



Danmarks  
Tekniske  
Universitet

62531 udviklingsmetoder til it-system

62532 versionsstyring og testmetoder

02312 indledende programmering

**Gruppe 23**



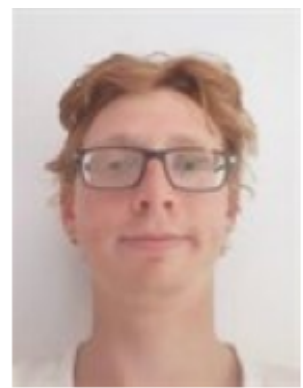
Hans Christian  
Leth-Nissen  
s205435



Kasper Falch  
Skov  
s205429



Phillip Igor Meyer  
Clausen  
s205468



Henrik Lynggaard  
Skindhøj  
s205464

## Github repo:

[https://github.com/Kageschwump/23\\_final](https://github.com/Kageschwump/23_final)

## Google Docs:

<https://docs.google.com/document/d/17TcM1jsXYAAHpsXZbpXwT5SxPJbn8eayOMi7kEr-yU/edit?usp=sharing>

## Timeregnskab

Tidsforbrug i timer	Uge 1, 2 og 3							
Navn/Dato	4-1-2021	5-1-2021	6-1-2021	7-1-2021	8-1-2021	9-1-2021	10-1-2021	Total for uge 1
Henrik Skindhøj	1	4	3	3	3	3	3	20
Kasper Falch Skov	1	4	3	3	3	2	3,5	19,5
Filip Igor Meyer	1	4	3	3	3	3	3,5	20,5
Hans Christian Leth-Nissen	1	4	3	4	3	3	3,5	21,5
Navn/Dato	11-1-2021	12-1-2021	13-1-2021	14-1-2021	15-1-2021	16-1-2021	17-1-2021	Total
Henrik Skindhøj	4	6	5	5	8	10	14	72
Kasper Falch Skov	2	6	5	6,5	8	10	14	71
Filip Igor Meyer	4	6	5	5	8	10	14	72,5
Hans Christian Leth-Nissen	4	6	5	6,5	8	10	12	73
Navn/Dato	18-1-2021							
Henrik Skindhøj	2							
Kasper Falch Skov	2							
Filip Igor Meyer	2							
Hans Christian Leth-Nissen	2							
Sum af alle								296,5

## Abstract

In this project, we have been asked by a customer to further develop our Monopoly Junior game, and make it into a normal Monopoly game. We have done this by following the given guidelines, provided by the client, and having weekly meetings with the client. By having the client be a part of the process of creating the program, we have made sure that the client is satisfied with the finished product. We started the project by developing artifacts and requirements for the program and afterward made a program, which satisfied the client's guidelines. To make sure our program functions as intended, we have made tests of certain functions in the program. We arrive at the conclusion that our program is a functioning Monopoly game that has been tested and follows the set requirements.

# Indholdsfortegnelse

<b>Github repo:</b>	<b>2</b>
<b>Google Docs:</b>	<b>2</b>
<b>Timeregnskab</b>	<b>2</b>
<b>Abstract</b>	<b>2</b>
<b>Indholdsfortegnelse</b>	<b>3</b>
<b>Indledning</b>	<b>5</b>
<b>Kravspecifikation</b>	<b>5</b>
F.U.R.P.S.	6
Functionality - must have	6
Functionality - nice to have	7
Usability	8
Reliability	8
Performance	8
Supportability	8
<b>Analyse</b>	<b>9</b>
Feature list	9
Use case diagram	9
Use cases	10
Domæne model	17
System sekvens diagram	18
Aktivitetsdiagram	20
<b>Design</b>	<b>21</b>
Sekvensdiagram	21
Spiller oprettelse	22
Klassediagram	23
<b>Implementering</b>	<b>24</b>
Test driven development	27
<b>Dokumentation</b>	<b>27</b>
Dokumentation for brug af GRASP principper	29
Creator	29
Information expert	29
Controller	30
Low coupling	31
High Cohesion	32

<b>Test</b>	<b>33</b>
Code Coverage	33
Unit test	34
<b>Projektplanlægning</b>	<b>44</b>
Gantt-kort	44
Feature Branching	44
Konfigurationsstyring	45
<b>Konklusion</b>	<b>46</b>
<b>Bibliografi</b>	<b>47</b>
<b>Bilag 1: Gantt Kort</b>	<b>48</b>
<b>Bilag 2: Sekvensdiagram</b>	<b>49</b>
<b>Bilag 3: Aktivitetsdiagram</b>	<b>50</b>
<b>Bilag 4: Design klassesdiagram</b>	<b>51</b>

