









Danmarks tekniske universitet

02312 Indledende programmering

CDIO Final

Gruppe 14

	Mette Rejmers Tobiasen	S154673
	Kim Sandberg Bossen	S163290
	Mette L.B. Andersen	S172840
	Rasmus Holm Feldberg	S163591
	Niklaes Dino Robbin Jacobsen	S160198
	Mathias Fager	S175182

Indholdsfortegnelse

1. Indledning	3
2. Analyse	4
2.2 Kravspecifikation	4
2.2.1 Kravspecifikations tabel	4
Krav der er udskudt til senere iterationer	6
2.3 Aktør	6
2.3.1 Aktør table	6
2.4 Interessenter	7
2.4.1 Interessenter table	7
2.5 Use cases	8
2.5.1 Use case	8
2.5.2 Use case diagram	10
2.5.3 Use case matrix	10
2.6 Domænemodel	11
2.7 Risikoanalyse	11
2.7.1 Risiko table	12
2.7.2 Risiko	12
2.7.3 Løsningsforslag til Risiko	13
3. Design	15
3.1 GRASP	15
3.2 Model-View-controller MVC	17
3.3 Design klassediagram	18
3.4 Design sekvensdiagram	20
4. Implementering	21
4.1 Implementeringsværktøjer	21
4.2 Inkrementel programmering.	21
4.3 Kode gennemgang	23
5. Dokumentation	26
6. Konfiguration	26
7. Test	26
7.1 Test af Playerklassen	27
7.2 Test af Die	28
7.3 Test af Fieldklassen	29
7.4 Test af ChanceCardklassen	30
7.5 Test af ViewCTRL	31
8. Projektplanlægning	33

9. Konklusion	33
10. Perspektivering	34
10.1 Design:	34
10.2 Implementering:	34
11. Bilag	35
11.1 Bilag til Timeforbrug	35
11.2 Bilag til mødereferat	35
12. Litteraturliste- og kildefortegnelse	35

Timeforbrug

Navn	Time forbrug
Mette R.	
Mette S.	
Mathias	
Rasmus	
Kim	
Niklaes	OVER 9000 Hahaha :)

Fig 1. Timeforbrug. Her ses time forbruget for hver deltager

Mere detaljeret timeforbrug findes i bilag - 1.

Abstract

The purpose of this assignment, CDIO 4, is to develop a game according to the required specifications following a development strategy according to UP(Unified Process). The way we are doing this is to use the theoretical knowledge we have attained in the 3 interdisciplinary courses. It will comprise of an analysis, a design, an implementation and tests of the given assignment. This time we have had iterations in Inception, Elaboration and the Construction phases. The Transition phase has been left out as we do not have a customer to deliver the game to, and receive feedback from regarding further refinement, but we consider the report writing as our Transition Phase.

1. Indledning

Denne rapport repræsenterer den proces vi gennemgår ved at producere et spil.

Vi skal udvikle et Matador spil hvor vi får stillet en opgave fra en kunde, og skal integrere matador regler i opgaven således det hele ender med et spilbart spil. Spillet skal være et spil med 2-6 spillere. Vi vil starte med en analyse hvor vi kigger på kravspecifikationerne til spillet. Kravene til spillet er defineret ud fra et fysisk matador regelsæt. Vores analysedel vil ligne den som i CDIO 3, med en enkelt UseCase. Vi vil desuden lave kravspecifikationer samt risikovurdering. Vi kommer til at lave Proof of Concept, på forskellige risikofyldte dele af spillet. På baggrund af kravspecifikation og risikovurdering, vil vi identificere de mest risikofyldte dele af projektet, og starte med at designe og implementere disse. Vi vil forsøge at følge Unified Process udviklingsmetoden ved at lave iterationer af en dags varighed, dvs. vi får 14 iterationer indtil aflevering . Da vi har taget udgangspunkt i CDIO 3, og kurset Udviklingsmetoder ikke indgår i projektet længere, er der mere fokus på design, implementering og test. Vi vil bygge videre på strukturen fra CDIO 3, men ikke kopiere koden fra tidligere. Derimod vil vi skrive koden fra bunden, og finde inspiration fra vores tidligere klasser og struktur. Vi vil forsøge at følge bottom-up modellen, hvor vi først vil lave model-laget og lave relevante tests, hvorefter vi vil implementere de forskellige controllere, som skal styre model-laget. Denne gang vil vi også lave en Viewcontroller, som skal tage sig af alt kommunikation med GUI. Vi vil arbejde med risikofyldte dele af projektet som noget af det første, som fx view og GUI.

2. Analyse

I Analysefasen analyseres projektbeskrivelsen. Her identificeres kundens behov og krav til systemet. Her ses på Kravspecifikationer, Aktør, Interessante, domæne model og use cases. I analyse delen af Inception Phase (jvf. *Craig Larman's Applying UML and patterns*, s. 33.) diskuterer vi om projektet er realistisk, samt om der kan arbejdes videre på det efterfølgende, samt lavet risikoliste og vurderet risikofaktorerne samt skabt løsninger. Dette har vi forsøgt at sikre ved grundigt at diskutere kravspecifikationerne, og stille spørgsmål til kunden, så vi kan præcisere kravspecifikationerne i en sådan grad at de ikke er tvetydige, samt at om det er muligt at lave et projekt der kunne måles op mod kravene til CDIO 4.

Vi havde et rigtigt matadorspil, og blev enig om at åbne kassen og beskrive hvad vi så når vi tog objekterne ud af kassen. Vores forståelse af disse objekter og deres brug og sammenhæng, hjalp os til at skabe en abstraktion, vores domæne.

2.2 Kravspecifikation

I følgende fase analyseres kravene fra kunden og derefter systematisk fordeles kravene op i punkter, så vi kan dele dem op i funktionelle og ikke funktionelle krav.

2.2.1 Kravspecifikations tabel

Funktionelle krav
1. Spillet skal kunne spilles af 2-6 spillere 1.1 Spiller modtager 30000 i starten af spillet. 1.2 Spiller skal selv kunne vælge sit eget navn.
2. Spillet spilles på en spilleplade med 40 felter
3. Spiller starter ved start
4. Spilleren kaster med 2, 6 sidet terninger 4.1. Slår spilleren 2 ens får man en ekstra tur.
5. Hvis en spiller ejer alle grundene i samme farve, kan spilleren bygge huse og hotel. Dette skal ske inden spilleren slår med terningerne. 5.1 Før spilleren kan bygge hotel, skal der være 4 huse på grunden.
6. Hvis spilleren lander på en grund, som ingen ejer, kan spilleren købe grunden.
7. Hvis spilleren lander på et felt som en anden spiller ejer, skal spilleren betale leje i forhold til bebyggelsen på grunden. 8. Er der ingen bebyggelse på grunden skal spilleren betale dobbelt leje.

9. Ønsker en spiller at sælge hotel eller huse, får spilleren den halve pris, af købsprisen for et hus, tilbage af banken.
10. Hvis en spiller lander på et chancefelt, trækkes et chancekort, og udfører det som der står.
11. Lander en spiller på De Fængles feltet, eller trækker De Fængles kortet, skal spilleren rykke til fængsel feltet, og spiller modtager ikke 4000, hvis der passerer start.
12. Spilleren kan komme ud af fængsel ved at: A. Betale en bøde på 1000 kr., inden spilleren kaster terningerne. B. Benytte et løsladelseskort spilleren har trukket og gemt.
13. Hvis spilleren lander på skattefeltet, skal spilleren betale hhv. 2000 eller 4000, alt efter hvad der står på feltet.
14. Hver gang spilleren passerer start får spilleren 4000 kr fra banken, medmindre andet er angivet.
15. Hvis en spiller skal betale mere end spilleren ejer i værdier(penge, huse og grunde), er han bankerot og skal overdrage alle sine ejendele til sin kreditor, efter at have solgt evt. bygninger tilbage til banken, og er derefter ude af spillet.
16. Når spilleren sælger en grund til en anden spiller, så er det til den pris der står på brættet.
17. Hvis spillet tvinger en spiller til at betale penge, så går en proces i gang hvor man finder ud af om spilleren har penge nok. Hvis det ikke er tilfældet tvangssælger spillet bygninger, og starter fra den billige ende. Hvis det heller ikke er nok vurderer spillet om spilleren har grunde nok til at kunne betale sin forpligtigelse, hvis det ikke er tilfældet går spilleren konkurs. Hvis det er tilfældet tvangssælger spillet grunde startende i den billige ende indtil spilleren har penge nok på sin konto til at kunne betale sin forpligtigelse.
Ikke-funktionelle krav
18. Spillet skal kunne spilles på en PC i databaren.
19. Maven skal bruges til konfigurationsstyring af GUIen
20. Vores platform skal bestå af et operativsystem, Java og Eclipse, samt bibliotek GUI.jar

Krav der er udskudt til senere iterationer

1. Hvis en spiller ikke vil købe den grund spilleren er landet på, sættes grunden til auktion for de andre spillere og sælges til højestbydende
2. Spilleren kan pantsætte sine grunde
3. Hvis spilleren slår 2 ens 3 gange i træk må man ikke flytte 3. gang, og spilleren skal rykke direkte i fængsel. Spilleren modtager IKKE 4000 kr. hvis man rykker over start.
4. Spilleren kommer ud af fængsel hvis man:
 - a. Hvis spilleren slår 2 ens, hvorefter spilleren straks flytter antal øjne man har slået.
5. Spilleren kan ikke blive i fængsel mere end 3 omgange, hvis spilleren ikke kaster 2 ens 3. Gang, så skal spilleren betale bøden på 1000 kr. og slå med terningerne, hvorefter spillerens brik flyttes summen af øjnene.
6. Spilleren kan kun pantsætte ubebyggede grunde for halvdelen af grundens pris.
7. Hvis spilleren vil hæve pantsætningen, koster det 10% af grundens pris + halvdelen af grundens pris.
8. Spilleren kan ikke kræve leje af en pantsat grund.
9. Spilleren kan sælge sine pantsatte grunde til en anden spiller, men hvis denne spiller vil hæve pantsætningen gælder regel 7.
10. Spilleren kan ikke sælge en pantsat grund til banken.
11. Der skal bygges jævnt dvs. spilleren skal have bygget på alle grundene i en gruppe med 1 hus før spilleren kan bygge 2 huse på en grund.
 - a. Når en spiller vil bygge huse, startes der altid på med den/de grunde med færrest huse på.
12. Hvis man lander på indkomst skattefeltet, kan man vælge at betale med enten 4000 kr eller 10% af sine værdier, grunde og virksomheder, bygninger og kontanter.
 - a. Vi har udskudt 10% reglen, 4000 kr virker.

