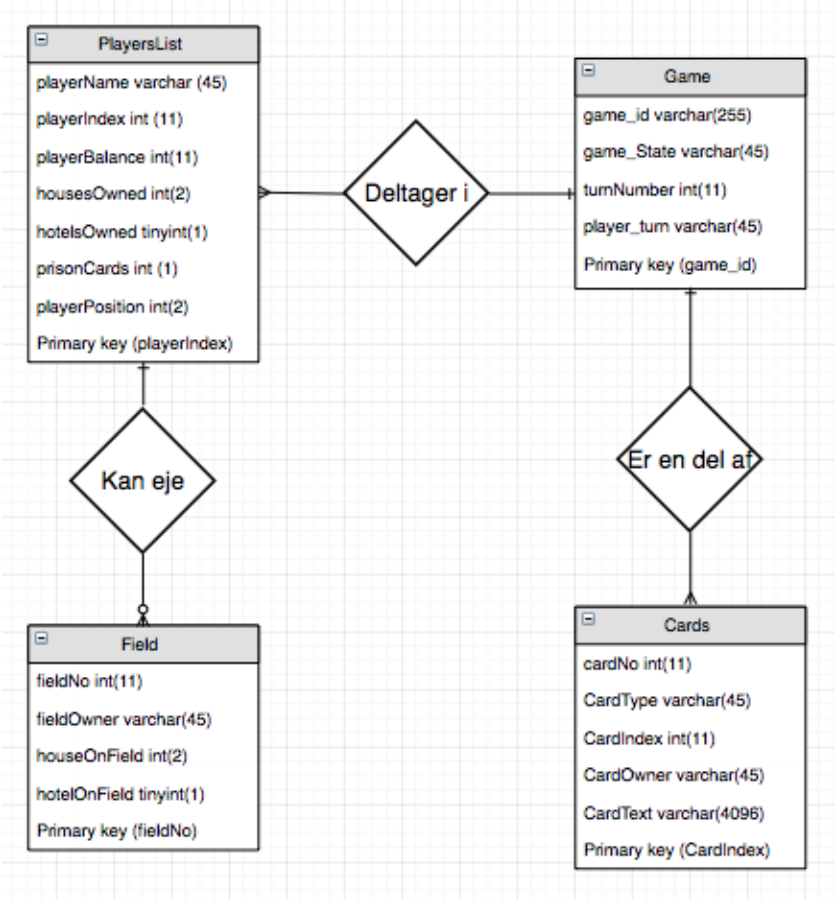
8 Konklusion

Vi har igennem dette projekt fået færdigudviklet et komplet Matadorspil, med en dertil fungerende database, så man både kan gemme og hente et

uafklaret spil.
Vores spil har nu, i forhold til 1. semester, fået implementeret langt flere funktioner. Salg og køb af huse, hoteller og pantsætning samt fængsel og muligheden for, at kunne trække lucky-cards er nu blevet en realitet.
Vores kode er programmeret således, at den er let og overskuelig. Alle de forskellige krav til vores spil er blevet implementeret.