## 1. Indledning

Vi har fået til opgave at udarbejde et færdigt “dansk matadorspil”. Til opgaven har vi fået tildelt spillets regler, og spillebræt, og vi har så haft mulighed for selv at vælge hvilket regler og funktioner der vil være mest vigtigt for et funktionelt matadorspil. Vi vil i løbet af projektet deltage i to møder med kunden, hvor vi vil vise hvor langt vi er kommet, hvilke ting vi mener vi kan nå at få lavet til de givne deadlines, og hvilke funktioner vi vil vælge fra, for at kunne forhindre forsinkelser i tidsplanen. Vi vil omskrive spillets regler til en række krav, som vi vil opstille efter FURPS-princippet. Vi vil ud fra disse krav udarbejde en række fully-dressed use cases. Vi vil dernæst opstille passende klasser, som vi mener er relevante for programmet, og ud fra dem vil vi lave de nødvendige artefakter. Vi vil i projektet lave system tests, for at sikre os at spillet virker som forventet, samt lave brugertest med en person uden tidligere kode-erfaring, for at sikre brugervenlighed. Der vil blive udført en række unit-test, for at sikre at visse funktioner virker som forventet. I programmet vil vi tage brug af en GUI som vi har fået givet af kunden igennem et Maven-repository. Vi forventer at få lavet et testet og færdigt program, som har tildelt en rapport med artefakter der beskriver programmets forløb

## 2. Kravspecifikation

Vi har i gruppen diskuteret om hvordan vores krav skal opdeles, og da dette program gerne skulle indeholde flere features end vi har lavet i de andre programmer, er vi kommet frem til at vi opdeler vores funktionelle krav. De funktionelle krav vil blive opdelt efter de mest essentielle krav, som vi mener gør et Matadorspil sjovt at spille, efterfulgt af de krav der nødvendigvis ikke er essentielle, men giver programmet lidt spænding.

En grund til denne opdeling er kravet for kunden om at det færdiglavede program, hellere skal indeholde færre implementerede features der virker ufejlbarligt, end et program fyldt med forskellige features der ikke virker.

En anden grund til opdeling af de funktionelle krav, kommer på baggrund af første møde med kunden, hvor de krav der var blevet formuleret fra vores side af, ikke stemte overens med kundens visioner for hvilken retning projektet skulle gå.

Resten af F.U.R.P.S.-kravene havde kunden ikke de store indvendinger imod, så de ville ikke blive opdelt på samme facon som de funktionelle krav.

## F.U.R.P.S.

### Functionality - must have

F1	Spillet skal kunne spilles mellem 3-6 spillere.
F2	Spillerne starter med 30.000 kr.
F3	En spiller slår med to terninger og rykker det antal felter, terningerne viser i øjne.
F4	Spilleren skal have mulighed for at vælge, om han/hun vil købe den ejendom man lander på eller ej.
F5	Hvis en spiller lander på et felt, der er ejet af en anden spiller, skal spilleren betale husleje.
F6	Der trækkes chancekort fra et dæk af 39 kort
F7	Spillet indeholder huse og hoteller til ejendommene, der er fire huse og et hotel til hver ejendom.
F8	Leje-summen forøges betydeligt ved opførelse af huse og hoteller.
F9	Man skal eje alle grunde i samme farve for at kunne bygge huse og hoteller
F10	En spiller går fallit, når denne ikke kan betale husleje eller er løbet tør for penge.
F11	Når en spiller lander eller passerer "Start" modtager spilleren 4000 kr af banken
F12	Spillerne begynder ved "Start"
F13	Spillerne skal rykke sig med uret
F14	Hvis en spiller ejer alle ejendomme af samme farvekode, fordobles huslejen.
F15	Spilleren skal altid rykke fremad, aldrig tilbage(En undtagelse hvis chancekort nr. 28(antal (2)) trækkes, rykkes der tre felter tilbage.)
F17	Man trækker et chancekort efter at have ramt "Prøv Lykken"

F18	Den spiller der går fallit til sidst har vundet spillet
-----	---

### Functionality - nice to have

F19	Sælge ejendomme tilbage til banken, skal ske inden man slår med terningerne, eller når man går konkurs, og stadig har ejendomme tilbage
F20	Hvis en spiller lander på et felt som spilleren selv ejer, skal spilleren ikke betale husleje
F21	Hoteller kan kun bygges når der er blevet bygget 4 huse på ejendommen som hotellet bygges på
F22	Huse og ejendomme kan sælges til den halve pris.
F23	Den yngste spiller starter
F24	Hvis en spiller lander på et "Indkomstskat" felt, skal spilleren vælge imellem at betale 4000 kr.
F25	Hvis en spiller lander på et "På besøg" felt, sker der ikke noget
F26	Hvis en spiller lander på "Parkering"-feltet, sker der ikke noget
F27	En spiller kommer i fængsel enten ved at lande på felt nr. 30 eller ved at trække chance kort nr. 40(antal 2)(Du fængsles, modtag ikke 4000 kr hvis du passerer start)
F28	Man kan komme ud af fængsel ved at betale 1000 kr, bruge et løsladselskort fra chance kortene eller slå et to ens og ryk med det samme frem som øjnene viser.
F29	Hvis en spiller lander på "Statsskat"-felt skal spilleren betale 2000 kr
F30	Bryggeri-felterne har en husleje der bliver regnet ud ved hjælp af øjnene spilleren har slået, hvis et bryggeri ejes, er huslejen summen af øjnene x100, og ved ejerskab af begge bryggerier er huslejen summen af øjnene x200.
F31	Rederi-felterne har en husleje der ved et rederi er på 500 kr, og vil så blive fordoblet for hvert ekstra rederi en spiller ejer.