2.3 Aktør

Aktør er dem der 'hands on' bruger systemet.

2.3.1 Aktør table

Følgende tabel viser hvilke aktører der bruger programmet, hvilken rolle og hvilke formål aktøren har med spillet.

Aktør	Primær	Supporterende	Offstage	Formål
Spiller 2-6	x			At vinde spillet ved at have den største værdi.

Figur 2. Aktør table.

2.4 Interessenter

Interessenter er de personer der er interesseret i projektet.

2.4.1 Interessenter table

Følgende table viser interessenter for vores projekt.

De interessenter for projektet er bruger af spiller og kunde.

Interessenter	Type	Område i projektet, der har interesse	Interessenternes ønsker, krav og mål.	Interessenternes betydning for projektet
Bruger / Spiller	Direkte	Spilleets regler og brugerflade	At spille spillet og brugervenlighed.	
Kunde	Indirekte	Et godt slut produkt, som kan spilles uden problemer.	At have et fungerende spil	Sætter rammerne for spillet igennem kravspecifikationen. Betalder for udviklingen

2.5 Use cases

Når man laver UseCases, afdækkes primær aktørers interaktion med systemet. Use case beskriver funktionen af systemet fra brugernes synsvinkel. Use cases er dermed med til at understrege brugers mål og perspektiv.

Der er en 'main Use case', som har som bruger-mål at spille spillet. Dertil er der lavet subusecases.

(jvf. principper i Craig Larman's *Applying UML and patterns*, Inception s. 61).

2.5.1 Use case

Use case UC1 : Spil Spillet - Fully dressed

Use Case: Spil Spillet
ID: 01
Brief description: At spiller kan spille spillet
Primary Actors: Spillerne, 2-6
Secondary Actors:
Preconditions: <ol style="list-style-type: none"> 1. Brugerne har adgang til databaren, samt har login til databaren. 2. Java og Eclipse er installeret på PC. 3. Minimum 2 spillere 4. Starter spil og indtaster spillere og tilhørende navne til. 5. Begynder ved start
Main Flow: <ol style="list-style-type: none"> 1. Spille sekvens starter med en spillers tur. 2. Spiller slår med terningerne 3. Systemet fortæller spiller, hvilket felt spilleren er landet på. Spiller kan lander på felterne 1-40. <ol style="list-style-type: none"> 3.1 Spiller kan købe grunden, hvis spiller lander på property field hvis dette ikke allerede er ejet. <ol style="list-style-type: none"> 3.1.1 Hvis en spiller lander på et ejet property field, skal spiller betale leje for at lande på feltet. 4. Hvis en spiller ejer alle felter af samme farve, må ejeren bygge huse og hotel <ol style="list-style-type: none"> 4.1 Hvis en spiller lander på et felt med huse eller hotel, skal han betale leje ud fra hvor mange huse eller hotel der står på feltet. 5. Hvis spiller lander på et specialefelt, følges reglen der hører til feltet. 6. Hvis spiller passerer start modtager han 4000 kr. 7. Skifter spilsekvens fra en spiller til anden 8. Spillet spilles indtil en af spillerne er gået bankerot 9. Vinder er den med mest værdi dvs. penge på konto + værdi af ejendomme.
Postconditions: Tabt eller vundet spil
Alternative Flows: <p>Hvis Spiller lander på chancekort Spiller får et chancekort og følger dets regler.</p> <p>Hvis Spiller lander på "De fængsles" feltet Spiller rykker i fængsel, hvis spiller passere start får man ikke 4000 kr.</p> <p>Hvis Spiller lander på et taxfield Spiller betaler enten 2000 eller 4000.</p> <p>Hvis spiller passere start Spiller modtager 4000.</p>

2.5.2 Use case diagram

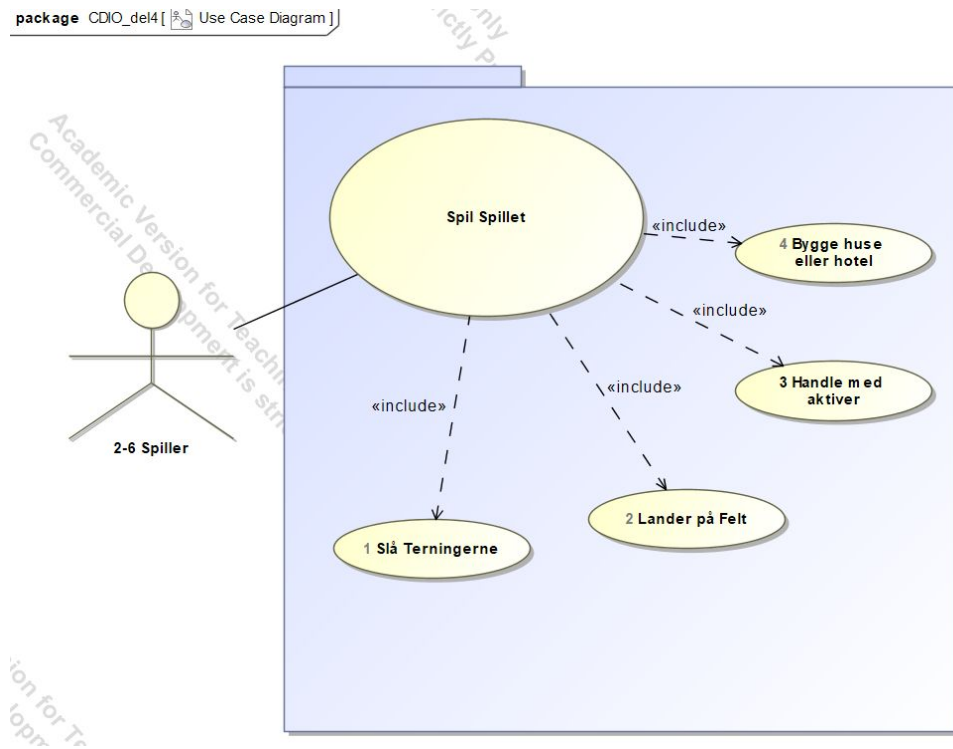


Fig. Use case diagram

2.5.3 Use case matrix

Requirements Traceability Matrix laves for at danne et overblik over hvilke use cases der dækker, hvilket krav. - Use case 01 dækker alle de funktionelle kravspecifikationer

Krav	UC1	SUC 1	SUC 2	SUC 3	SUC 4
K2	x				
K4	x				
K5	x				
K6	x				
K7	x				
K8	x				
K9	x				
K10	x				
K11	x				

Fig, traceability Matrix

2.6 Domænemodel

Domænemodel er et statisk struktureret diagram, der viser hvordan de forskellige klasser i systemet arbejder sammen. Det bruges til at skabe overblik over det system kunden ønsker. Og skabe en ide til hvorledes systemet kan modelleres. (jvf. principper i Craig Larman's *Applying UML and patterns, Elaboration Iteration 1 s. 131.*).

