Danmarks Tekniske Universitet

62531 Development Methods for IT Systems
62532 Version control and test methods
02312 Introductory Programming

CDIO 3

27th November 2020

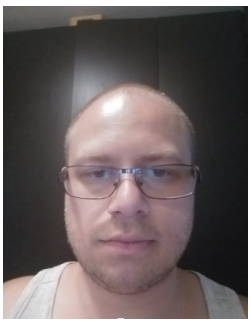
Lucas Arleth Lykke
s205447

Mike Patrick Nørlev Andersen
s205417

Sebastian Andreas Almfort
s163922

Martin Koch
s182935

Anne Sophie Bondegaard Petersen
s194582



Jan Engers Møller Pedersen
s205419

1 Resumé

I denne rapport gennemgås processen, hvorved det er lykkes at løse det nye projekt der er kommet til **IOOuterActive**. Denne gang er det spillet **Monopoly Junior** der skal implementeres. Udviklingen af dette system, og rapporten er konstrueret ved at bygge videre på viden og metoder fra de tidligere spil, der er blevet implementeret af **IOOuterActive**, i rapporterne CDIO 1 og CDIO 2.
Denne rapport gennemgår hele forløbet fra de første overvejelser omkring denne opgave, til det endelige produkt, der bliver leveret af **IOOuterActive**.

Indholdsfortegnelse

2 Timeregnskab

	Timer
Patrick	40
Martin	40
Sebastian	40
Lucas	40
Jan	40
Anne Sophie	40

Tabel 1: Timetabel

3 Indledning

I dette projekt har vi fået til opgave, at designe, implementere og teste et Monopoly Junior spil mellem 2-4 personer. Under udarbejdelsen af projektet vil vi benytte UML til at udforme kravsspecificering, Use Case beskrivelser, artefakter og C.A.S.E teori, til at danne et godt overblik over designet af produktet, samt for at kunne udvikle spillet bedst muligt. Videre vil du kunne læse om de testtyper vi har brugt for at sikre os at softwaren fungerer som ønsket. Disse ting tilsammen lader os udvikle det bedst stykke mulige software.

4 Kravspecifikation

I dette kapitel er grundstenen til gennemførelsen af dette projekt lagt. Kundens krav bliver gennemgået, analyseret og opstillet således, at der kommer et overordnet overblik over hvad der skal opfyldes og hvordan det kan blive opfyldt. Disse krav bliver så anvendt i hhv. analysen af hvordan det anmodede system skal fungere, og i designet af selve systemet, så det sikres at vores prioriteret krav opfyldes, når elementerne endelig implementeres. Hvis man bliver i tvivl over hvad prioriteret krav der skal implementeres og hvordan, vil man altid kunne vende tilbage til dette kapitel.
Således gøres det altså muligt, at kunne sikre sig, at man får gennemgået alle de prioriteret krav der er blevet stillet, herunder spillets regler.

4.1 Krav liste

Kravlisten er opbygget fra ventre mod højre, med et kravID, en beskrivelse af det pågældende krav, og en indikation af, hvorvidt det endelig system overholder kravet.
Kravlisten indeholder de prioriterede krav for projektet.

Krav	Kommentar	Overholdt / Ikke overholdt
R1	Kan bruges på Windows computer	✓
R2	Spil mellem 2 - 4 personer	✓
R3	Spillerne skal kunne lande på et felt og så fortsætte derfra på næste slag	✓
R4	Spilleren skal gå i ring på brættet	✓
R5	De 20 chancekort skal blandes i starten af spillet.	✓
R6	Et start pengebeløb deles ud blandt spillerne	✓
R7	Når en spiller går fallit, er vinderen spilleren med højest formue blandt de resterende spillere.	✓

Krav	Kommentar	Overholdt / Ikke overholdt
R8	Den yngste spiller starter	✓
R9	Spillere modtager et fastlagt beløb når målfelt passeres	✓
R10	De angivne felter i spillets regler skal være implementeret i spillet	✓
R11	Hver spiller skal havde en personlig figur	✓
R12	Hvis en spiller lander på et tomt felt, skal vedkommende købe feltet.	✓
R13	Hvis en spiller lander på et ejet felt, skal vedkommende betale for at holde der	✓
R14	Hvis en spiller ejer alle felter med en specifik farve fordobles huslejen	✓
R15	Skal kunne trække chancekort og udføre instruktionen på chancekortet	✓
R16	Landes på direkte-i-fængsel feltet, ryger man i fængsel	✓
R17	Betal kaution eller brug løsladelseskort (hvis ejer pågældende kort)	✓

Den ovenstående liste opgiver de krav der er valgt der skulle implementeres i systemet,

og samtidig har været en checkliste for at se, om alle prioriterede krav er blevet implementeret.
Der er nogle få krav i spillet, hvor det er blevet afgjort at de ikke skulle implementeres. Et af disse krav, er kravet om, at hver spiller kun kan eje 12 felter.

4.1.1 Chance kort

For at gøre det nemmere at få overblik over diverse chancekort, har vi tildelt hvert chancekort et nummer. Yderligere har vi markeret hvorvidt det er et "passivt", "aktivt" eller "dobbelt aktivt" chancekort. Er chancekortet "passivt" laves der en automatisk handling hvor aktøren ikke har mulighed for at tage et valg. Er chancekortet "aktivt" skal aktøren træffe et valg, for at kunne opnå chancekortets effekt. Er det et "dobbeltaktiv" kort, er der samme effekt som et "aktivt" kort, dog skal flere spillere tage et valg. Dette kan ses i følgende tabel:

Nummer	Beskrivelse	Passiv/Aktiv/Dobbelt aktiv
1:	Giv dette kort til "BILEN", og tag et Chancekort mere. BIL: På din næste tur, tag til et vilkårligt ledigt felt og køb det. Hvis der ingen ledige felter er, skal du købe et fra en anden spiller!	Dobbelt aktiv
2:	Ryk frem til START, modtag M2	Passiv
3:	Ryk OP TIL 5 felter frem	Aktiv
4:	Ryk frem til et orange felt. Hvis det er ledigt får du det gratis! Ellers skal du betale leje til ejeren	Aktiv
5:	Ryk 1 felt frem, ELLER tag et chancekort mere	Aktiv

6:	Giv dette kort til SKIBET, og tag et chancekort mere. SKIB: på din næste tur skal du sejle frem til et vilkårligt ledigt felt. Hvis der ingen ledige felter er, skal du købe et fra en anden spiller	Dobbelt aktiv
7:	Du har spist for meget slik, BETAL M2 til banken	Passiv
8:	Ryk frem til et orange eller grønt felt. Hvis det er ledigt, får du det gratis! Ellers skal du betale leje til ejeren	Aktiv
9:	Ryk frem til et lyseblåt felt. Hvis det er ledigt, får du det gratis! Ellers skal du betale leje til ejeren	Aktiv
10:	Du løslades uden omkostninger! Behold dette kort, indtil du får brug for det	Passiv
11:	Ryk frem til Strandpromenaden	Passiv
12:	Giv dette kort til KATTEN, og tag et chancekort mere. KAT: på din næste tur, tag til et vilkårligt ledigt felt og køb det. Hvis ingen ledige felter, skal du købe et fra en anden spiller	Dobbelt aktiv