## Usability

U1	Der skal benyttes GRASP-mønstre til udarbejdelse af projektet
----	---

## Reliability

Vi har valgt at genbruge reliability kravene fra CDIO 3, da de er tilstrækkelige for at sikre at vores program bliver testet ordentligt.

R1	Der skal laves mindst 3 testcases med tilhørende fremgangsmåder, testprocedure og testrapporter.
R2	Der skal laves mindst én JUnit test til centrale metoder.
R3	Der skal laves mindst en brugertest. Testen skal udføres med en der ikke har kendskab til kodning.

## Performance

P1	Programmet skal kunne køre på DTUs maskiner, uden påfaldende forsinkelser
----	---

## Supportability

S1	Projektet skal udarbejdes i Git, da kunden skal kunne følge med i udviklingen af spillet.
S2	Koden skal committes med git for hver delopgave der bliver løst, eller mindst en gang i timen.
S3	Hvert commit skal indeholde en beskrivende commit besked.



### 3. Analyse

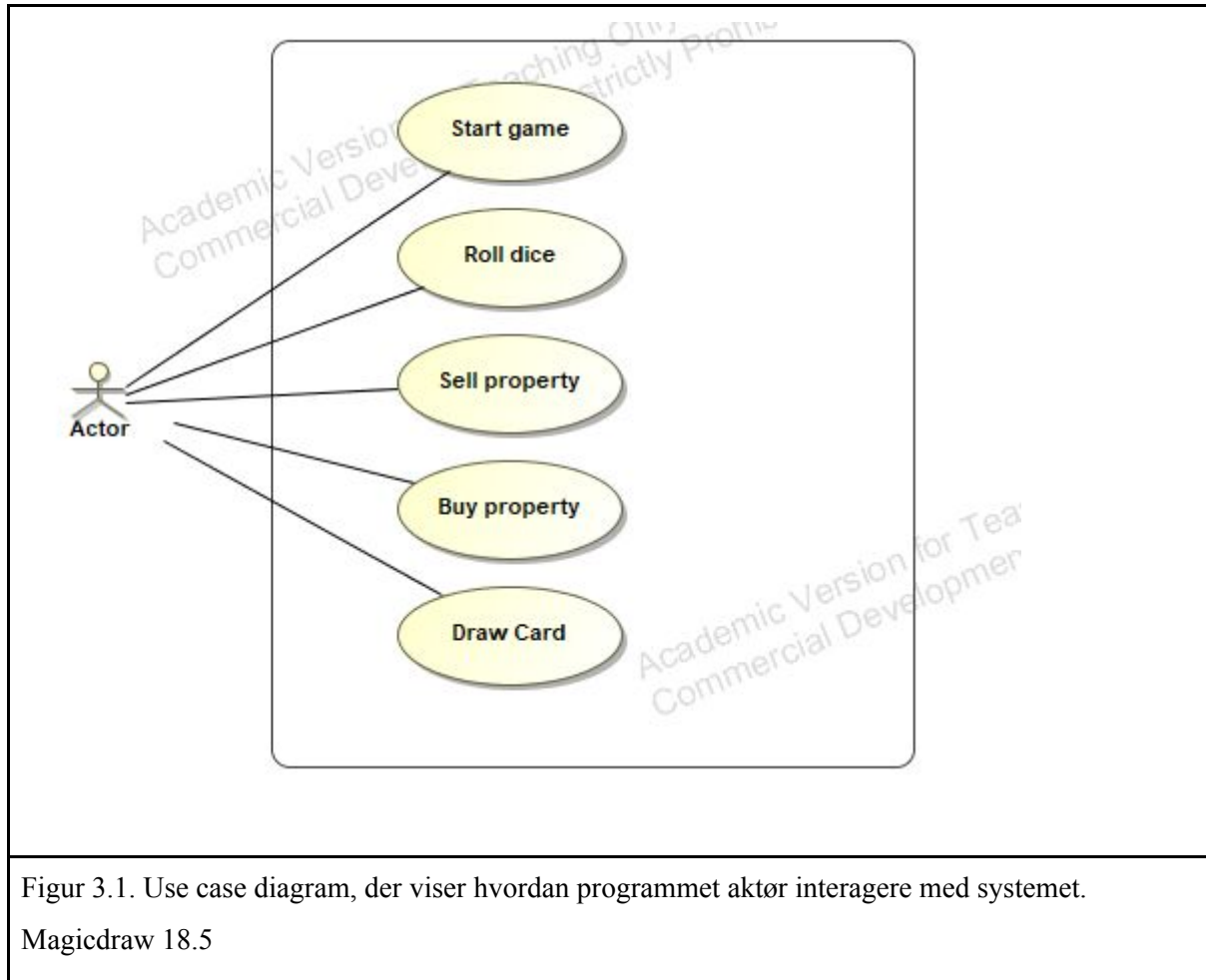
#### Feature list

Matador spillet vil indeholde en række forskellige features. Dette spils features kan analyseres ved brug af en feature list. En feature er kvalificeret ved at bære en funktion i spillet. Som eksempel er køb af hus og husleje features, samt at trække kort fra et dæk.