2.6.1 Domænemodel diagram

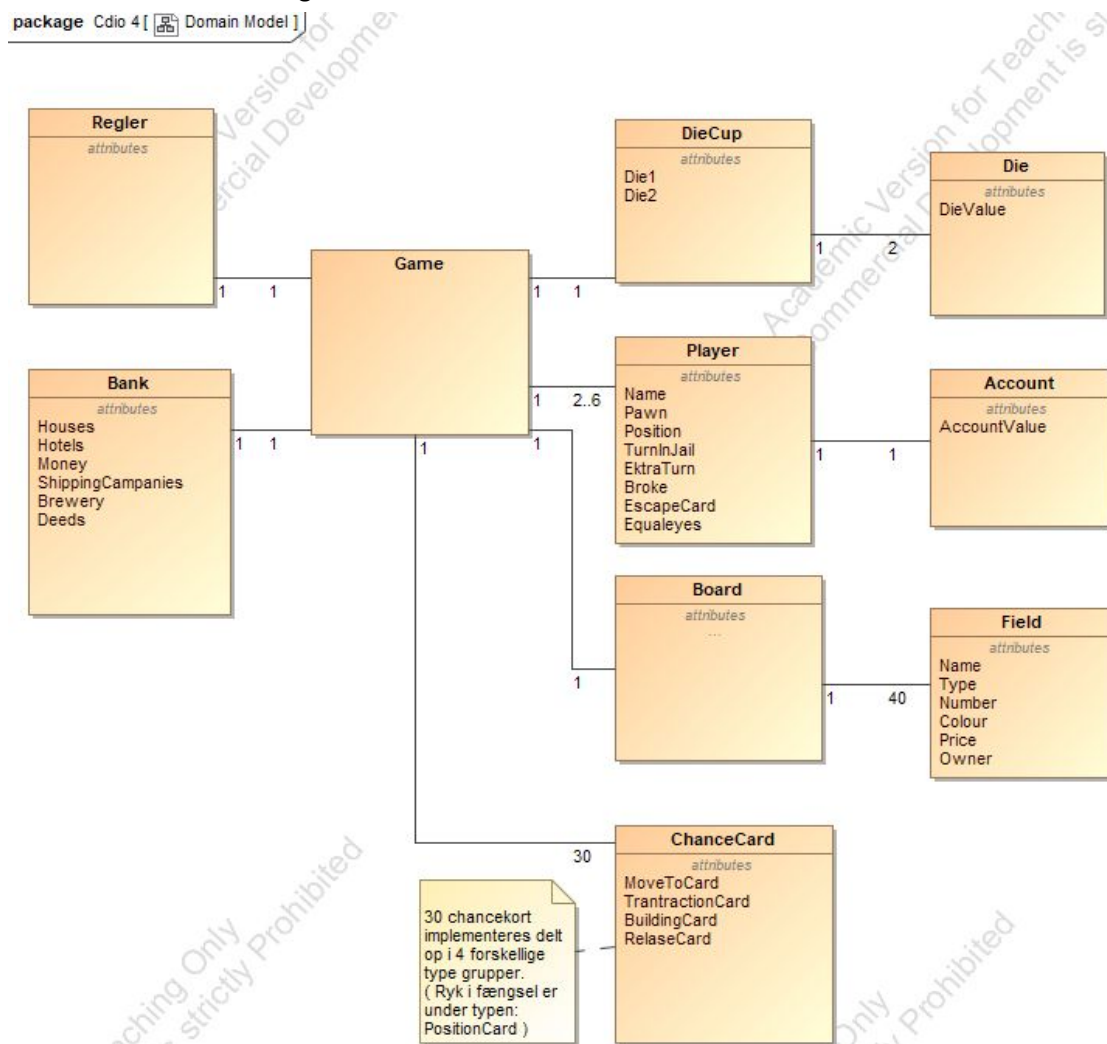


Fig. Domænemodel

2.7 Risikoanalyse

Disse risiko punkter er blevet udarbejdet til at varetage de risici der kan opstå i projektet. Først sættes risici op i punktform og derefter bliver der udarbejdet en risiko tabel i Excel. Det gæses i dybden med de kritiske risiko, og løsningsforslag skrives under punkterne. Risikoanalyse er til for at kunne identificere de steder i vores projekt både i design og implementeringen, hvor vi ser på de største forhindringer, så vi kan få udarbejdet de mest risikofyldte først. (jvf. principper i Craig Larman's *Applying UML and patterns, Introduction s. 27.*).

2.7.1 Risiko table

I tabellen er der påvist med farve indikatorer, hvor stor skaden er.

Vandret ses hvor stor skaden er fra intervallet 0-10. Horisontalt vises sandsynligheden for skaden sker fra intervallet 0-1 med et decimaltal.

		Hvor stor er skaden										
		0	1	2	3	4	5	6	7	8	9	10
Sandsynlighed for at det sker	0											
	0,2				4	5						
	0,4						6					1
	0,6							2	7			
	0,8								3			
	1											

Fig. Risiko tabel for projektet. Vandret ses hvor stor skaden er og horisontalt vises sandsynligheden for skaden sker.

2.7.2 Risiko

De overvejende Risiko for projektet er at:

1	Løber tør for tid
2	At vi ikke har fået afstemt eller forstået kravene omkring de grundlæggende/kritiske elementer i projektet.
3	At kunden ændrer de grundlæggende kravspecifikationer undervejs.
4	At vi ikke har fået afstemt kravene omkring de mindre kritiske elementer i projektet
5	Github går ned og vores data forsvinder/ikke tilgængelig
6	At vi ikke kan finde ud af at lave de kritiske dele af projektet i inception-fasen.
7	At vi ikke får atomiseret softwaren i tilstrækkelig grad, således at vi kan skabe et spil, med alle de undtagelser som matador indeholder.
8	At GUI ændrer sig radikalt under udviklingsprocessen.

2.7.3 Løsningsforslag til Risiko

Her udregnes på data for at finde den største risici ud af de 8 overvejende punkter. Risikofaktoren findes ved ligningen :

$$\text{Risikofaktoren} = \text{Hvor stor er skaden} \times \text{Sandsynlighed for at det sker}$$

Risici	
1	Løber tør for tid
	1: risikofaktor = $10 * 0,4 = 4$
	At vi ikke kan overholde vores iterationer og projekt løber fra os.
	Løsning:
	Vi holder overblik, ved hjælp af møder
	Vi prioriterer benhårdt funktionalitet således at vi ender med enkelte færdige funktioner i stedet for flere ufærdige.
	At vi bruger hjælpelærere når det er nødvendigt.
2	At vi ikke har fået afstemt eller forstået kravene omkring de grundlæggende/kritiske elementer i projektet.
	2: risikofaktor = $6 * 0,6 = 3,6$
	At vi ikke har forstået hvilke artefakter kunden vil have.
	At vi ikke gennemskuer hvilke elementer der er de kristiske.
	Løsning:
	Vi tager kontakt til hjælpelærere og undervisere og bliver helt klar over hvad de vil have.
	Vi diskuterer vores struktur ud fra hvordan man spiller spillet hvis man sidder omkring det.
3	At kunden ændrer de grundlæggende kravspecifikationer undervejs.
	3: risikofaktor = $7 * 0,8 = 5,6$
	Det kunne være at man pludseligt ville have andre eller flere artefakter end dem vi har lært at bruge."BCE"
	At man pludseligt bestemmer at man vil have at vi skal bruge en eks. ver 4.0 af gui der opfører sig helt anderledes end den vi har.
	At der sent kan komme krav omkring design som går imod det vi har lavet.
	Løsning:
	Vi være på forkant med hvad de vil have.
	Vi vil argumentere for at vi ikke vil arbejde med noget vi ikke har lært.
	Vi må lære den nye GUI.
	Nye krav ville kunne blive prioriteret væk pga. tid, ellers må vi be om mere tid hvis de SKAL implementeres.
4	At vi ikke har fået afstemt kravene omkring de mindre kritiske elementer i projektet
	4: risikofaktor = $3 * 0,2 = 0,6$
	At vi ikke har fået tildelt ansvaret godt nok til de forskellige klasser.

	<i>Løsning:</i>
	En god design-fase.
	Møderne.
5	Github
	<i>5: risikofaktor = 4 * 0,2 = 0,8</i>
	Manglende forståelse for problemer med Github
	Opstående fejl
	<i>Løsning:</i>
	Kommunikation med gruppe, ift. sparing af viden
	Vi har en Github Master som kan løse problemerne for de andre i gruppen.
6	At vi ikke kan finde ud af at lave de kritiske dele af projektet i inception-fasen.
	<i>6: risikofaktor = 5 * 0,4 = 2</i>
	At vi ikke kan finde ud af GUI i tilfredsstillende grad således vi kan lave en VIEW controller.
	At vi ikke kan finde ud af at arbejde med Jagged Arrays således at vi kan lave grupper.
	At vi ikke kan finde ud af at implementere Aktion og Chance Controller.
	<i>Løsning:</i>
	Vi undersøger enhver løsning vi bruger, for at være sikker på, at der ikke er noget vi roder os ud i, som vi ikke kan håndtere.
	At vi finder en alternativ måde at forholde os til gruppering af felter eks. 2 felter frederiksberg og rådhuspladsen hører sammen etc.
	Vi undersøger GUI, og laver prototype på den
	Hvis vi ikke kan implementere Aktion og Chance controlleren, betyder det at vi ikke kan løse kompleksiteten af reglerne med vores nuværende struktur, og så må vi enten lave om på strukturen eller bortprioritere reglen
7	At vi ikke får atomiseret softwaren i tilstrækkelig grad, således at vi kan skabe et spil, med alle de undtagelser som matador indeholder.
	<i>7: risikofaktor = 7 * 0,6 = 3,2</i>
	At vi får lavet for store klasser og metoder så vi ikke kan udføre den detaljegrad af undtagelser og specielle tilfælde der er i spillet. Eks. man skal kunne sælge huse og grunde før man slår, men man skal kunne gøre det samme hvis man skal betale og man ikke har kontanter nok, men har aktiver man kan sælge for at få kontanter nok.
	<i>Løsning:</i>
	Ved at lave statiske metoder
	At virkeligt atomisere processerne i designfasen, således vi får den rigtige grad af samhørighed og binding, samtidig med at vi kan genbruge funktionalitet forskellige steder i controll.

Fig. Løsningsforslag til Risiko

3. Design

Vi kunne nu skabe en endnu mere detaljeret abstraktion, design klasse diagrammet, på baggrund af domænet.