13:	Giv dette kort til HUNDEN, og tag et chancekort mere. HUND: på din næste tur, tag til et vilkårligt ledigt felt og køb det. Hvis ingen ledige felter, skal du købe et fra en anden spiller	Dobbelt aktiv
14:	Det er din fødselsdag! Alle giver dig M1	Passiv
15:	Ryk frem til et lyserødt eller mørkeblåt felt. Hvis det er ledigt, får du det gratis! Ellers skal du betale leje til ejeren	Aktiv
16:	Du har lavet alle dine lektier, modtag M2 fra banken	Passiv
17:	Ryk frem til et rødt felt. Hvis det er ledigt, får du det gratis! Ellers skal du betale leje til ejeren	Aktiv
18:	Ryk frem til SKATERPARKEN for at lave det perfekte grind! Hvis ingen ejer det, får du det gratis! Ellers skal du betale leje til ejeren	Passiv
19:	Ryk frem til et rødt eller lyseblåt felt. Hvis ingen ejer det, får du det gratis! Ellers skal du betale leje til ejeren	Aktiv

20:	Ryk frem til et brunt eller gult felt. Hvis ingen ejer det, får du det gratis! Ellers skal du betale leje til ejeren	Aktiv

Tabel 2: Beskrivelser af chancekort

4.2 Krav matrix

Kravmatricen er en matrix der er opstillet således, at man nemt kan få et overblik over, om alle de opstillede krav er opfyldt ud fra de Use Cases der er opstillet i kapitel 5.2.

Requirements/ Use cases	ID: 1.0	ID: 1.1	ID: 1.2	ID: 1.3	ID: 1.4	ID: 1.5
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						
R9						
R10						
R11						
R12						
R13						
R14						
R15						
R16						
R17						

Tabel 3: Requirements Tracing Matrix

4.3 Krav prioritering

Under projektformuleringen er der specificeret, at det er tilladt at prioritere hvilke krav som implementeres i systemet, og hvilket der bliver set bort fra. De krav der er blevet fravalgt, er krav der ikke ses som bærende i spillet. Det er krav, hvilket spillet stadig er et færdigt produkt foruden, bare med lidt færre krav og regler.
Der er nogle krav som vi senere i forløbet har valgt ikke at implementere, da de ikke bringer værdi til spillet. Et af disse krav stammer fra begrænsningerne fra det fysiske spil som vores software er baseret på, nemlig at der kun er M90 i spillet. Dette er i bedste fald en unødvendig restriktion i vores software.
Andet krav er grænsen for at en spiller maksimalt må eje 12 felter. Dette er igen en regel der intet gør ud over at hæmme det naturlige flow i spillet.

5 Use Cases

Use Cases er scenarier hvori flowet af systemet gennemgås. Dette er med til at give en idé og et overblik over hvad systemet skal indeholde, men også hvordan man opbygger systemet, så det opfører sig som ønsket.
Det er både spillets Main flow der bliver gennemgået, og hvilket andre flows der kan springe ud fra Main flowet. Disse kaldes Alternativ flow.

5.1 Use Case overblik

Ved hjælp af kravsspecifikationen, har vi her udarbejdet et Use Case diagram for at give en visuel repræsentation af den overordenede interaktion mellem aktøren og systemet, og herunder forskellige Use Cases i systemet.

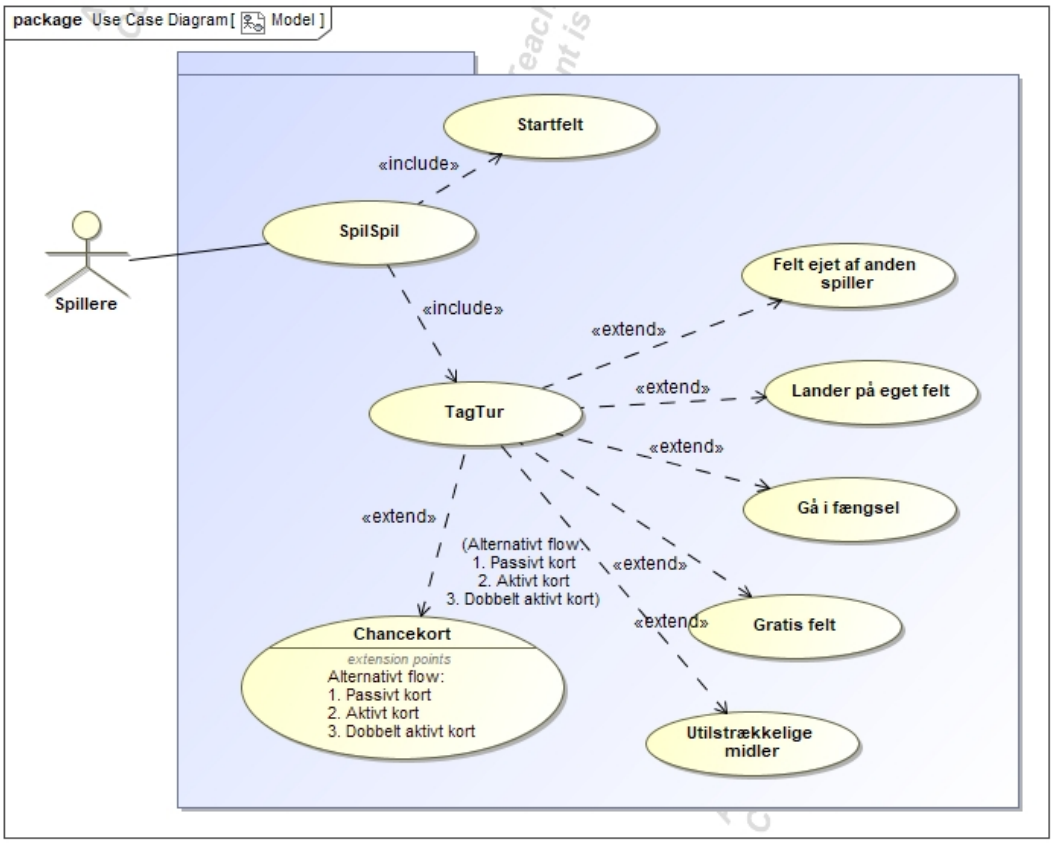


Figure 1: Use case diagram

Det ses på diagrammet, at spillet altid indeholder to tilfælde. Spillet starter ved startfeltet, og spillerene tager en tur. Ud fra tilfældet 'Tag tur', er der givet eksempler på forskelige scenarier, hvoraf enkelte er beskrevet herunder. Yderligere er der et enkelt

extension point, som ses under chancekort. Hvad disse extensions repræsenterer, blev uddybet tidligere i rapporten.

5.2 Use Cases

Use Cases beskriver i detaljer udvalgte flows i systemet.