1.	Terningkast.
2.	Valg af antal spiller
3.	Spiller avatar præferencer
4.	Pengebeholdning
5.	Felt funktioner
6.	Chance kort og funktioner
7.	Hus og hotel køb
8.	Salg af ejendomme

## Use case diagram

Vi har benyttet de samme use cases som fra CDIO 3, med nogle få rettelser og tilføjelser så det passer til CDIO Final projektet. Vi er kommet frem til 5 use cases, da det er de eneste der er behov for, da spillet kører af sig selv, ud fra nogle foruddefinerede regler. Nedenunder ses først og fremmest det udarbejdede Use case-diagram, og derefter vores fem fully dressed use cases.



Figur 3.1. Use case diagram, der viser hvordan programmet aktør interagerer med systemet.  
Magicdraw 18.5

## Use cases

Use case section:	Beskrivelse
Navn	UC 1.1 - Start game
Scope	System
Level	User Goal
Primær aktør	Player
Preconditions	Spillet er ikke allerede igang
Postconditions	Matador er sat op, og klar til at spille
Main Success Scenario	1: Spiller: Kører programmet 2: System: Spørger om antal spillere 3: Player: Indtaster antal spillere 4: System: Opretter spillere 5: System: Beder om navne 6: Player: Indtaster navne 7: System: Beder om alder 8: Player: Indtaster alder 9: System: Starter spillet
Extensions	NaN
Frequency of occurrence	Hver gang der skal startes et nyt spil

Use case section:	Beskrivelse
Navn	UC 1.2 - Roll dice
Scope	System

Level	User Goal
Primær aktør	Player
Preconditions	Matador er oprettet/startet
Postconditions	Der er rullet en terning
Main Success Scenario	<p>1: System: Venter på at spilleren klikker 'Rul'</p> <p>2: Player: Klikker på "Rul", og kaster terningerne</p> <p>3: System: Genererer et tal fra 2-12, læser øjnene, og lægger øjnene sammen med spillerens placeringen på brættet..</p> <p>4: Spiller: Opfylder pligten som feltet angiver.</p> <p>5: System: Prompter en ny spiller til at kaste med terningerne</p>
Extensions	<p>4.1.1: Spilleren: Lander på 'Prison-square'</p> <p>4.1.2: System: Flytter spilleren til fængslet</p> <p>4.2.1: Spilleren: Lander på en 'Chancecard-square'</p> <p>4.2.2: System: Prompter UC 1.3</p> <p>4.3.1: Spilleren: Passerer start</p> <p>4.3.2: System: Tildeler spilleren 4000 kr.</p> <p>4.4.1: Spiller: Lander på en anden spillers 'Property'-felt</p>

	<p>4.4.2: System: Sender penge til spilleren der ejer 'Property'-felt</p> <p>4.5.1: Player: Lander på en 'Property-square' der ikke er ejet af en spiller</p> <p>4.5.2: Player: Får <b>valget</b> om man vil købe den 'Property-square' eller ej.</p> <p>4.5.3: System: Ved køb tildeler den 'Property-square' til spilleren</p> <p>4.5.4: System: Hvis "Property-square" ikke bliver købt, prompter ny spiller til at kaste med terningerne.</p> <p>4.6.1: Spiller: Lander på 'Visit-Jail-Square'</p> <p>4.6.2: System: Går videre til næste spillers tur</p> <p>4.7.1: Spiller: Lander på 'ParkingSquare'</p> <p>4.7.2: System: Går videre til næste spillers tur</p> <p>4.8.1: Spiller: Lander på 'BetaltSkatSquare'.</p> <p>4.8.2: Spiller: Betaler beløbet.</p> <p>4.8.2.1.1: Spiller: Skal betale indkomstskat</p> <p>4.8.2.1.2: System: Giver spilleren valget imellem at betale 10% af sin formue, eller 4000 kr.</p> <p>4.8.2.1.3: Spiller: Tager sit valg, og betaler beløbet.</p> <p>4.8.2.2.1: Spiller: Skal betale ekstraordinær statsskat.</p> <p>4.8.2.2.2: System: Trækker 4000 kr. fra</p>
--	---

	spillerens pengebeholdning
Frequency of occurrence	Hver gang en spiller skal kaste en terning

Use case section:	Beskrivelse
Navn	UC 1.3 - Draw card
Scope	System
Level	User Goal
Primær aktør	Player
Preconditions	En player lander på "ChanceCardSquare"
Postconditions	Chancekortets betingelser er udført
Main Success Scenario	1: System: Trækker et kort for spilleren 2: Spiller trykker "ok" 3: System: Udfører kortets betingelser
Extensions	NaN
Frequency of occurrence	Hver gang en player lander på "ChanceCardSquare"