Vi har taget udgangspunkt i designet fra vores CDIO3. Vi er en gruppe der består af medlemmer fra 3 grupper, og 2 af de tidligere designs mindede om hinanden, derfor valgte vi som vi gjorde. Designet tager udgangspunkt i MVC, hvor vi har et model, et view og et controller lag. Vi har forsøgt at lave så lav en binding mellem view, control og model og control som muligt. Dette for at skabe et design hvor man forholdsvist smertefrit kan skifte enkeltdelene ud uden det betyder for meget for resten af konstruktionen. Control-klasserne har en lidt højere binding til hinanden end de har til view og model, dette er af implementerings årsager og vil være beskrevet i dette afsnit.

3.1 GRASP

(General Responsibility Assignment Software Patterns)

GRASP er et design mønstre vi har brugt til at hjælpe med at identificere hvilke ansvar, der skal tildeles vores klasser/objekter i vores spil og hvordan de interagerer med hinanden.

Vi har under designet talt om lav binding og høj samhørighed som er to principper der tilhører GRASP principperne som skal overholdes for at skabe så nemt læseligt, vedligeholdelsesvenligt og robust kode.

Vi har kigget på 5 design mønster:

Creator:

Creator klasser i vores system er Startgame der opretter vores ActionCTRL hvor alle vores objekter bliver oprettet af de klasser der skal bruge dem for at kunne udføre deres funktion (*jvf. principper i Craig Larman's Applying UML and patterns, Elaboration Iteration 1 s. 281-282.*).

Information Expert:

Information Expert klasser i vores spil er klassen **[klasserne skrives]** er information Expert.

For at kunne få tildelt et ansvar og udføre en opgave skal man have den rette information. Så en Information Expert Klasse er en klasse der har den information, der skal bruges for at kunne leve op til det tildelte ansvar. Nogle klasser skal samarbejde med en andre for at kunne fuldføre deres ansvar. (*jvf. principper i Craig Larman's Applying UML and patterns, Elaboration Iteration 1 s. 284.*)

Eksempelvis:

Klassen Boardt samarbejder med superklassen Field, som nedarver til subklasser. Board-klassen nedarver til forskellige field-typer som overskriver metoderne i board-klassen, derudover indeholder nogle af subklasserne også sine egne metoder udover dem som er nedarvet fra board-klassen.

Objektet Player har informationer om sin egen position og om spilleren har fået en ekstra tur. Player-klassen samarbejder med account-klassen for at kunne fuldføre sit ansvar for at kunne lave transaktioner med TradeCTRL.

Man kan argumentere for at de klasser der udgør vores model, at hvis disse samtidig håndterer et ansvar, så er de Information Experts.

Low Coupling

Lav kobling hjælper os med at gøre systemet vedligeholdelsesvenligt samt at sikre kodekvalitet. Vi har designet sådan at klasserne passer til det ansvar som de antyder, eks. Diecup indeholder die1 og die2 som bliver rystet i diecup samt hvad terningerne slår, som henter die1 og die2 fra Dieklassen. Account kender sin balance og Player kan give den videre til TradeController. Dette gør at vi kan udskifte et af elementerne i kæden forholdsvis smertefrit, så længe vi sørger for at have det samme metodesæt, input og output..(jvf. principper i *Craig Larman's Applying UML and patterns, Elaboration Iteration 1 s. 286*).

Vi har Information Experts, Creators og Controllere, hvilket hjælper os til at have så lav en kobling som muligt. Derudover bruger vi MVC(Model View Creator), til yderligere at skabe lav kobling, og dermed adskille grafical user interface og model, og skabe relevante controllere imellem view og modellag.

Controller

Vi har 8 Control klasser ActionCTRL, BankruptcyCTRL, ChanceDeckCTRL, DropdownCTRL, JailCTRL, LandOnFieldCTRL, TradeCTRL og WinnerCTRL. Vi startede med at ligge alt logikken i ActionCTRL, men fandt ud af længere henne i processen, at ActionCTRL blev alt for bloated. Efter vores møde med vores hjælpelærer delte vi vores ActionCTRL ud på flere controllere, hvilket gav os en bedre struktur og en nemmere måde at kunne forstå vores system.

High Cohesion

Vi har forsøgt at skabe høj samhørighed i klasserne, så der er en rød tråd imellem klassernes navne, attributter, samt metoder, altså at klasserne har et veldefineret ansvar som bruges som en rød tråd til at skabe velovervejede navne på klasserne, samt at overveje hvilket ansvar klassen har og derudfra lave attributter og metoder som giver mening under klassens navn.

3.2 Model-View-controller MVC

Før selve programmeringen af spillet har vi gjort os nogle overvejelser ift design mønsteret Model-view-controller MVC.

MVC formidler et billede af systemet. Selvom systemet er sammenhængende er det vigtigt at kunne forstå afgrænsningen/adskillelse af klasserne ved at dele klasserne op i model, view og controller. Dette er en måde, på et mere overordnet plan, at strukturere klasserne, sådan man under design kan være mere præcis i tildelingen af ansvar. (jvf. Ian Bridgwood undervisningssides lektion 7, side 28-29) .

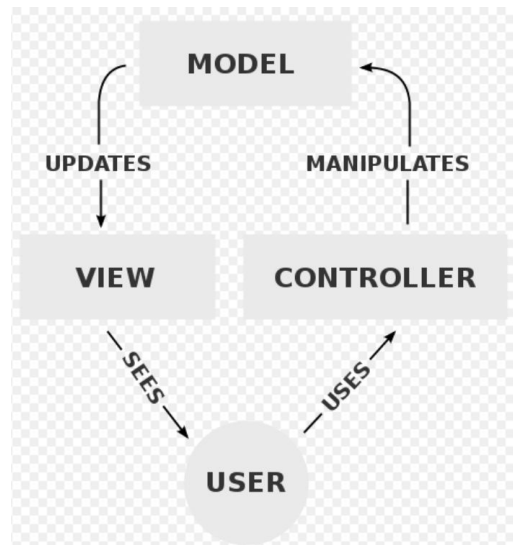


Fig.

Model:

I model-laget findes de klasser der genererer - og opbevarer data. Klasserne har til opgave at informere Control-laget. Model-laget består af DieCup, Die, Player, Account, Board, Field, ChanceCard samt ChanceDeck. Vi har en nedarvning fra ChanceCard til subklasserne TaxCard, TransactionCard, MoveToCard og ReleaseJailCard. Superklassen Field nedarver til TaxField, GoToJailField, NoactionField og NoActionField. Udover det har vi lavet en ny superklasse OwnerFields som er nedarvet fra superklassen Field. Fra superklassen Ownerfields har vi en nedarvning til PropertyFields, ShippingFields og BreweryFields.

View :

I view-laget sendes der forespørgsler fra bruger til Control-laget. Det er her hvor brugeren interagerer og forholder sig til systemet. Som view klasse har vi implementeret GUI klassen, der står for det grafiske interface til brugeren.

Controller:

I Control-laget ligger de klasser, der manipulerer med model-laget og indeholder logikken i spillet. Controller-laget interagerer med model-laget ved at opdatere (*modtage informationer*) og interagere med View-laget ved at opdatere klassen (*modtager forespørgsler fra bruger*). Dette sker i

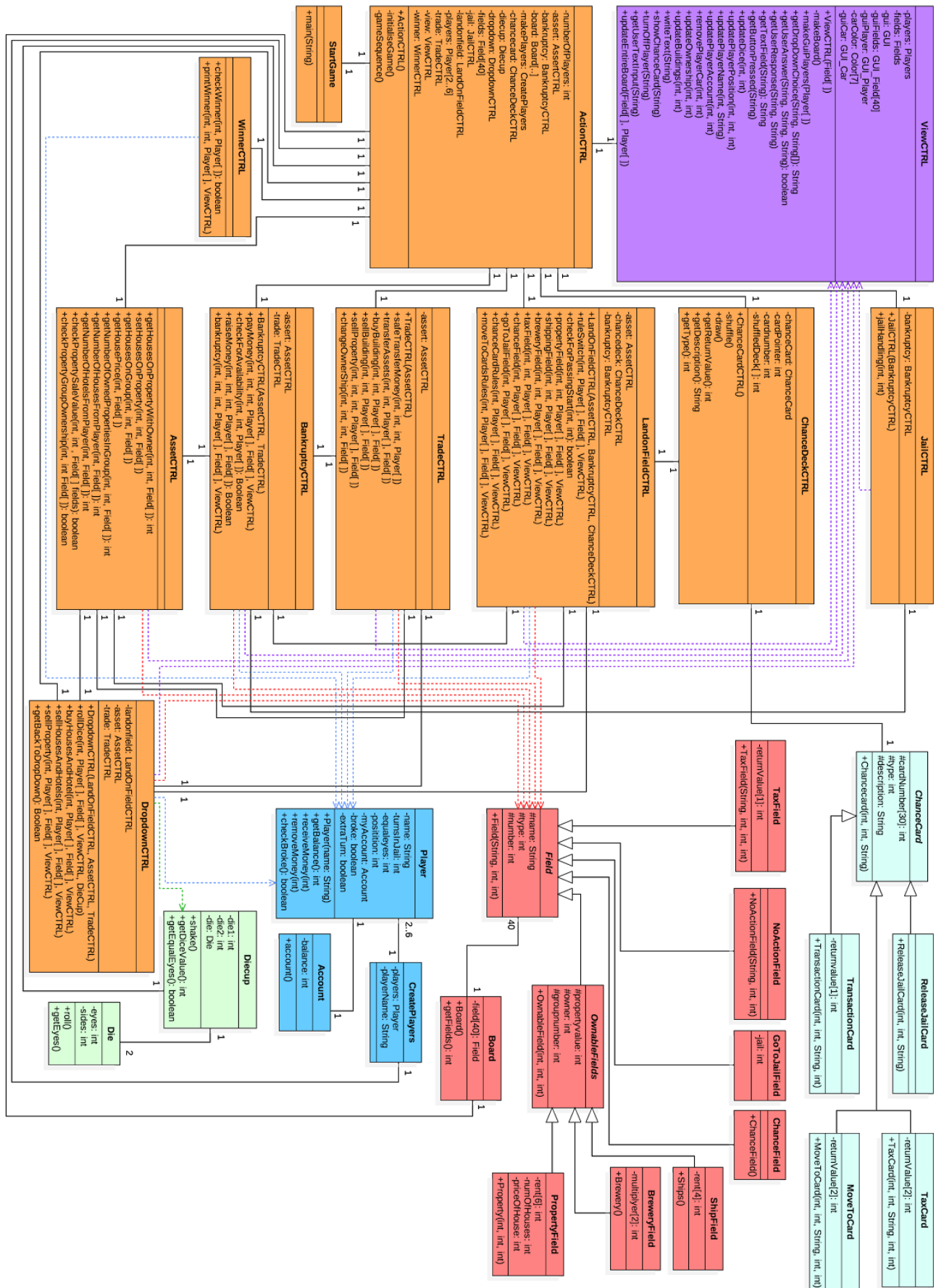
ActionCTRL og ChanceCradCTRL. Derudover har vi implementeret en viewCTRL der håndterer alle GUI-kald samt opdateringer, og en StartGame som har til formål at initialisere spillet.