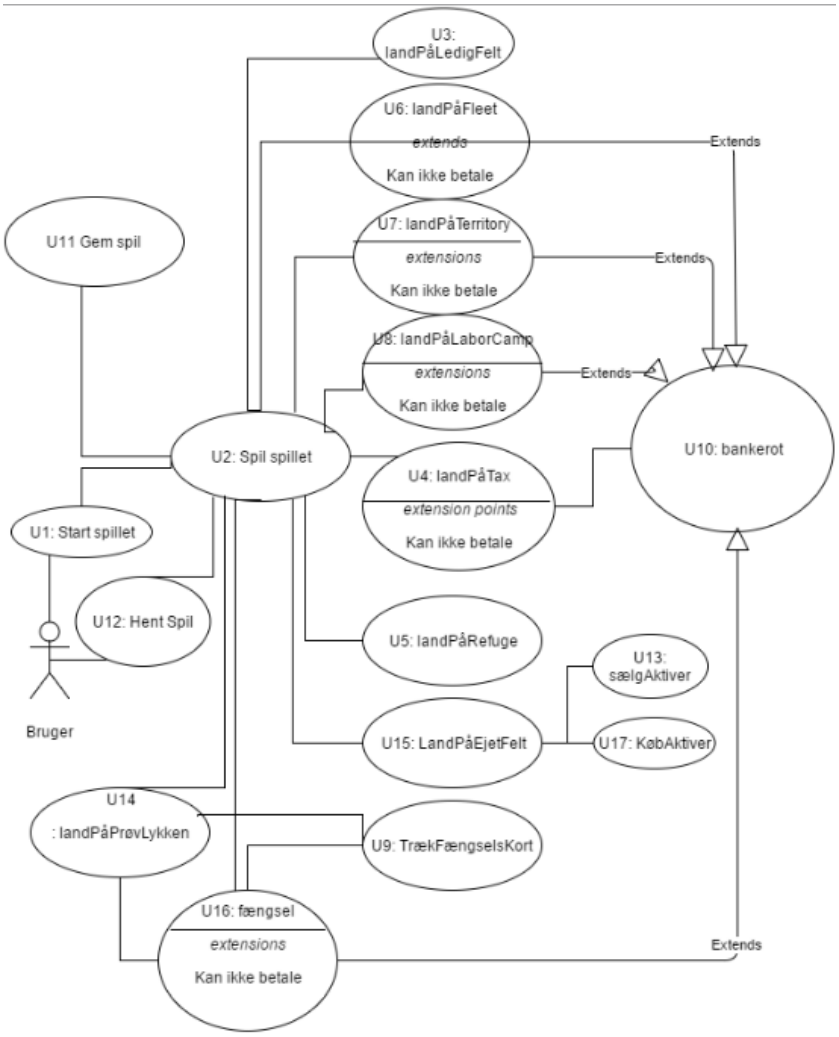
Som ovennævnte indikerer, kan vi hermed konkludere, at vores projekt er vellykket.

9 Bilag



Figur 9.1: EER-diagram

10 UseCase diagram



Figur 10.1: UseCase diagram

11 Usecases 3-8

Use case:	Alternative flow landPåLedigtFelt
ID:	U3
Beskrivelse:	Når en bruger lander på et ledigt felt, vil han have muligheden for at købe feltet.
Primære aktører:	Bruger
Sekundære aktører:	none.
Preconditions:	Spilleren skal lande på et felt der ikke er ejet af nogen
Main flow:	1. Spiller informeres om feltets pris
	2 Hvis spiller har råd til feltet
	2.1. Spiller får mulighed for at købe feltet.
	2.2 Hvis bruger vælger at købe feltet
	2.2.1 Feltes pris fratrækkes nu spillerens konto
	2.2.2 Spilleren ejer feltet
Postcondtions:	Spilleren ejer feltet og feltets pris er fratrukket spillers konto
Alternative flows:	1.1 Spiller har ikke råd til at købe feltet
	2.3 Spiller vælger ikke at købe feltet

Figur 11.1: Usecase U3

Use case:	Alternativt flow: landPåTaxFelt
ID:	U4
Beskrivelse:	Spilleren lander på et tax felt og skal betale en sum penge
Primære aktører:	Bruger
Sekundære aktør	Ingen
Preconditions:	Spilleren er landet på et tax felt
Main flow:	1 Hvis spilleren lander på Goldmine 1.1 Spilleren skal betale 2000 kr 1.2 Hvis spilleren ikke kan betale extension point: kanlkkeBetale 2 Hvis Spilleren lander på tax Caravan 2.1 Brugeren vælger: 2.1.1 Betaler 4000 kr 2.1.2 Betaler 10 % af samlede aktiver 2.1.3 Hvis spilleren ikke kan betale extension point: kanlkkeBetale
Postcondtions:	Spiller har betalt tax, eller er gået bankerot
Alternative flows:	

Figur 11.2: Usecase U4

Use case:	Alternative flow - landPåRefugeFelt
ID:	U5
Beskrivelse:	Splleren lander på refuge felt som udbetaler penge til ham
Primære aktører:	Bruger
Sekundære aktører:	Ingen
Preconditions:	Spilleren er landet på et refuge felt
Main flow:	1. Hvis spilleren lander på Walled City 1.1 Spilleren får 5000 2. Hvis spilleren lander på Monastery 2.1 Spilleren får 500
Postcondtions:	Spilleren har fået indsat 5000 eller 500 på sin konto
Alternative flows:	none