Use case section:	Beskrivelse
-------------------	-------------

Navn	UC 1.4 - Buy Property
Scope	System
Level	User Goal
Primær aktør	Player
Preconditions	En spiller er landet på et felt med en ledig Ejendom. Spilleren har nok penge
Postconditions	En spiller har købt en ejendom.
Main Success Scenario	<p>1: Spiller: Lander på en ledig ejendom</p> <p>2: System: Spørger om spilleren vil købe ejendommen eller ej</p> <p>3: Player: Vælger at købe ejendommen</p> <p>4: Player: Betaler det beløb det står på det felt spilleren landede på</p> <p>5: System: Trækker beløbet fra spillerens pengebeholdning og tildeler ejendommen til spilleren.</p>
Extensions	<p>3.1.1: Spiller: Vælger ikke at købe ejendommen</p> <p>3.1.2: System: Giver turen videre til næste spiller</p> <p>3.2.1 System: Tjekker om spilleren allerede ejer et hus på feltet, og tjekker samtidig om spilleren ejer et hus på alle de andre felter i samme farve</p> <p>3.2.2: System: Spørger spilleren om personen vil købe et hus mere og stille på feltet</p> <p>3.2.3: Spiller: Vælger at købe et hus mere</p>

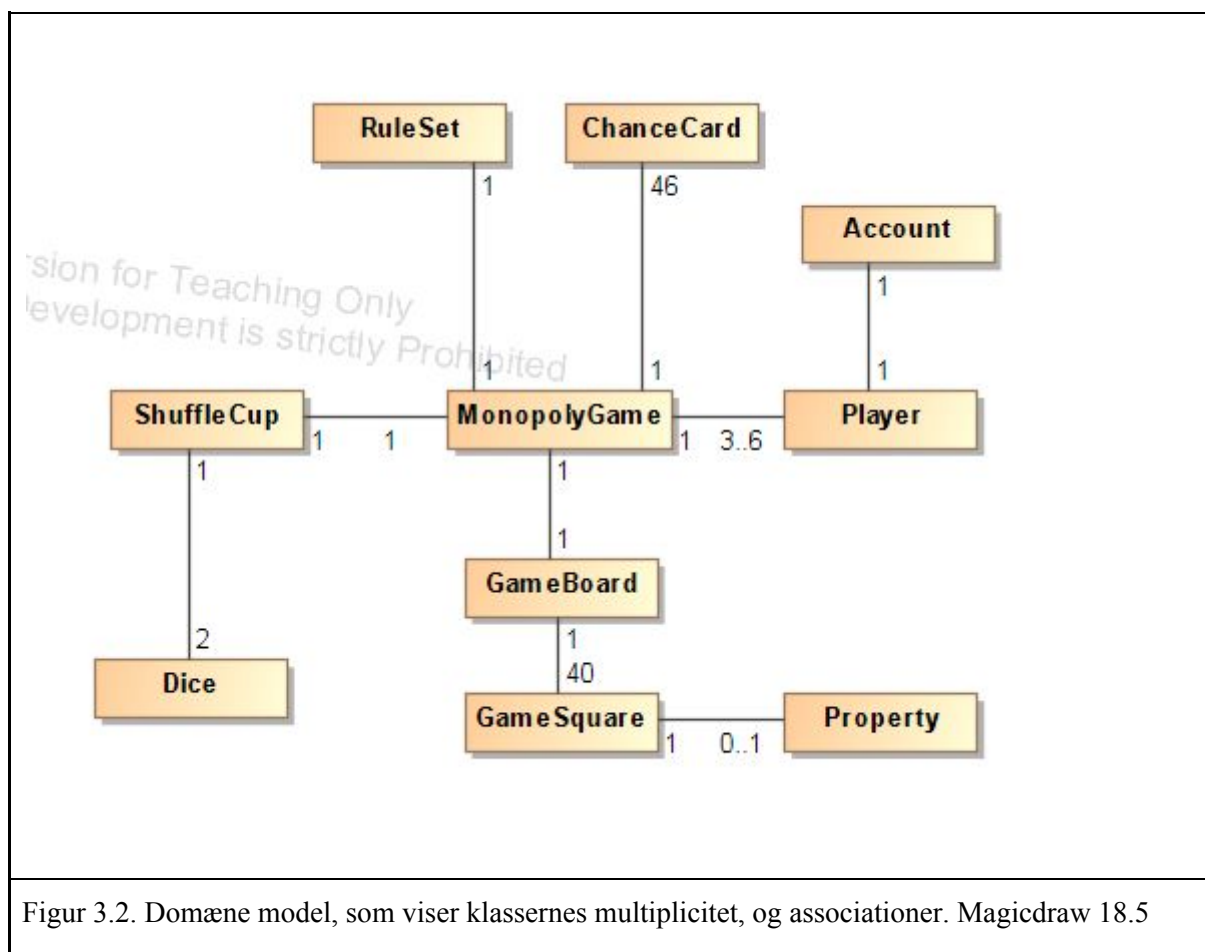
	<p>3.2.4: System: Trækker beløbet fra spillerens pengebeholdning og tildeler ejendommen til spilleren.</p> <p>3.2.4.1: Spiller: Har 4 huse på feltet i forvejen.</p> <p>3.2.4.2: Spiller: Får et hotel på feltet, istedet for de 5 huse.</p>
Frequency of occurrence	Hver gang en spiller lander på en ledig ejendom