3.3 Design klassediagram

I følgende diagram vises en statisk struktur af systemet som inkludere klasserne og deres attributter, metoder og relationerne mellem objekterne.

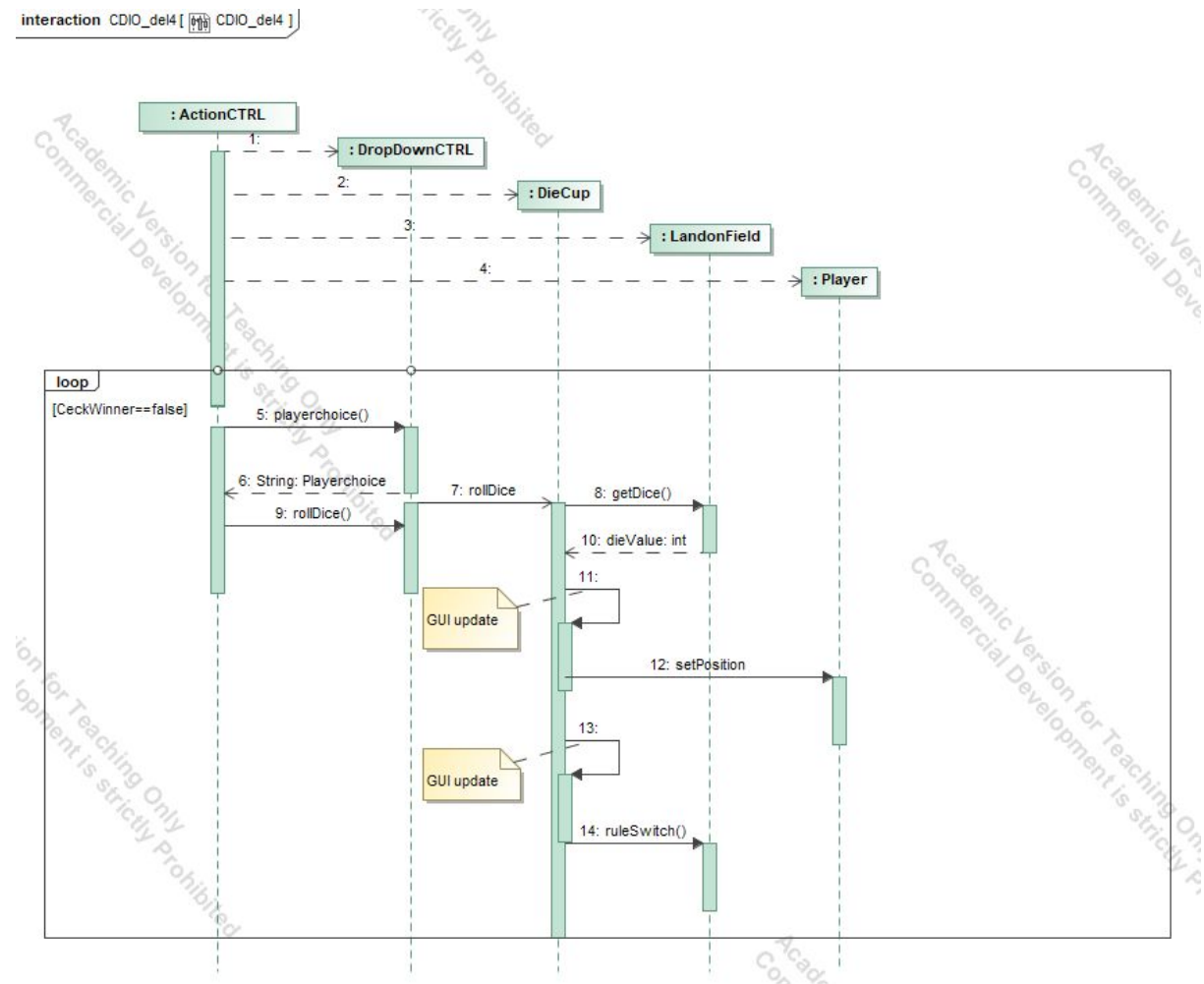
Diagrammet er blevet videreudviklet fra domænemodel i analysedelen, vi har udspecificeret attributter og tilføjet metoder. Dette er blevet udarbejdet samtidig med design sekvensdiagrammet for at hente inspiration fra hinanden.

(jvf. principper i Craig Larman's Applying UML and patterns, Elaboration Iteration 1 s. 249.).



3.4 Design sekvensdiagram

Et Design sekvensdiagram er et dynamisk diagram, der viser interaktion mellem de forskellige objekter i systemet. Design sekvensdiagram inkluderer objekterne og deres attributter, metoder og relationerne mellem objekterne.



4. Implementering

4.1 Implementeringsværktøjer

I vores inceptionphase bestemte vi i vores configurationdisciplin hvilke værktøjer vi ville bruge. Vi lavede først et nyt remoteGit på GitHub til versionsstyring af koden. Vi har i gruppen valgt at bruge GitKraken da det har et overskueligt interface, det gør konflikthåndtering nemt, og det danner et fint overblik over versionsstyringen på GitHub. Desuden bruger vi Maven til konfigurationsstyring i forhold til GUI, Java og JUnit testing.

4.2 Inkrementel programmering.

UP går ud på at fragmentere arbejdsprocessen i små iterative dele hvor vi hver gang revurderer krav, risikovurderer, ser på analyse og design, og derefter implementerer.

Vi startede med at programmere model-laget først. Vi gik igennem alle modelklasserne på tavlen, således alle var med i processen med at skabe designklasser ud fra domænemodellen. I den proces talte vi også en smule om controllerproces, fordi en modelklasses attributter er afgørende for hvorledes man benytter den i Controllerne. Vi forholdt os også til abstraktionen af vores forståelse af domænet således vi bestemte ansvar for de forskellige klasser. Dernæst delte vi modellen med tilhørende branch, ud på hver medlem, til implementering.

Vi lavede en branch for hver model der indeholdt nedrivning, og derudover fik hver klasse sin egen branch. Modellen blev merget sammen som den blev færdig

I de næste iterationer valgte vi at håndtere programmering af controller-klasserne, med fokus på først at få ViewControlleren(VC), StartGameController og GameController (GC) lavet. I denne process revurderede vi vores design og fik slået GC sammen med ActionController(AC) klassen. Vi beholdte vores main metode i StartGameController. I samme iteration lavede vi en ny klasse, toolbox, som var en delegate controller til AC klassen, hvor vi uddelegerede opgaver fra AC til toolbox klassen for at undgå AC blev for oppustet . Alle metoder der blev placeret i toolbox var metoder der blev genbrugt flere gange i AC, eller hjælpe metoder til toolbox

Vi mergede Toolbox, StartGame, VC med modellen og håndterede den merge konflikt der opstod.

Midt i implementeringsfasen valgte vi at programmere AC klassen og dele os op i teams, hvor hver især fik en del af AC at lave. Vi mergede klassen sammen med de andre dele af spillet. I sidste iteration var alt logikken kodet og integrationsprocessen gik i gang med bughunting med indbygget debugging værktøj i Eclipse, samt forskellige udgaver af fields objektet. Disse udgaver ligger stadig i Board klassen så i kan se hvorledes de ser ud. Vi valgte at bruge en teamviewer og skridt for skridt få vores applikation til at virke.

M2:

Efter vores M2 revurderede vi analyse og design og valgte at aflaste vores AC klasse, og lavede 6 nye controller klasser: DropdownCTRL, JailCTRL, LandonFieldCTRL, TradeCTRL, WinnerCTRL og BankruptcyCTRL, som alle har adgang til view controller. Dette gjorde vi fordi vores AC var meget lang og uoverskuelig. Toolbox havde også fået for stort et ansvarsområde, og vi valgte at skille ansvar ud til TradeCTRL.