Figur 11.3: Usecase U5

Use case:	Alternative flow - landPåFleetFelt
ID:	U6
Beskrivelse:	Spilleren lander på et ejet Fleet felt
Primære aktører:	Bruger
Sekundære aktører:	none.
Preconditions:	Spilleren skal landet på et Fleet felt der er ejet af en anden spiller
Main flow:	1. Hvis ejeren af feltet har 1 fleet felt betales 500kr. extension point: kanlkkeBetale 2. Hvis ejeren af feltet har 2 fleet felter betales 1000kr. extension point: kanlkkeBetale 3. Hvis ejeren af feltet har 3 fleet felter betales 2000kr. extension point: kanlkkeBetale 4. Hvis ejeren af feltet har 4 fleet felter betales 4000kr. extension point: kanlkkeBetale
Postcondtions:	spiller har betalt beløb, eller er sat bankerot
Alternative flows:	

Figur 11.4: Usecase U6

Use case:	Alternative flow - landPåTarritoryFelt
ID:	U7
Beskrivelse:	Når en spiller lander på et felt ejet af en anden spiller, skal han betale et beløb til ejeren
Primære aktører:	Bruger
Sekundære aktører:	none.
Preconditions:	Spiller skal have landet på et territory felt, der er ejet af en anden spiller
Main flow:	1. Spiller skal betale feltets leje til ejeren. 2 Hvis spilleren ikke kan betale beløbet extension point: kanIkkeBetale 2.1 ellers Beløbet fratrækkes nu spillerens konto
Postcondtions:	spiller har betalt beløb, eller er sat bankerot
Alternative flows:	

Figur 11.5: Usecase U7

Use case:	Alternative flow - landPåLaborCampFelt
ID:	U8
Beskrivelse:	Når spilleren lander på et laborcamp felt skal han slå med terningerne, og betale 100 gange antallet af øjne til ejeren.
Primære aktører:	Bruger
Sekundære aktører:	none.
Preconditions:	Spilleren skal lande på et labor camp felt der er ejet af en anden spiller.
Main flow:	1. Spiller slår med terningerne. 2 Spiller ganger antal øjne på terningerne med 100. 3. Spiller ganger beløbet med antal labor camps eget af ejeren 4. Spiller skal betale beløbet til ejeren extension point: kanIkkeBetale
Postcondtions:	spilleren har betalt. eller er gået bankerot
Alternative flows:	

Figur 11.6: Usecase U8

12 Litteraturliste

[Lar01] Craig Larman. *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition*. One Lake Street, Upper Saddle River, NJ 07458: Pearson Education, Inc., publishing as Addison-Wesley, ISBN 978-0130925695, 2001.

[SFS11] Abraham Silberschatz, Henry F. Korth og S. Sudarshan. *Database System Concepts, Sixth Edition*. 1221 Avenue of the Americas, New York, NY 10020: The McGraw-Hill Companies, 2011. ISBN: 978-125-95298-3.

13 Appendiks: Konfiguration

13.1 SQL scripts

Under vejs er brugt flere forskellige SQL scripts for at kunne gøre implementering af en database muligt. Herunder beskrives nogle af de scripts som vi har brugt undervejs. Først her vores create script, for vores gameTable også kaldet DDL.

```
public void createGame(String gameName){
    String query = ("CREATE TABLE IF NOT EXISTS " + gameName+".game " +
            "(game_id varchar (255) NOT NULL, " +
            "game_State varchar(45) NOT NULL, " +
            "turnNumber int (11) NOT NULL, " +
            "player_turn varchar (45) DEFAULT NULL, " +
            "PRIMARY KEY (game_id));");
    try {
        doUpdate(query);
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figur 13.1: CREATE script

Scriptet her er relativt simpelt, hvis ikke tabellen i forvejen findes, så oprettes den, i den pågældende database med de korrekte attributter. Herefter ligger en tom tabel klar til brug. Derefter skal informationer fyldes i tabellen, dette sker med et ADD script, som er vist herunder, igen er eksemplet for vores gameTable.

```
public void addToGameTable(String gameName, String gameState, int turnNumber, String playerTurn) {
    String query = ("INSERT INTO " + gameName+".game(game_id, game_State, turnNumber, player_turn)"+
            "VALUES('"+gameName+"', '"+gameState+"', '"+turnNumber+"', '"+playerTurn+"');");

    try {
        doUpdate(query);
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