Use case section:	Beskrivelse
Navn	UC 1.5 - Sell Property
Scope	System
Level	User Goal
Primær aktør	Player
Preconditions	Det er den pågældende spillers tur
Postconditions	Spilleren ejer ikke længere ejendommen, som spilleren har valgt at sælge, og har modtaget halvdelen af de penge som der blev betalt.
Main Success Scenario	<p>1: Spiller: Trykker på 'sælg ejendom' knappen</p> <p>2: System: Fjerner huset fra spillepladen</p> <p>3: System: Giver spilleren halvdelen af pengene som ejendommen koster</p>
Extensions	NaN



Frequency of occurrence	Hver gang en spiller vælger at sælge en ejendom
-------------------------	---

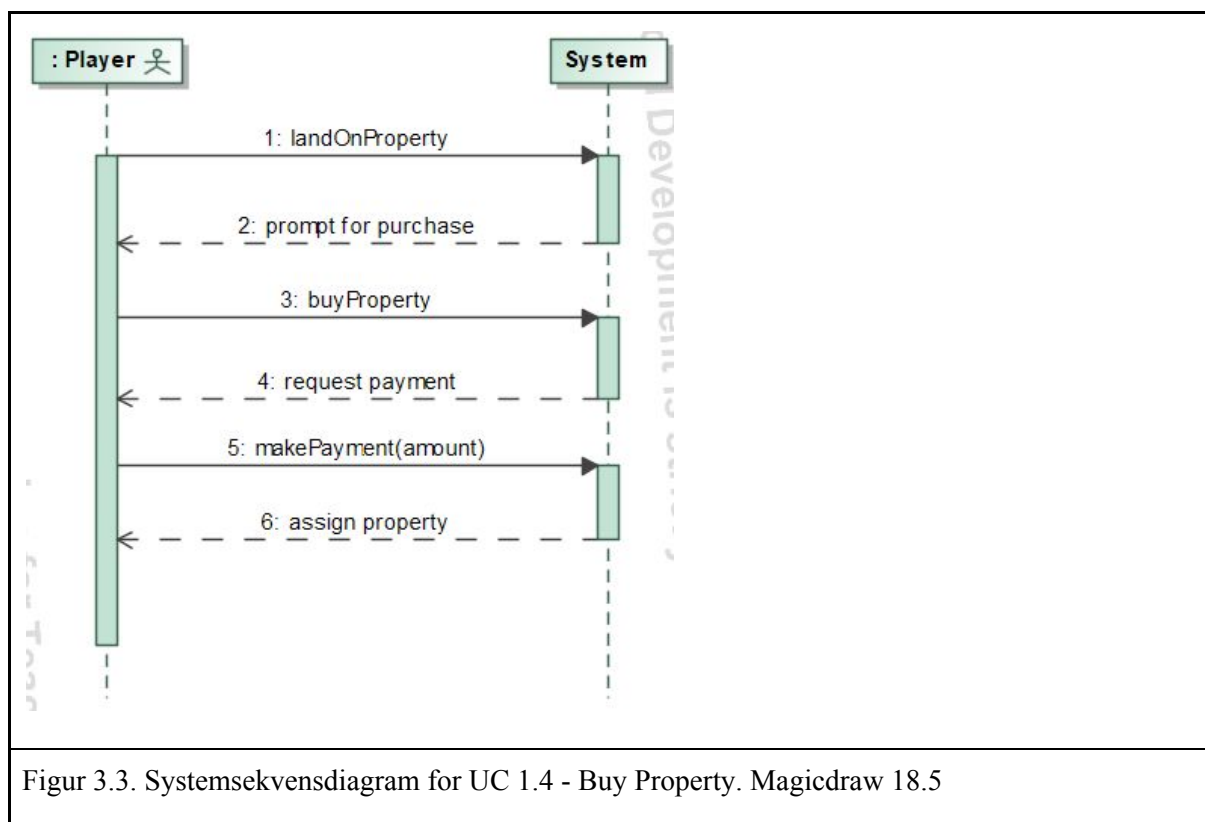
## Domæne model



Objekt orienteret programmering (OOP) er netop smart ved realisering af fysiske objekter og opgaver. Domænemodellen er produktet af en tankeleg. Ved åbning af et fysisk matadorspil, findes der objekter i pakken. Pakken indeholder spillebrikkerne, spillebrættet, Monopoly-kort og terninger. Så findes der selvfølgelig også et papir med reglerne angivet. Alle disse objekter kan angives, som klasser i vores system. Flere af vores klasser i domæne modellen vil altså indeholde endnu flere klasser i vores Design klassediagram.

Ud fra den samlede information af det fysiske bræt, kan vi også angive nogle attributter i systemet. En terningerne har tegn på hver af sine sider, hvilket kan angives som “face values”. Spillerne bruger Monopoly-penge, hvilket kan føre til en “account” klasse.