Intet kode er genbrugt fra CDIO 3, alt er programmeret fra bunden og det valgte vi at gøre på baggrund af ikke at skulle ende med at have et projekt hvor vi samlede og tilrettede kode, men et projekt hvor vi arbejdede os på nedefra og byggede systemet inkrementelt fra bunden. Derfor valgte vi istedet at benytte de gamle klasser til inspiration fremfor kopiering.

Vi har arbejdet med arrays istedet for arraylists, dette var en udfordring i forhold til dynamisk at ændre størrelse og indhold i arrayet. Det er vi kommet udenom ved at gennemløbe brættet mange gange for at identificere størrelse på array samt indhold på array.

Eks. når du skal sælge en grund kommer en liste over grunde der er dine og som ikke indeholder huse på gruppen. En gruppe er eks. Hvidovrevej og Rødovrevej. I denne gruppe er der 2 grunde.

Den liste ændrer sig alt efter hvad du gør i spiller menuen, og derfor skal den opdateres med modellagets nye data. Dette gøres ved at brættet gennemløbes hver gang option om at sælge grund vælges.

Dette betyder mange gennemløb i løbet af et fuldt spil, men i dette tilfælde, med så lille en datamængde, er det ikke kritisk for afviklingen af spillet.

Vi har delt controller ansvar ud på flere delegate controllere efter konsultation med hjælpelærer Christian. Da controllerne bruger hinanden rigtigt meget på kryds og tværs, har vi bundet controllerklasserne sammen i en stærk binding.

Dette var et valg mellem pest eller kolera. Enten havde vi en bloated AC, eller også måtte vi lave det design vi har nu. Vi kunne godt have sendt de relevante controllere via parametre i metode kaldene, men det betød at metodekaldene fik vanvittigt lange parameterlister. Vores løsning var at binde controller klasserne, således at en controller initialiserede de andre controllere den skulle bruge som attributter, og når vi instantierede en controller, så blev referencen til de relevante controllere via konstruktøren lagt ind i controller-objektets attributliste.

Derimod er modellens dataobjekter fields og players samt viewobjektet view kun til udlån, da de bliver sendt som parametre hvor det er relevant.

Vi har arbejdet med alle de teknikker vi har lært på nær fil I/O. Dette fordi vi var presset i tid, og måtte prioritere.

4.3 Kode gennemgang

Kode gennemgang af konkret situation "En spiller lander på et chancefelt og trækker et transactionCard".

Forklaring er skrevet med grønt og de linjer koder der bliver kørt under program eksekveringen er skrevet med kursiv.

```
public void ruleSwitch (int currentPlayer, Player[] players, Field[] fields, ViewCTRL view) {  
    Felttypen findes.  
    int fieldType = fields[players[currentPlayer].getPosition()].getType();  
    int owner = 0;  
    Hvis et felt er et OwnerFields, så bestemmes ejeren af feltet, således denne kan sendes med  
    ind i metodekaldet til de 3 typer af OwnerFields der er.  
    if (((fields[players[currentPlayer].getPosition()]) instanceof OwnerFields) {  
        owner = (((OwnerFields)fields[players[currentPlayer].getPosition()]).getOwner());  
    }  
    //Feltreglen bestemmes, og kaldes  
    switch (fieldType) {  
        case 0: //PropertyField (OwnerFields)  
            propertyField(currentPlayer, owner, players, fields, view);  
            break;  
        case 1: //ShipFields(OwnerFields)  
            shippingField(currentPlayer, 1, players, fields, view);  
            break;  
        case 2: //BreweryFields(OwnerFields)  
            breweryField(currentPlayer, owner, players, fields, view);  
            break;  
        case 3: //TaxFields  
            taxField(currentPlayer, owner, players, fields, view);  
            break;  
        case 4: //ChanceField  
            chanceField(currentPlayer, players, fields, view);  
            break;  
        case 5: //StartField. View opdateres her, da der ikke er nogen metode til dette felt.  
            view.writeText(players[currentPlayer].getPlayerName() + " er landet på " +  
                fields[players[currentPlayer].getPosition()].getName() + "");  
            break;  
        case 6: //NoActionField. View opdateres her, da der ikke er nogen metode til dette felt.  
            view.writeText(players[currentPlayer].getPlayerName() + " er landet på " +  
                fields[players[currentPlayer].getPosition()].getName() + "");  
            break;  
        case 7:  
            //GoToJailField.  
            goToJailField(currentPlayer, players, fields, view);  
            break;  
        Default: //Hvis der opstår en fejl ved bestemmelsen af en felttype  
            System.out.println("Felt-typen der pharses er ikke korrekt.");  
    }  
}
```

```
}

```

Feltet er bestemt til et chancefelt, og reglen for chanceField bliver udført.

```
public void chanceCardRules (int currentPlayer, Player[] players, Field[] fields, ViewCTRL view) {
    //Finder typen af chancekort 1-4
    int chanceCardType = chancedeck.getType();
    //Valuearray[?]. Heri ligger forskellige værdier som metoden skal bruge for at
    implementere //chancekortet på modellaget.
    //For TransactionCard er valueArray[0] = transaktionsværdien
    // For MoveToCard, er valueArray[0] = typen af Moveto
    Absolut = 1.
    Nærmeste rederi = 2
    Fængsel = 3
    Ryk 3 felter tilbage = 4

    //valueArray[1] = feltnr man flytter til for de absolutte flyt.

    int[] chanceCardValueArray = chancedeck.getReturnValue();
    int owner = 0;
    //hvis det er et Ownerfelt så hent ejeren.
    if (((fields[players[currentPlayer].getPosition()]) instanceof OwnerFields) {
        owner = (((OwnerFields)fields[players[currentPlayer].getPosition()]).getOwner());
    }
    //Bestem typen
    switch (chanceCardType) {
    case 1: // TransactionCard
        //Transaktionsværdien findes.
        //Her kunne man godt bare have brugt chanceCardValueArray[0] i det
        //efterfølgende, men for at gøre koden læsbar, vælger vi at ligge værdien over i en
        //beskrivende variabel.
        int transactionValue = chanceCardValueArray[0];
        //Bestem om det er et negativt tal, for så skal pengene trækkes fra konto
        if ((transactionValue) < 0) {
            //realVærdien beregnes
            int realvalue = (transactionValue * (-1));
            //informations tekst til spiller
            view.setText("Der trækkes " + realvalue + "kr. fra " +
                players[currentPlayer].getPlayerName() + "'s konto.");
            //Da der er tale om et ForcePay, altså en betaling vi ikke som spiller kan
            //kontrollere og som er obligatorisk, så går en proces i gang med at finde
            //ud af om spilleren går bankerot pga. Betalingen.
            bankruptcy.payMoney(currentPlayer, owner, realvalue, players,
                fields, view);
        } else { //Hvis det ikke er et negativt tal, altså spilleren får penge.
            //Tekst besked til spilleren.
            view.setText("Der tilføjes " + transactionValue + "kr. til " +
                players[currentPlayer].getPlayerName() + "'s konto.");
            //Opdater Modellag
            players[currentPlayer].recieveMoney(transactionValue);
        }
        //Opdater Viewlag
        view.updatePlayerAccount(currentPlayer)
    }
}
```



```
        break;

    case 2: // MoveToCards
        // logik og viewCTRL-kald ligger i denne metode. Her håndteres hvilken type
        // MoveTo regel der skal implementeres, forklares ikke yderligere.
        moveToCardsRules(currentPlayer, players, fields, view);

        break;

    case 3: // ReleaseCards
        // Her får spilleren et release card
        players[currentPlayer].addReleaseCard();
        // Besked til spiller.
        view.setText(players[currentPlayer].getPlayerName() + " har nu 1
            løsladelseskort mere, og totalt " +
            players[currentPlayer].getReleaseCard() + "kort.");
        break;

    case 4: // TaxCards her er chanceCardValueArray[0] skatten på huse, [1] skatten på hoteller
        // Bestem vedhjælp af asset objektet hvor mange huse og hoteller spilleren har.
        int numberOfhouses = asset.getNumberOfHousesFromPlayer(currentPlayer,
                                                                    fields);

        int numberOfhotels = asset.getNumberOfHotelsFromPlayer(currentPlayer, fields);
        // Beregn total sum til betaling
        int totalSum = (chanceCardValueArray[0] * numberOfhouses) +
            (chanceCardValueArray[1] * numberOfhotels);
        // Besked til spiller om beløb
        view.setText("Der trækkes " + totalSum + "kr. fra " +
            players[currentPlayer].getPlayerName() + "'s konto.");
        // Da der er tale om et ForcePay, altså en betaling vi ikke som spiller kan
        // kontrollere og som er obligatorisk, så går en proces i gang med at finde // ud af
        // om spilleren går bankerot pga. Betalingen.
        bankruptcy.payMoney(currentPlayer, owner, (totalSum), players, fields, view);
        Break;

    default: // Hvis der er fejl i datastruktur
        System.out.println("ChanceCard-typen der parses er ikke korrekt.");
}
}
```

5. Dokumentation

Vi har via et værktøj i Eclipse kort en code coverage, som kan ses [her](#), dog så blev der ikke fortsættet til at en spiller taber, så den del af koden som omhandler at springe en tabt spiller over, opdaterer GUIen i forhold til det etc. er ikke en del af denne coverage, så det ses som ikke kørt kode. Dog så er vi tilfredse med denne code coverage, eftersom en stor procent af klasserne bliver brugt i forhold til den mængde forskellige muligheder spillet kan tage. Næsten hele model laget bliver brugt, hvilket er godt, eftersom det er der alt vores data lægger. Desuden har vi en rigtig god coverage af vores controllers med et gennemsnit på hele 86 procent, hvilket betyder at selvom der er mange forskellige kombinationer af muligheder i de controllers, bliver de alligevel brugt effektivt, og der er ikke spildt kode.