Use case: SpilSpil
ID: 1.0
Brief description: Spillet spilles Primary actors: Alle spillere Secondary actors: Ingen
Preconditions: 1. Spillernes rækkefølge er valgt 2. Chancekort er blandet 3. Spillerne står på startfeltet Main flow: 1. Første spiller slår med terningen 2. Alt efter hvilket felt man lander på, udføres den operation der beskrives i reglerne. 3. Spilleren giver terningen videre til næste spiller, der udfører punkt 1 (a) punk 1, 2 og 3 går i loop, hvor rækkefølgen af spillere bliver fulgt 4. Når en spiller erklæres konkurs, optælles de resterende spilleres formue, og en vinder findes
Postconditions: Vinderen er fundet
Alternative flows: 1. Lander på tomt felt 2. Lander på felt ejet af andre 3. Lander på chancekort 4. Lander på fængsel

Tabel 4: Use Case for Main flow

Alternativ flow: TomtEjemdomsfelt
ID: 1.1
Brief description: Spiller lander på tomt ejendomsfelt
Primary actors: Spiller
Secondary actors: Banken
Preconditions: 1. Spilleren lander på tomt ejendomsfelt Main flow: 1. Spiller slår med terning 2. Spiller rykker pladser frem der svarer til antal øjne på terning 3. Spiller køber feltet som der landes på, ved at betale banken det beløb, der er opgivet på feltet 4. Spiller placerer et "solgt skilt" på feltet. 5. Terningerne gives videre til næste spiller
Postconditions: Det er næste spillers tur

Tabel 5: Alternativ flow 1 for Use Case 1

Alternativt flow: FeltEjetAfAndenSpiller
ID: 1.2
Brief description: Spiller er landet på et felt, der er ejet af en anden spiller
Primary actors: Spiller(1)
Secondary actors: Spiller(2) der ejer det pågældende felt
Preconditions: 1. Spiller(1) lander på et felt ejet af en anden, spiller(2) 2. Spiller(1) har råd til husleje
Flow: 1. Spiller(1) lander på felt ejet af spiller(2) 2. Spiller(1) betaler den opgivet husleje til ejeren, spiller(2) 3. Spiller(1) giver terningerne videre til næste spiller i rækkefølgen. Spillet fortsætter i SpilSpil, flow punkt 1
Postconditions: Næste spillers tur

Tabel 6: Alternativt flow 2 for use case 1

Alternativt flow: SpillerGårKonkurs.
ID: 1.3
Brief description: Spiller har ikke penge nok til at købe en ejendom der er landet på, betale afgift fra chancekort, eller eller betale en anden spiller husleje.
Primary actors: Spiller
Secondary actors: Alle andre spillere
Preconditions: Spiller lander på felt og kommer ud for et af følgende tre scenarier: 1. Spiller lander på ejendomsfelt ejet af anden spiller og kan ikke betale husleje 2. Trækker chancekort med afgift, som spilleren ikke kan betale 3. Spiller lander på tomt ejendomsfelt, men har ikke råd til at købe ejendommen
Flow: 1. Alle spillere der ikke er gået konkurs tæller deres penge. (a) Hvis to spillere har samme antal penge, optælles værdien af alle spilleres ejendomme 2. Spilleren med størst formue afgøres som vinder af spillet 3. Spillet slutter
Postconditions: Vinderen er fundet

Tabel 7: Alternativt flow 3 for use case 1

Alternativt flow: SpillerRammerChancekort
ID: 1.4
Brief description: Spiller er landet på Chancefeltet, og skal trække et chancekort
Primary actors: Spiller
Secondary actors: Måske andre spillere
Preconditions: 1. Spilleren er landet på et chancefelt
Flow: 1. Spiller trækker et chancekort, og udfører instruktionen på kortet 2. Det er næste spillers tur 3. Spillet fortsætter i SpilSpil, flow punkt 1
Postconditions: Næste spillers tur

Tabel 8: Alternativt flow 4 for use case 1

Alternativt flow: GåIFængsel
ID: 1.5
Brief description: Spiller lander på gå direkte i fængsel feltet
Primary actors: Spiller
Secondary actors: Ingen
Preconditions: 1. Spilleren er landet på direkte i fængsel feltet
Flow: 1. Spiller rykker til fængselfeltet (a) Hvis start krydses, modtages der ingen penge fra banken (b) Har stadig indtjening fra husleje 2. Turen går videre til næste spillers tur 3. Ved næste tur, betal 1M eller brug løsladelseskortet (Hvis har det) til at blive løsladt 4. Spil derefter videre som var det din tur, i Use case SpilSpil, punkt 1
Postconditions: Næste spillers tur

Tabel 9: Alternativt flow 5 for use case 1

6 Analyse

I dette afsnit anvender vi Unified Process, til at udvikle relevante artefakter til beskrivelsen af det ønskede system. Artefakterne er udviklet med henblik på at beskrive et system, som overholder kravsspecifikationen.

6.1 Domænemodel

Vores domænemodel definerer de overordnede elementer i spillet.
Spillet udgår fra Gameboard klassen, som gør brug af en terning og indeholder 20 chancekort. Spillet spilles af mellem to og fire spillere som på et givent tidspunkt kan råde over op til hhv. 12 "solgt"-skilte og 90 penge.

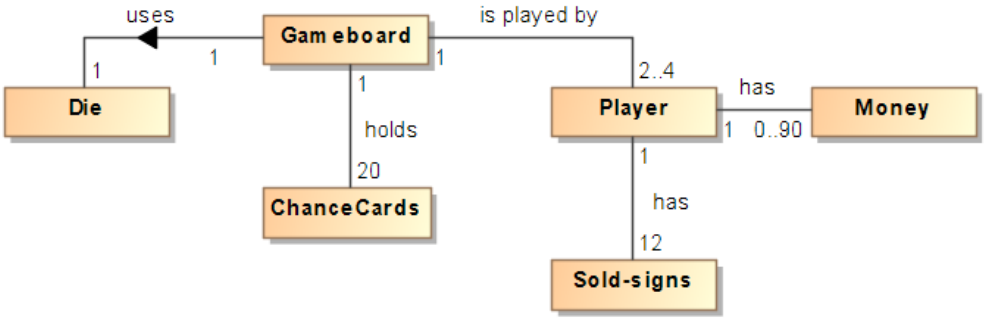


Figure 2: Domænemodel

6.2 Systemsekvensdiagram

Vores systemsekvensdiagram viser interaktionen mellem vores aktør og systemet, spilleren og spillet.
Spilleren initierer spillet, opsætter sin spiller og starter spillet efter anmodning fra systemet. På skift kaster spillerne terningen og køber eller betaler for feltet de lander på, eller reagerer på et chancekort. Dette foregår så længe ingen af spillerne har tabt.
Når en spiller har tabt, returneres denne information og hernæst en tilkendegivelse af vinderen. Spillerne får herefter muligheden for at spille igen.