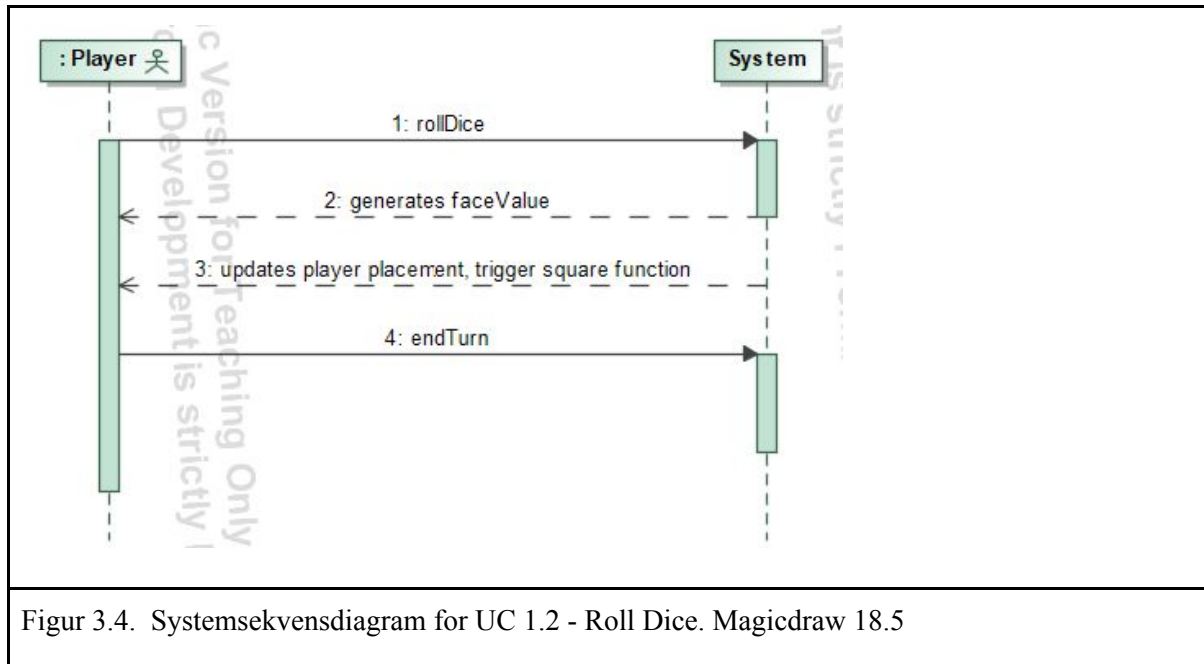
## System sekvens diagram



Før spilleren kan købe en ejendom, skal visse kriterier opfyldes. Spilleren skal stå på den pågældende ejendoms brik. Ejendommen må ikke allerede været ejet af modstanderen. Spilleren der ønsker at købe ejendommen, skal have tilstrækkelige midler for at kunne foretage købet af den ønskede ejendom.

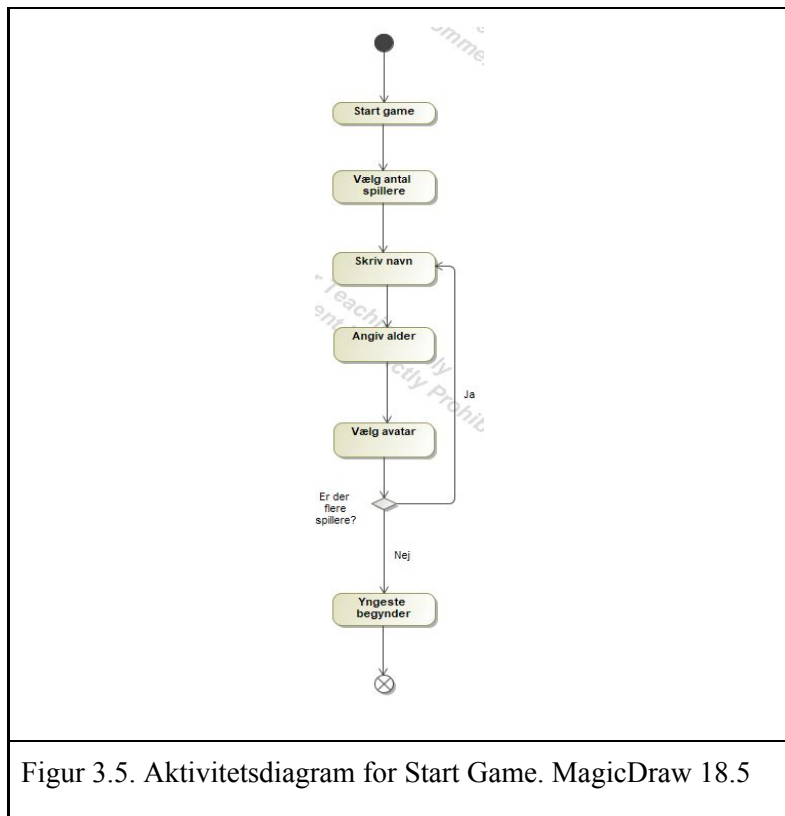
Når alle kriterier er opfyldt, kan processen begynde. Her er interaktionen mellem spilleren og systemet opsat til succes-scenariet. Efter system-interaktionen, slutter spillerens tur.