6. Konfiguration

I følgende afsnit er der ønske fra kunden om at vide, hvilke krav der er til styresystem og installerede programmer. Linket under viser en beskrivelse af dette, sammen med en vejledning i hvordan kildekoden compiles, installeres og afvikles.

<https://drive.google.com/file/d/1xScD4wqbyUgXFwCPsG3x4zbszystUtng/view?usp=sharing>

7. Test

Vi har foretaget JUnit tests på store dele af modellaget, samt lavet en test situation for ViewCtrl, således vi kunne teste om vi kunne manipulere GUI korrekt gennem vores egen controller.

Derudover har vi lavet forskellige modeller i Board for spillebrættet, så det blev lettere at lave tests og bughunt af Controllerne. Eks. har vi en udgave af spillebrættet hvor spiller 1 ejer det hele og der er mindst 1 hus på hver PropertyField. Dette gjorde det meget lettere for os at teste bankerot, køb og salg af huse og grunde.

7.1 Test af Playerklassen

Test af konstruktør

```
12 import static org.junit.Assert.*;
13 public class PlayerTest {
14     Player playerTest = new Player("Adam");
15     @BeforeClass
16     public static void setUpBeforeClass() throws Exception {
17     }
18     @AfterClass
19     public static void tearDownAfterClass() throws Exception {
20     }
21     @Before
22     public void setUp() throws Exception {
23     }
24     @After
25     public void tearDown() throws Exception {
26     }
27     /**
28      * Konstruktør skal ikke testes.
29      */
30     @Test
31     public void testPlayer() {
32         //fail("Not yet implemented");
33     }
34 }
```

Test af GetBalance, ReciveMoney og RemoveMoney

```
44 @Test
45 public void testGetBalance() {
46     int expected = 30000;
47     int actual = playerTest.getBalance();
48     assertEquals(expected, actual);
49 }
50 /**
51  * Tester recieveMoney ved at lægge 5000 til spillerens konto og derefter køres en getBalance for at se om spilleren har 35000.
52  */
53 @Test
54 public void testRecieveMoney() {
55     playerTest.recieveMoney(5000);
56     int expected = 35000;
57     assertEquals(expected, playerTest.getBalance());
58 }
59 /**
60  * Tester removeMoney, først trækkes 5000 fra spillerens pengebeholdning på de 30000, og derefter bruges getBalance til at se om
61  * spillerens pengebeholdning nu på de 25000 som kommer fra 30000-5000.
62  */
63 @Test
64 public void testRemoveMoney() {
65     playerTest.removeMoney(5000);
66     int expected = 25000;
67     assertEquals(expected, playerTest.getBalance());
68 }
69 }
```

Testresultat

Finished after 0,078 seconds

Runs: 4/4 ✖ Errors: 0 ✖ Failures: 0

test.PlayerTest [Runner: JUnit 4] (0,000 s)	Failure Trace
testRecieveMoney (0,000 s)	
testRemoveMoney (0,000 s)	
testGetBalance (0,000 s)	
testPlayer (0,000 s)	

7.2 Test af Die


Gennemløb 100000 gange.

```
1 package test;
2
3 import static org.junit.Assert.fail;
4
5
6 public class DieTest {
7
8
9     model.Die die = new model.Die();
10
11     /**
12      * testRoll() - Tester om roll metoden fungerer.
13      */
14     @Test
15     public void testRoll() {
16         int countd1=0, countd2=0, countd3=0, countd4=0, countd5=0, countd6=0;
17
18         for (int i = 0; i<100000; i++) {
19             die.roll();
20             switch(die.getEyes()) {
21                 case 1: countd1++;
22                 break;
23                 case 2: countd2++;
24                 break;
25                 case 3: countd3++;
26                 break;
27                 case 4: countd4++;
28                 break;
29                 case 5: countd5++;
30                 break;
31                 case 6: countd6++;
32                 break;
33                 default: fail("Virker ikke");
34                 break;
35             }
36         }
37     }
38 }
```

Test om terning ikke afviger med mere end 4%

```
36     }
37     // Vi tester om terning er cirka lige hyppigt med en afvigelse på 4%
38     int test = (int)(100000*16.6667)/(100); //1/6 af størrelsen på testen
39     int borderUp = (int)(test + (test*4)/(100));
40     int borderDown = (int)(test - (test*4)/(100));
41
42     //Dump programmet hvis hvert tilfælde er ude for den øvre og nedre grænse
43     if (countd1 > borderUp || countd1 < borderDown || countd2 > borderUp || countd2 < borderDown ||
44         countd3 > borderUp || countd3 < borderDown || countd4 > borderUp || countd4 < borderDown ||
45         countd5 > borderUp || countd5 < borderDown || countd6 > borderUp || countd6 < borderDown){
46         fail("Hyppigheden af de forskellige forekommer ikke lige ofte");
47     }
48 }
49 }
50 }
51 }
```

Testresultat



The screenshot shows the JUnit test runner interface. At the top, it says "Finished after 0,081 seconds". Below that, a summary bar shows "Runs: 1/1", "Errors: 0", and "Failures: 0". A green progress bar is visible. The test list shows "test.DieTest [Runner: JUnit 4] (0,000 s)" and "testRoll (0,000 s)". A "Failure Trace" tab is also visible.

7.3 Test af Fieldklassen

Test af oprettelse af et felt

```
26  @After
27  public void tearDown() throws Exception {
28  }
29
30  @Test
31  public void testField() {
32      Field field = new Field("start",1,2);
33
34      String nameactual = field.getName();
35      int actualtype = field.getType();
36      int actualnumber = field.getNumber();
37      String nameexpected = "start";
38      int typexpected = 1;
39      int numberexpected = 2;
40
41      assertEquals("name virker ikke", nameexpected, nameactual);
42      assertEquals("number virker ikke", numberexpected, actualnumber);
43      assertEquals("type virker ikke", typexpected, actualtype);
44  }
45  }
46
```

Testresultat

The screenshot shows a JUnit test runner interface. At the top, it says "Finished after 0,063 seconds". Below this, there are statistics: "Runs: 1/1", "Errors: 0", and "Failures: 0". A green progress bar is visible. The main area shows a list of test results. The first entry is "test.FieldTest [Runner: JUnit 4] (0,001 s)" with a green checkmark icon. Below it, a sub-entry "testField (0,001 s)" is also shown with a green checkmark icon. To the right of the test list, there is a tab labeled "Failure Trace".

7.4 Test af ChanceCardklassen

Oprettelse af et chancekort

```
ChanceCardTest.java | ViewCTRLTEST.java
1 package test;
2 import static org.junit.Assert.*;
11
12 public class ChanceCardTest {
13
14     ChanceCard chancecard = new ChanceCard(1,1,"description");
15
16     @BeforeClass
17     public static void setUpBeforeClass() throws Exception {
18
19     }
20
21     @AfterClass
22     public static void tearDownAfterClass() throws Exception {
23     }
24
25     @Before
26     public void setUp() throws Exception {
27     }
28
29     @After
```

Test af konstruktøren er oprette korrekt og getNumber, GetType, GetDescription

```
35
36
37     /**
38      * tester om konstruktøren ChanceCard er oprettet korrekt, om nummeret bliver returneret som forventet
39      */
38     @Test
39     public void testGetNumber() {
40         int expected = 1;
41         int actual = chancecard.getNumber();
42         assertEquals("getNumber metoden virker ikke", expected, actual);
43     }
44
45
46     /**
47      * tester om konstruktøren ChanceCard er oprettet korrekt, om typen bliver returneret som forventet
48      */
49     @Test
50     public void testGetType() {
51         int expected = 1;
52         int actual = chancecard.getType();
53         assertEquals("getType metoden virker ikke", expected, actual);
54     }
55
56
57     /**
58      * tester om konstruktøren ChanceCard er oprettet korrekt, om beskrivelsen bliver returneret som forventet
59      */
60     @Test
61     public void testGetDescription() {
62         String expected = "description";
63         String actual = chancecard.getDescription();
64         assertEquals("GetDescription metoden virker ikke", expected, actual);
65     }
66
67 }
```

Testresultat

```
Finished after 0,031 seconds
Runs: 1/1      Errors: 0      Failures: 0
testFieldTest [Runner: JUnit 4] (0,000 s)
testField (0,000 s)
```

7.5 Test af ViewCTRL

Skabelse af plads til objekterne

```
1 package test;
2 import view.*;
3
4
5
6
7
8
9 public class ViewCTRLTEST {
10     Board board;
11     Field[] fields;
12
13     CreatePlayers makePlayers;
14     Player[] players;
15
16     ViewCTRL view;
17     ChanceDeckCTRL chanceCard;
18     DieCup dieCup;
19
20     public ViewCTRLTEST() {
21         initialiseGame();
22     }
23 }
```