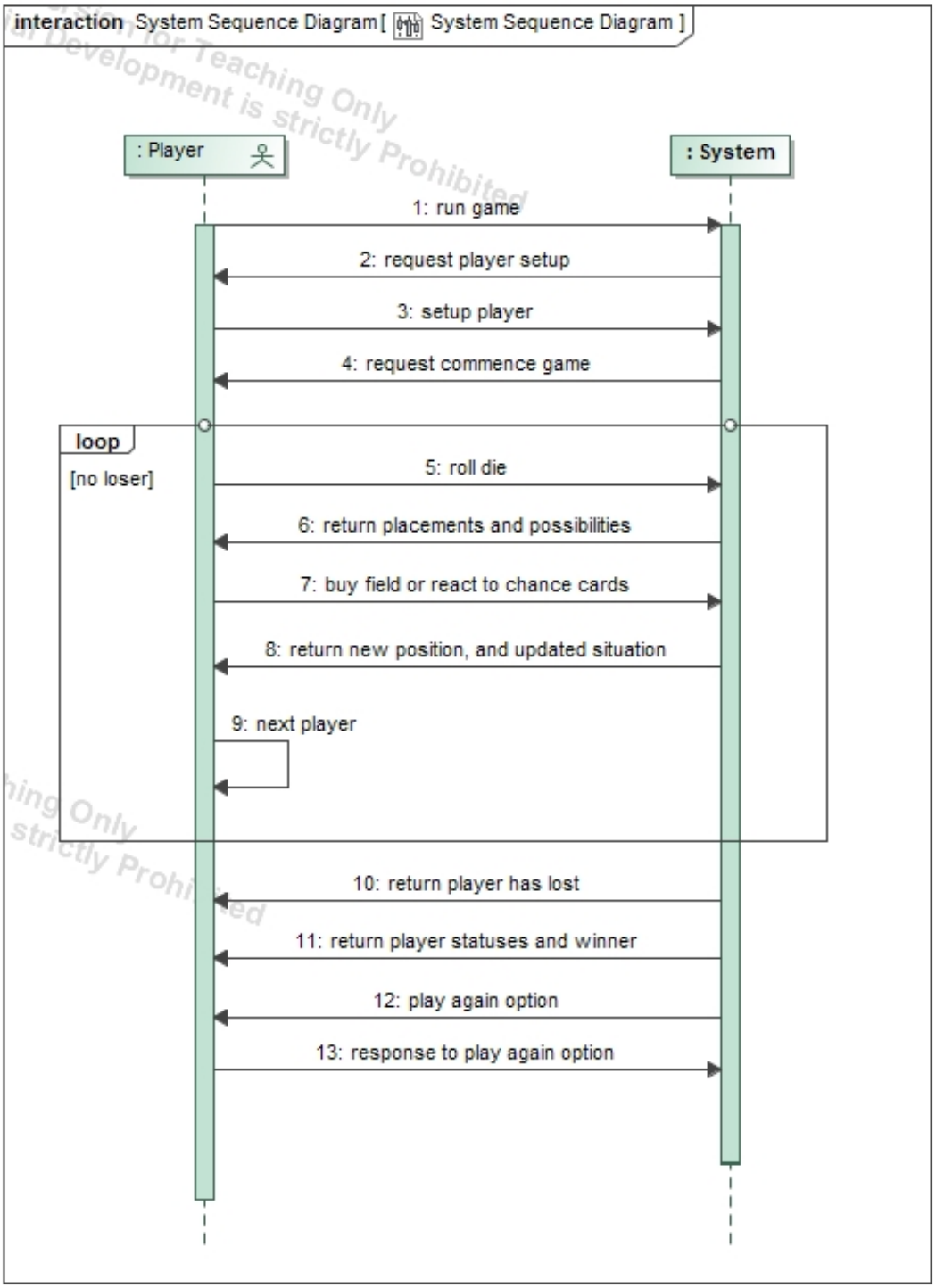


Figure 3: Systemsekvensdiagram

7 Design

I designdelen af vores projekt har vi lagt planerne for, hvordan funktionerne angivet i analysedelen skal udføres i praksis. Ved hjælp af forskellige artefakter vil vi beskrive hvordan vores system skal fungere indenfor rammerne af de anførte krav. Derudover vil vi i dette afsnit beskrive vores tanker omkring G.R.A.S.P. konceptet under indførelse, heraf i designet af vores system.

7.1 Sekvensdiagram

Processen bag en spillers tur, ses i nedenstående sekvensdiagram.
Der er her illustreret, fra at GUI'en modtager et signal fra brugeren, som starter processerne i GameBoard gennem IUController-klassen.
Fra GameBoard-klassen anvendes terningen i Die-klassen, spillerens position hentes og opdateres med hensyn til det felt der svarer til forskellen mellem terningens værdi, og spillerens position.
I det 11. kald landOnField(Player), sker der flere ting. Spillerens position opdateres i forhold til det nye felt, afhængigt af om spilleren lander på en ejendom, et chancefelt, gå-i-fængsel, start-feltet eller en gratis felt at besøge. Afhængigt af feltet, opdateres spilleren pengebalance ligeså.
Disse informationer sendes herefter tilbage til UIControlleren, som kommunikerer ud til brugergrænsefladen.
Det 20. kald "updateGUI", dækker over de opdaterede værdier, som sendes gennem UIControlleren, og ud til brugeren.
Disse processer går igen, i takt med at hver spiller tager sin tur, og fortsætter så længe ingen spiller har tabt.

I næste underafsnit, giver vi i endnu et sekvensdiagram, et eksempel på hvad der sker, når en spiller lander specifikt på et chancefelt.