Figur 13.2: ADD script

Når nu tingene er tilføjet til de respektive SQL tabeller, så ville der ikke være ret meget mening i det, hvis ikke man kunne manipulere med disse informationer, derfor er der selvfølgelig også brugt noget DML, til at manipulere med alt dette.

```
public void updateGameTable(String gameName, String gameState, int turnNumber, String playerTurn) {
    String query = ("UPDATE " + gameName+".game " +
            "SET game_State='" + gameState + "', turnNumber='" + turnNumber + "', player_turn='" + playerTurn + "'" +
            "WHERE game_id='"+gameName+"';");

    try {
        doUpdate(query);
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

Figur 13.3: UPDATE script

Til sidst har det selvfølgelig også været nødvendigt at lave nogle querys for at kunne læse alt denne information ind tilbage i spillet, ved load. Derfor er brugt følgende scripts og kode til at indlæse det hele igen. Igen ses et eksempel fra vores gameTable

```
public ArrayList<String> loadGameToArray(String query) throws SQLException{
    ArrayList<String> list= new ArrayList<String>();
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next())
    {
        String PlayerName = rs.getString("player_turn");
        int TurnNumber = rs.getInt("turnNumber");
        list.add(PlayerName);
        list.add(String.valueOf(TurnNumber));
    }
    stmt.close();
    rs.close();
    return list;
}
```

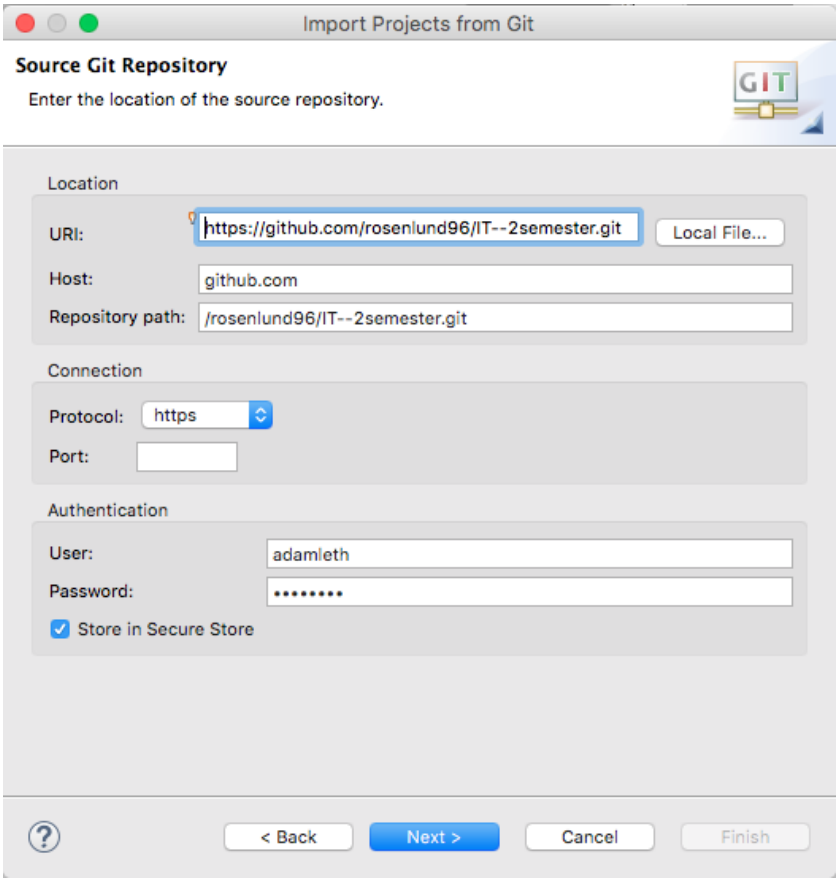
Figur 13.4: QUERY til load

13.2 Klon repository

Der bliver i dette projekt brugt GitHub til udviklingen af programmet. Det har betydet at vi har kunnet arbejde på projektet løbende og alle kunne bidrage med commits undervejs.
For at kunne installere vores spil, skal man først have installeret Java 1.8. Herefter skal vores projekt importeres til Eclipse hvilket sker således:

- File -> Import -> Git -> Projects from Git -> Clone URI

Her skal man vælge stien til vores repository som kan importeres ved følgende indtastning:



Figur 13.5: Klon repository

Da vores spil kører ved hjælp af en MySQL-database, skal dette også installeres. Vi har under udviklingen brugt USB WebServer, derfor kan det være at login informationerne samt port til databasen, skal ændres, inden man kan logge på. Der er intet behov for at have oprettet specifikke databaser, når man vil køre spillet, dette sker nemlig helt automatisk, når man laver et nyt spil.