Oprettelse af objekter og test af at de kan oprettes og bruges.

```
24 public void initialiseGame() {
25     int numberOfPlayers;
26
27     //Lay bræt model.
28     board = new Board();
29     fields = board.getFields();
30
31     //Opret bræt.
32     view = new ViewCTRL(fields);
33
34     //Hent antal spillere.
35     String[] lines = {"2","3","4","5","6"};
36     numberOfPlayers = Integer.parseInt(view.getDropDownChoice("Vælg antal spillere 2-6", lines)); //DropDown testet.
37     view.writeText(Integer.toString(numberOfPlayers));
38     System.out.println("Dropdown og writeText testet");
39
40     //Lay player array.
41     makePlayers = new CreatePlayers(6, view);
42     players = makePlayers.getPlayers();
43
44     //Opret antal spillere på bræt.
45     view.makeGuiPlayers(players);
46
47     //Lay chancekort CTRL.
48     chanceCard = new ChanceDeckCTRL();
49     view.showChanceCard(chanceCard.getDescription()); // Vis et chancekort er testet
50     System.out.println("Chancekort er testet");
51
52     //Lay rullebæger.
53     dieCup = new DieCup();
54     dieCup.shake();
55     view.updateDice(dieCup.getDie1Value(), dieCup.getDie2Value()); //Terninger er testet
56     System.out.println("terninger er testet");
57 }
```


Test at en spiller kan rykke, update af playerposition og hans account. Test af update af buildings og ownership

```

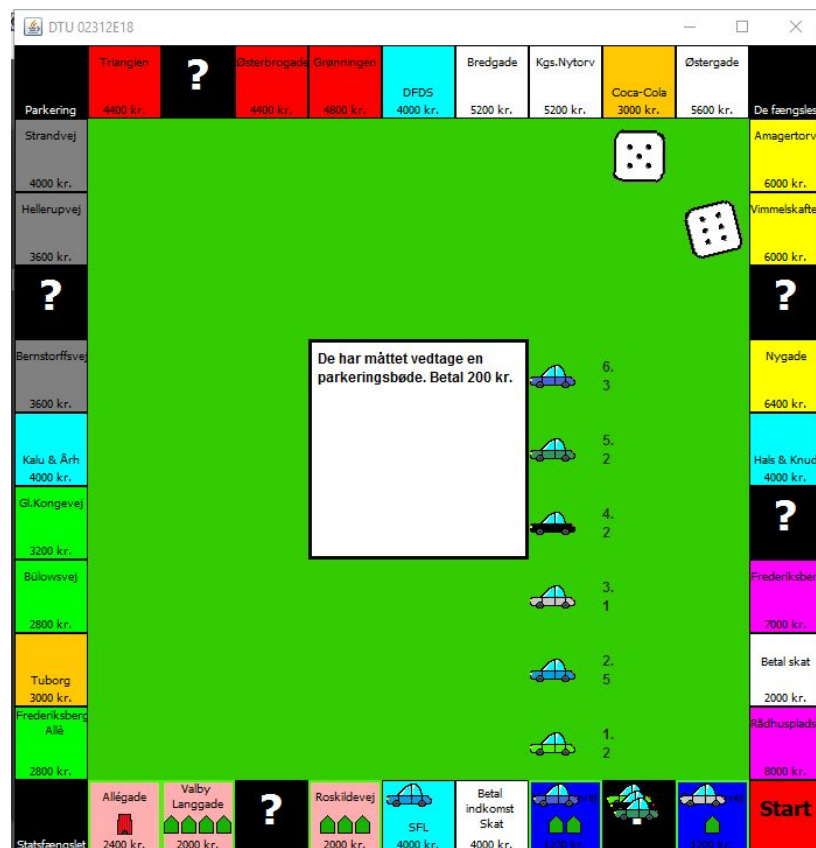
52 //lav raflebaer.
53 dieCup = new DieCup();
54 dieCup.shake();
55 view.updateDice(dieCup.getDie1Value(), dieCup.getDie2Value()); //Terningerne er testet
56 System.out.println("terninger er testet");
57
58 //test flyt alle 6 spillere
59 for(int x=1;x<=6;x++) {
60     Random number = new Random();
61     int ranNum = number.nextInt(6)+1;
62     view.updatePlayerPosition(x,0,ranNum);
63     view.updatePlayerAccount(x, ranNum);
64 } //updatePlayerPosition og updatePlayerAccount er testet
65 System.out.println("updatePlayerPosition og updatePlayerAccount er testet");
66
67 view.updateBuildings(1, 1);
68 view.updateBuildings(3, 2);
69 view.updateBuildings(6, 3);
70 view.updateBuildings(8, 4);
71 view.updateBuildings(9, 5);
72 System.out.println("updateBuildings er testet");
73
74
75
76 view.updateOwnership(1, 1);
77 view.updateOwnership(1, 3);
78 view.updateOwnership(1, 6);
79 view.updateOwnership(1, 8);
80 view.updateOwnership(1, 9);
81 System.out.println("updateOwnership er testet");
82
83
84 }

```

Testresultat



Billede af ViewCTRLtest.



Ud fra vores tests har vi lavet et spil som lever op til vores krav. Vores unit test af model-laget er gået igennem, og vi kan ud fra vores Brugertests med forskellige opsætninger af spillebrættet afprøve, at vores system virker efter hensigten. Ud fra vores ViewCTRLTEST kan vi visuelt se at vi kan flytte en spiller, sætte en ejer, trække et chancekort, købe en grund, købe huse og hoteller.

8. Projektplanlægning

Vi bruger som udgangspunkt udviklingsmetoden Unified Process, men på grund af projektets størrelse og varighed, er vores iterationer delt op i små korte forløb, som kun varer over en dag eller to for hver iteration. Vi har taget udgangspunkt i vores tidligere projekt CDIO 3, hvilket gør processen lidt anderledes end hvis vi skulle starte med at lave et helt nyt projekt fra bunden. Dette betyder at der er flere elementer i analysefasen som går hurtigere, da vi bygger videre på en gammel spilstruktur med ændringer og tilføjelser.

Dette betyder at analysefasen og designfasen nemmere bliver blandet sammen, da vi allerede har et udgangspunkt og derfor hurtigere overgår til designdetaljer.

I hele processen tager vi udgangspunkt i grasp principperne for at skabe lav binding og høj sammenhørighed.

Tidsplan													
Inception		Elaboration		Construction								Transitions	
I1	I2	E1	E2	C1		C2		C3		C4		T1	T2
02/01/2018	03/01/2018	04/01/2018	05/01/2018	06/01/2018	07/01/2018	08/01/2018	09/01/2018	10/01/2018	11/01/2018	12/01/2018	13/01/2018	14/01/2018	15/01/2018
Prioriteringer													
Business modeling	Business modeling	Business modeling										Business modeling	
Requirements	Requirements	Requirements	Requirements	Requirements	Requirements	Analyse og design	Analyse og design					Requirements	
	Analyse og design	Analyse og design		Implementering	Implementering	Implementering	Implementering	Implementering	Implementering	Implementering	Implementering	Analyse og design	
				Test				Test				Test	
												Deployment	
	Opstart i eclipse												Afslutning
	Opstart i git												

9. Konklusion

Vi kan konkludere at vores CDIO 3 har haft stor betydning for hvordan vi lagde ud med at planlægge strukturen for denne opgave, dog er en hel del ekstra klasser, samt metoder tilføjet til projektet. Vi har været gode til at prioritere hvilke krav der skal tages hånd om først, altså dem med størst risiko og udskyde de mindre risikable ting til sidst, hvor vores struktur og de fundamentale dele af koden er skrevet. Vi har afleveret et system der lever op til de krav der er beskrevet i afsnittet om krav. Vi havde en ide om at holde vores arbejdsrytme til 8 timer om dagen, men stort set dagligt blev dagene længere end planlagt. Vi har været gode til at kommunikere omkring fridage samt generelt om projektet og rapporten. De daglige møder har gjort at vi hele tiden har haft stort fokus på hvor langt vi er og hvad dagen skal gå med, og på den måde har vi drevet vores proces.

10. Perspektivering

Ting vi kunne gøre anderledes.

Design:

Man kunne lave en `playersCTRL` og en `fieldsCTRL`, som får ansvar for netop de områder i modellen, det kunne skabe en højere samhørighed, og dermed en lettere forståelse for koden.

Vi bestemte os til efter samtale ved M2 at ændre på vores AC. Dette var ikke så uproblematisk som vi havde håbet, og det krævede en del snak om binding og løsninger. Vi mistede nok lidt overblikket i forhold til samhørighed i den proces, da vi kom under et ret stort tidspres og havde vi flere iterationer ville vi nok kigge nærmere på klasser og ansvar.

Implementering:

Vi kunne godt have brugt en slags Portefølje klasse hvor en spillers ejendele var repræsenteret. Vi talte om det i starten, men gjorde det ikke fra fordi vi ikke var klar over hvor mange gange vi måtte gennemløbe spillepladens data, og da vi blev klar over det, var det for sent at foretage ændringen, da andet blev prioriteret over dette.

Vi har rigtig mange gennemløb af spillepladen for at opdatere dropdown menuer etc. Dette ville skulle løses på anden måde ved større datamængder. Evt. `ArrayLists`, men dem kan vi ikke bruge endnu.

11. Bilag

11.1 Bilag til Timeforbrug

[Link](#) til docs dokumlagent til vores timeregnskab.

11.2 Bilag til mødereferat

[Link](#) til docs dokument til vores mødereferater.

12. Litteraturliste- og kildeliste

- Java Software Solutions, John Lewis and William Loftus Eighth Edition, from 2015, ISBN 10: sd1292018232
- Applying UML and patterns an introduction to Object-Oriented Analysis and Design and Iterative Development, Craig Larman, Third Edition, ISBN: 0 13 148906-2