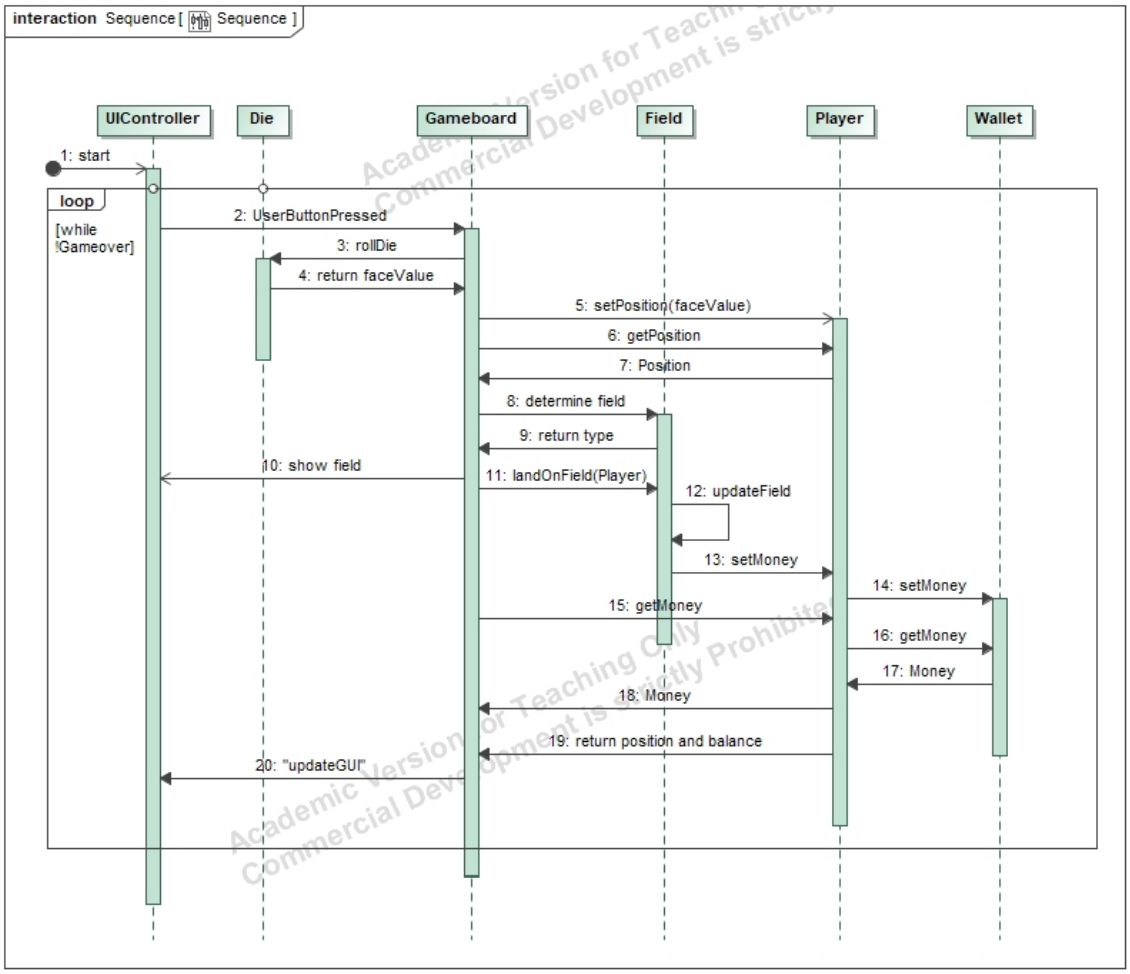


Figure 4: Sekvensdiagram for en spillers tur

7.1.1 Chancefelt

Herunder, i Figure 5, kan man se metodekald og dataflow når man lander på et chancefelt. Det 8. kald er en abstraktion, da den præcise måde hvorpå det håndteres divergerer fra kort til kort.
Et eksempel herpå er når et ChooseToMove kort trækkes. Hvis det valgte felt ikke er ejet, tildeler man spilleren feltets pris, i penge, da når de så lander på det og køber det, svarer det til at man får det gratis.

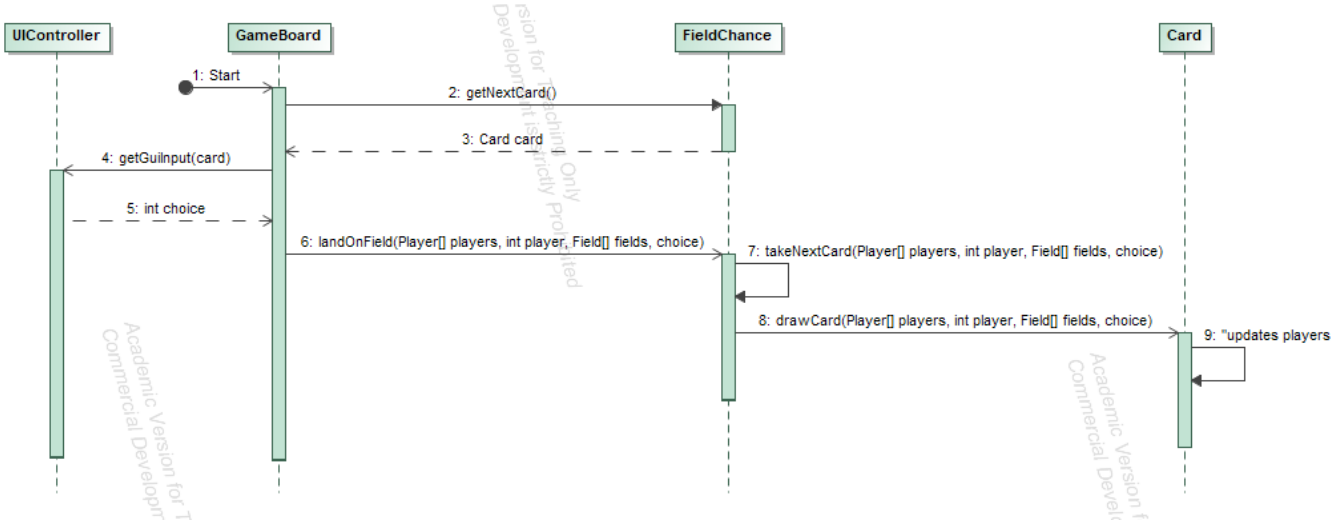


Figure 5: Sekvensdiagram for at lande på FieldChance

7.2 Designklassediagram

I vores design klassediagram, kan det ses på Figure 6, at vi har valgt at bruge GameBoard til den krævede funktion, at indeholde et array af felter, men også som controller for vores logiske lag. Vi har videre en klasse FieldsOnBoard der indeholder informationen omkring felter, samt evnen til at returnere et array af felter som GameBoard kan bruge.

Vi har den abstrakte klasse Field som vi nedarver fra, til en række forskellige subclasses, hvor den der er ændret mest er vores FieldChance. Vi har kun 1 FieldChance i memory, men denne reference sendes så til 4 steder i vores FieldArray. Dette gør, at man hele tiden tager chancekort fra den samme bunke som FieldChance har et DynamicArray af. DynamicArray er vores egen forsimplede implementation af en ArrayList. Den er ikke medtaget i klassediagrammet da den ikke hører til et særligt sted. Den er generelt brugt til at håndtere variable bunker af chancekort, men disse findes både hos Player klassen når en spiller skal holde på et kort, men som sagt også i FieldChance.

Vi har en Wallet klasse, der tilhører Player. Denne bruges til at holde styr på pengebeholdningen og videre har vi et Player[ ] i GameBoard.
Som bro mellem det logiske lag og det grafiske lag har vi UIController. Denne bruges til at sende information til GUI så man har en separation mellem de 2 klasser.
Ud over Field der er en abstrakt klasse, så har vi også en abstrakt klasse ChanceCard. Vi har opdelt Field i 5 forskellige nedarvede klasser: Properties, Jail, FieldInfo, FieldStart og FieldChance.
Af de nedarvede klasser har vi kun inkluderet FieldChance, da denne er stedet hvor vi opbevarer vores chancekort og er derfor vigtig at have med for forståelsen af strukturen.
Vi har opdelt ChanceCard i 5 forskellige nedarvede klasser: ChooseToMove, PayOrGetPaid, GetOutOfJail, SpecificField og PlayerSpecific.

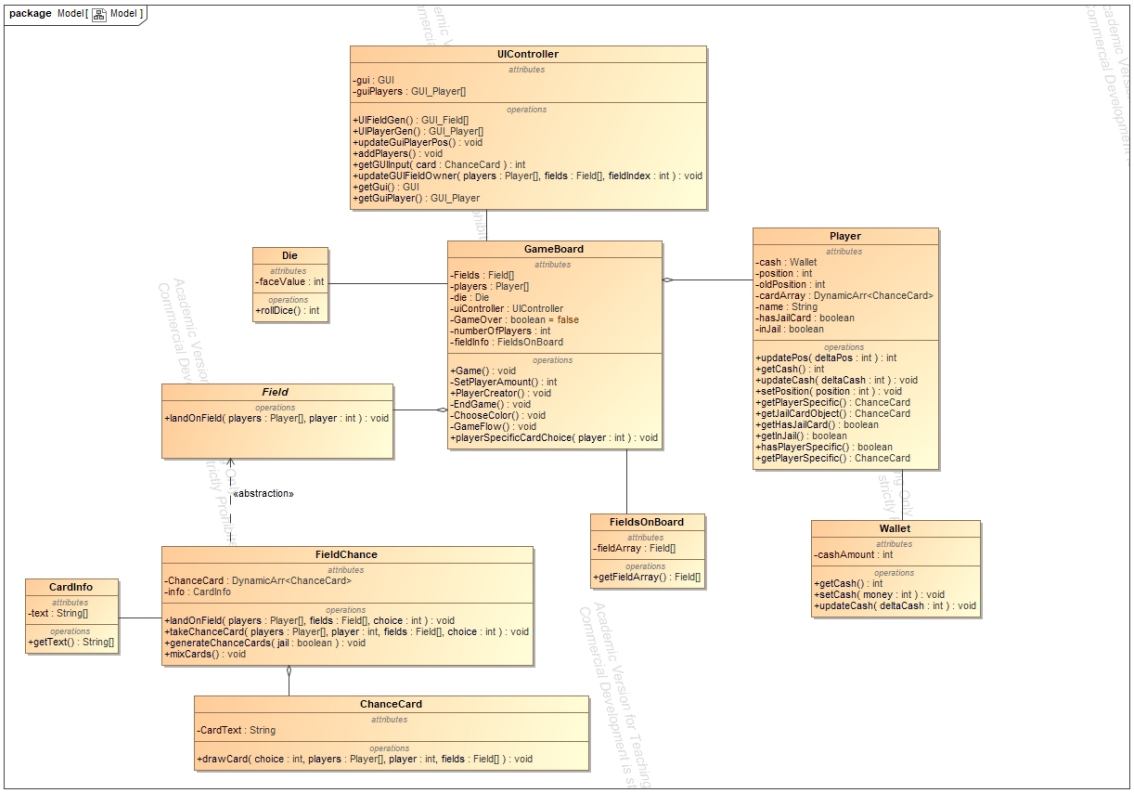


Figure 6: Design klassediagram

7.3 G.R.A.S.P

Ved at lægge G.R.A.S.P konceptet ned over modellerne som koden er skrevet ud fra, kan designet af modellerne og derved også koden gennemgås, og det kan give en mulighed for at se, hvad der bør forbedres for at programmet kommer til at virke mere optimalt.

7.3.1 Creator

I implementeringen af koden er der en række creators. Af disse vil vi her beskrive nogle stykker. **GameBoard** står for at oprette objekter til **Die**, **FieldsOnBoard**, **Player** og **Field**.
Den anden creator er **FieldChance** der står for at oprette objekter til **CardInfo** og **ChanceCard**.

7.3.2 Information expert

I dette projekt er der oprettet mange klasser, der løser mange forskellige funktioner, ved at bruge nedarvning. For at få et overblik over hvilket klasser der fuldfører hvilke funktioner, vil det bedste være, at kigge på Design Klassediagrammet. Hver klasse fuldfører de funktioner der står skrevet under Operations.
Under implementeringen har der været fejl der har skulle rettes til. For at dette har kunne lade sig gøre, er der visse steder blevet breaket med designet.

7.3.3 Coupling

Systemet har low coupling. Dette kan ses, ved at det er nogle få overordnet klasser **GameBoard**, **Player**, **Field** og **FieldChance**, som creatorer alle nødvendige objekter, eller som de andre klasser nedarver fra.
Fordelen ved at koden er implementeret med denne low coupling er, at det i fremtiden gør det nemmere at ændre eller implementere i koden. Yderligere har det den fordel, at det bliver nemmere at tage koden fra dette projekt og implementere ind i andre projekter.

7.3.4 Cohesion

Koden til dette projekt har en high cohesion. Det ses at koden har en high cohesion ved, at koden er implementeret på en sådan måde, at det er nemt og overskueligt at se, hvilket klasser der har det overordnet ansvar for hvilket funktioner.
Koden er skrevet således, at der er en overordnet klasse, hvori der nedarves en masse klasser, der løser de forskellige funktioner, som den overordnet klasse så står for at holde styr på.

7.3.5 Controller

I dette system er der to controllere. En controllere for den logiske lag, **GameBoard**, og en controller for view laget, **UIController**, der styrer flowet for GUI'en.

8 Implementering

8.1 LandOnField() metoden

I vores Field klasse har vi en abstrakt metode landOnField. Denne skal implementeres forskelligt, alt efter hvilken subclass der er tale om. Vi har følgende subclasses:

1. Jail
2. FieldInfo
3. FieldChance
4. FieldStart
5. Property

Ved subclassen Jail skal landOnField opdatere en intern variable hos spilleren. Derefter skal positionen af spilleren opdateres, så den stemmer overens med fængselsfeltet. Hvis denne variabel så er sand, skal en spiller betale 1M ved næste runde. Videre modtager spilleren ikke 2M for at passere start, hvis variablen er sand. På denne måde kan vi bruge samme logik til at rykke spilleren til fængsel, som ved alle andre positionsskift. Dog tjekker man om variablen er sand eller falsk for at tildele spilleren 2M eller ej, når de passerer start. Ved start på en ny tur ser man om personen er i fængsel, hvis ja skal de betale 1M eller bruge deres kort, hvor man derefter sætter variablen til falsk igen.

Subclassen fieldInfo indeholder de felter hvor landOnField intet gør, ud over at skrive om man har fået gratis parkering, eller skal på besøg i fængsel.

Subclassen FieldChance skal trække et chancekort til spilleren. Her har vi delt det op, så FieldChance står for at trække kort og holde styr på bunken af kort, hvorimod at Cards klassen så indeholder logikken for hvad der sker, når man trækker det enkelte kort

Subclassen Property har nogle forskellige flows alt efter om hvad der gælder af følgende eksempler. Er feltet ejet eller ej, hvem der eventuelt ejer feltet, og om de ejer andre felter af samme farve. Hvis feltet ikke er ejet skal man købe det. Her fjerner man et antal penge fra spilleren, og opdaterer herefter ejendommens "ownedBy" variabel. Hvis feltet er ejet, skal man tjekke om spilleren selv ejer feltet, eller om feltet ejes af en anden. Hvis spilleren selv ejer feltet skal der ikke ske noget. Hvis en anden spiller ejer feltet skal man tjekke om den anden spiller ejer hele gruppen. Hvis den anden spiller kun ejer feltet skal man betale det normale beløb og hvis hele gruppen ejes skal man betale det dobbelte. Når man betaler til en spiller, skal man flytte et antal M fra spilleren der lander på feltet til spilleren der ejer feltet.

Subclassen FieldStart skal returnere 2M når man passerer det og ikke er på vej i fængsel. For at give den ønskede effekt har vi implementeret et check i spillerens setPostion metode, og setSpecificPosition, så hvis spillerens position er mindre end oldPosition samt at dens inJail attribut er falsk, så tildeler man 2M.

8.2 drawCard() metoden

drawCard metoden fra vores chancekort håndterer hvad der sker når en specifik spiller trækker et specifikt kort. En yderligere komplikation kan ligge i, at der er en række kort hvor man skal foretage et valg når man trækker det.
For at valget kan komme med, har vi gjort på følgende måde. Hvis en spiller lander på chancefeltet, undersøger man hvilket kort der skal til at trækkes. Dette kort parser man så til UIControllerens funktion getGuiInput, der giver brugeren det ønskede output - i nogle korts tilfælde intet - og returnerer et heltal der repræsenterer valget. Dette heltal er -1 i tilfælde af at intet valg skulle tages.
Dette valg bruges så som parameter i drawCard Metoden, efter at være kørt gennem metoden takeNextCard() i FieldChance, der udover at kalde drawCard metoden, også håndterer ændringer af det DynamicArr af kort når man trækker.
De mange forskellige kort har vi som tidligere nævnt delt op i 5 nedarvede klasser. Disse nedarvede klassers drawCard metode har vi så yderligere delt op med switches. Man kunne også lave 20 nedarvninger, men det virkede som en mindre optimal løsning.

8.3 DynamicArr klassen

Da det er blevet set ned på, at man bruger den indbyggede metode arraylist i CDIO3 opgaven har vi lavet vores egen klasse fra bunden, som fungerer som en arraylist. Klassen indeholder forskellige metoder, hvoraf nogle af disse metoder er:

- decrease() - fjerne det bagerste element i arrayet
- add() - add klistrer et ekstra element på bagerst i arrayet
- addAtStart() - klistrer et ekstra element på foran arrayet (index 0)

Som eksempel, er add metoden vist på nedenstående figur.

```java
// we add the size of the array      everytime it has reached the limit.
public void add(T data) {
    if (size == current) {
        T[] newarr = (T[]) new Object[1 + size];

        //setting old array elements into new array
        for (int i = 0; i < size; i++) {
            newarr[i] = arr[i];
        }
        size++;
        arr = newarr;
    }
    //arr = new int[newarr.length];
    //arr = newarr;
    arr[current] = data;
    current++;
}
```

Figure 7: Dynamic array add metode

9 Test

Vi har benyttet en række forskellige testmetoder. White-box i form af Unit Test, integrationstest, systemtest, og black-box i form af brugertest under udviklingen af softwaren.

9.1 Unit test

I vores program har vi mange klasser, hvoraf størstedelen af klasserne holder styr på logikken af chancekort. Under programmeringen af chancekortene er der lavet en Unit Test til hver enkelt klasse, da chancekortene indeholder en stor del af logikken i koden. Med Unit Test testes små bider af koden isoleret, for at undgå uventede fejl.

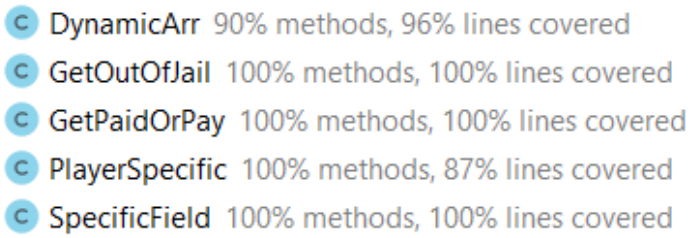


Figure 8: Unit Tests for chancekort

Som det ses på Figure 8, er der en metodedækning på 100%, dette betyder at alle metoderne er gennemkørt minimum en gang. Det skal siges, at selvom der er 100% dækning på metoderne, betyder det ikke nødvendigvis at det hele fungerer som det skal. Selvom hver klasse virker individuelt, betyder det ikke at det hele virker samlet. Dette bringer os til vores næste punkt i tests:

9.2 Integrationstest

Nu virker alle klasserne individuelt, men virker de samlet? For at finde ud af dette har vi lavet en ny Unit Test af superklassen for alle kort. Hver klasse som er nedarvet fra superklassen, er allerede testet, men for at teste denne superklasse fortages en integrationstest som demonstrer at forskellige dele af kildekoden virker i samspil med hinanden. Unit testen til denne klasse replikerer det normale Junior Matador spil, men man kan KUN lande på chancefelterne. På denne måde kører den igennem alle chancekortene en efter en, og der kan vurderes hvorvidt der er sammenspil mellem klasserne.

9.3 Systemtest

Efter diverse Unit Tests og integrationstests er det på tide at teste det hele samlet. Systemtesten har fungeret på den måde, at alle i gruppen har kørt programmet hvorefter skrevet ned hvad der ikke virker eller mangler. Der blev fundet få fejl eller mangler, som derefter blev rettet. Vi er positive over, at det færdige produkt virker som det forventes af kravspecifikationerne lavet i designfasen.

På nedenstående figur kan der ses et overblik over total "line coverage" for alle tests.

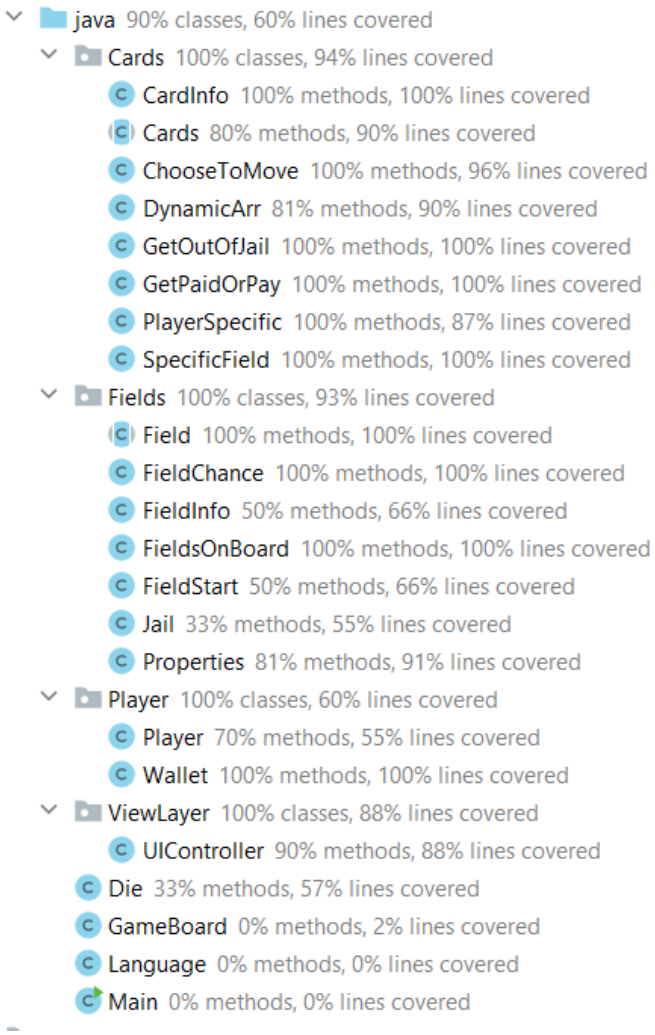


Figure 9: Linecoverage

9.4 Brugertest

Vi har foretaget flere brugertest. Disse brugertests gjorde at forbedringer blev lavet og nye problemer blev skabt. Kritikpunkterne vi modtog var som følgende

- Den grønne baggrund som programmet er født med var meget i iøjnefaldende. Dette gjorde at baggrundsfarven blev genovervejet og blev sat til at mere neutral farve
- Centerboxen som viser kort har en meget iøjnefaldende farve
- Teksten øverst til venstre der forklarer forløbet af spillet er for utydelig
- Chancekort forbliver fremme på brættet indtil et nyt kort bliver trukket. Dette forvirrer, da det fylder meget på skærmen

De to første punkter var kritikpunkter vi fik ved de første brugertest. Da vi løste de to øverste punkter, kom de to nederste punkter på, efter flere brugertest.

10 Konfiguration

10.1 Minimums Systemkrav

- CPU: Pentium 2 266 MHz processor
- Ram: 128 MB
- Lagerplads: 5 MB
- Programmer: Java JRE 8, Git, IntelliJ Idea 2020.2.1

10.2 Installation

Ved installation og brug af dette program, er det første trin at hente det fra Git. Dette kan gøres ved at benytte IntelliJ til at hente en klon af programmet gennem følgende link, som indeholder programmets filer: **https://github.com/sealnpenguin/CDIO3.git** Dog skal brugeren også få adgang til projektet gennem projektejeren.

10.3 Kompilering

Kompilering af programmet udføres ved hjælp af Java Virtual Machine, da den benytter klasse filerne til at oversætte det til maskinens sprog. Derfor vil det være en nødvendighed at systemet har adgang til Java JRE 8.

10.4 Afvikling

I Git repositoriet er den en .jar-fil som man kan benytte til at starte programmet. Dette kan gøres igennem IntelliJ eller kommandoprompten.

- IntelliJ:
  For at benytte IntelliJ for at køre programmet, højreklikkes CDIO3.jar filen, for derefter at trykke på 'run'

  Yderligere er det muligt at køre programmet ved brug af Main-klassen. Dette gøres ved at højreklikke på Main-klassen, for derefter at trykke på run. Eller man kan åbne selve Main-klassen for derefter at trykke Shift+F10.

- Kommandoprompt:
  For at benytte kommandoprompten til at køre programmet, skal man navigere til .jar-filens placering ved brugt af CD, også kaldet Indsæt filepath. Derefter skrives i kommandprompten "java -jar CDIO3.jar", hvilket vil starte programmet.

10.5 Versionstyring

Eftersom programmet er udviklet gennem flere versioner, er det nødvendigt at kontrollere hvorvidt den lokale version er opdateret eller ej. Dette gøres ved at åbne programmet i IntelliJ, for derefter at trykke CTRL+T. Denne keyboard shortcut opdaterer alle ændringer der er i programmet som er blevet commitet til programmets "remote repository".

Ønsker man at kontrollerer hvorvidt man kigger på den stabile 'master' branch, er det en sektion i nederste venstre hjørne i IntelliJ der hedder "9: Git". Trykker man på den, åbnes et vindue der indeholder information om diverse branches. For at sikre sig man er på 'master' branchen, skal man trykke på 'master' under remote sektionen, for derefter at trykke på "checkout". Hvis der er et bogmærke ikon ved siden af branchen, fortæller det at det er den nuværende branch der bliver kigget på.
Hvis ønsket er det også muligt at kigge på den branch som bliver brugt til videre udvikling af systemet. Her skal det samme gentages, dog skal der i stedet trykkes på 'Development' branchen.

Det er muligt at se historikken over rapporten med følgende link:
https://www.overleaf.com/1747543364zrbvsykvjbgj
Linket kræver at man har en overleaf bruger, men en sådan bruger kan oprettes gratis. Inden i overleaf skal man trykke på knappen **History** oppe i højre hjørne. Ved at bruge linket bliver det muligt at se hvornår hvilket billeder er uploadet i rapporten, og det er muligt at se, hvordan rapporten er blevet skrevet fra start til slut.

11 Projektplanlægning

Igennem udviklingen af projektet har vi fokuseret på et agilt og iterativt forløb samt et fleksibelt samarbejde. Det vil sige, at der under udviklingen, har været funktionelle versioner af programmet, og vi har løbende lavet test, og været fleksible med hensyn til ændringer i kravsspecifikationen og rettelser dertil. Samtidig er vores planlægning og opgavefordeling forekommet løbende. Vores kommunikation forekom både over længere distancer samt ved planlagte møder. Her udnyttede vi muligheden for langdistancekommunikation ved hjælp af applikationer så som Messenger og Discord. Yderligere benyttede vi os af muligheden for at gøre brug af diverse lokaler og rum på DTU.

For at løse diverse problemstillinger og opgaver, har vi anvendt en Unified Process metodetilgang. Vores primære tilgang til systemets udvikling har været Use Case drevet. Vi har dermed påbegyndt projektet ved at udarbejde diverse analytiske og designmæssige artefakter af projektets opbygning, for derefter at benytte dette som fundament til videreudvikling af vores system.

12 Konklusion

Alt i alt har vi udviklet et system som er gennemtestet, således at det overholder de givne krav. Fra analyse- og designfase til implementation har vi dog været stødt på udfordringer i forhold til de oprindelige ideer. For eksempel har opbygningen af design klassediagrammet ikke kunne løse opgaven. Blandt andet har vi været nødt til at parse referencer til objekter rundt, hvilket bryder med couplingen i klassediagrammet. Dette var dog nødvendigt for at kunne lave funktionel software. Vi har besluttet, at det var vigtigere at levere software der opfyldte de angivne krav, end at levere et teoretisk korrekt stykke software, der ikke opfyldte kravene.
I fremtidige projekter ville vi med fordel kunne fokusere mere på hvordan data skal sendes rundt, ved alle programmets funktioner, inden vi begynder at designe sekvensdiagrammer og klassediagrammer da dette kan modvirke sådanne problemer.

13 Bilag

13.1 Spørgsmål

1. Lav et spil kunden har bedt om med de klasser I nu har? Hvad menes der med de klasser vi har?
   Svar: Det er der ment er at man skal genbruge tidligere klasser

2. Skal både de almindelige regler samt de avanceret regler implementeres således at man kan vælge hvilket regelsæt der bruges ved starten af spillet?
   Svar: Enten eller, at vælge i starten vil være godt.

3. Skal en af spillerne være bank, eller kan system være den faste bank?
   Svar: System som bank er ok! - endnu ikke set at en spiller bliver brugt som bank

4. Detaljeniveau i domænemodel?
   Svar: Alt fysisk giver god mening, dog undtaget bagateller som solgt skilte.

5. Skal Main klassen være med i Design klassediagrammet?
   Svar: Giver mening, men den er ikke vigtigt at have med.

6. Prison card? Tages ud eller bliver der med det samme lagt et nyt ind?
   Svar: Tages ud, indtil kortet er brugt.