Systemet sender beskeder omkring købet af ejendommen, når spilleren aktiverer funktionen buyProperty. Når købet er fuldent, ændrer systemet spillerens pengebeholdning, og turen afsluttes.



I system sekvensdiagrammet for Roll dice, vises det hvordan den primære aktør “player” snakker med systemet når en spiller starter sin tur og skal rulle med en terning. Når spilleren ruller med terningerne, giver systemet en tilfældighed faceValue, fra et til seks, til de to terninger og lægger dem sammen. Systemet opdaterer straks spillerens placering og udfører den pligt som er tildelt det felt spilleren er landet på, og spillet går så videre til næste tur.

## Aktivitetsdiagram

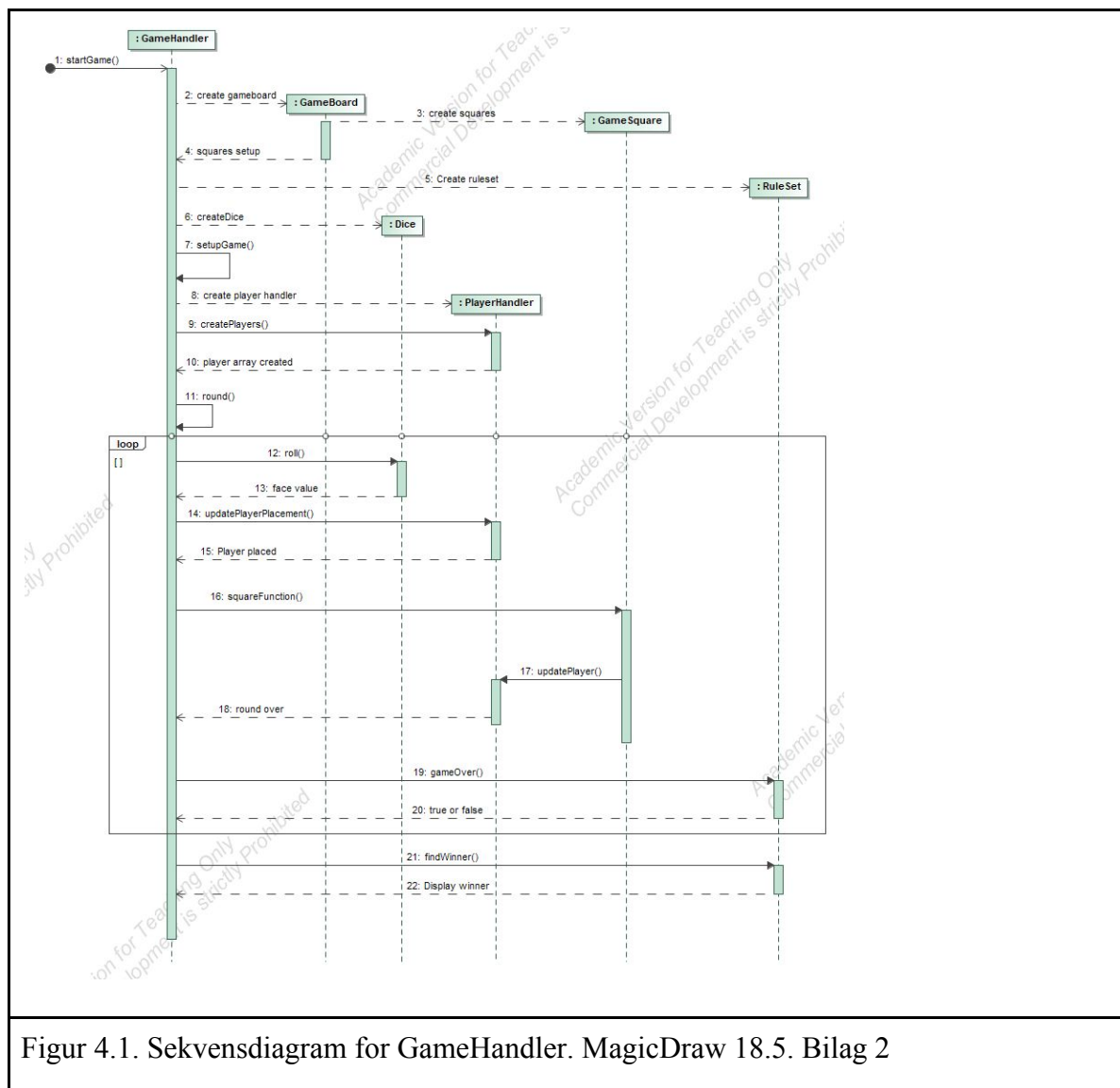


Figur 3.5. Aktivitetsdiagram for Start Game. MagicDraw 18.5

Programmets start er bedst beskrevet ved aktivitetsdiagram..Denne proces opstiller vores tomme spiller objekter, efter man har valgt antallet af spillere. Se kapitel 4, “spilleroprettelse”.

## 4. Design

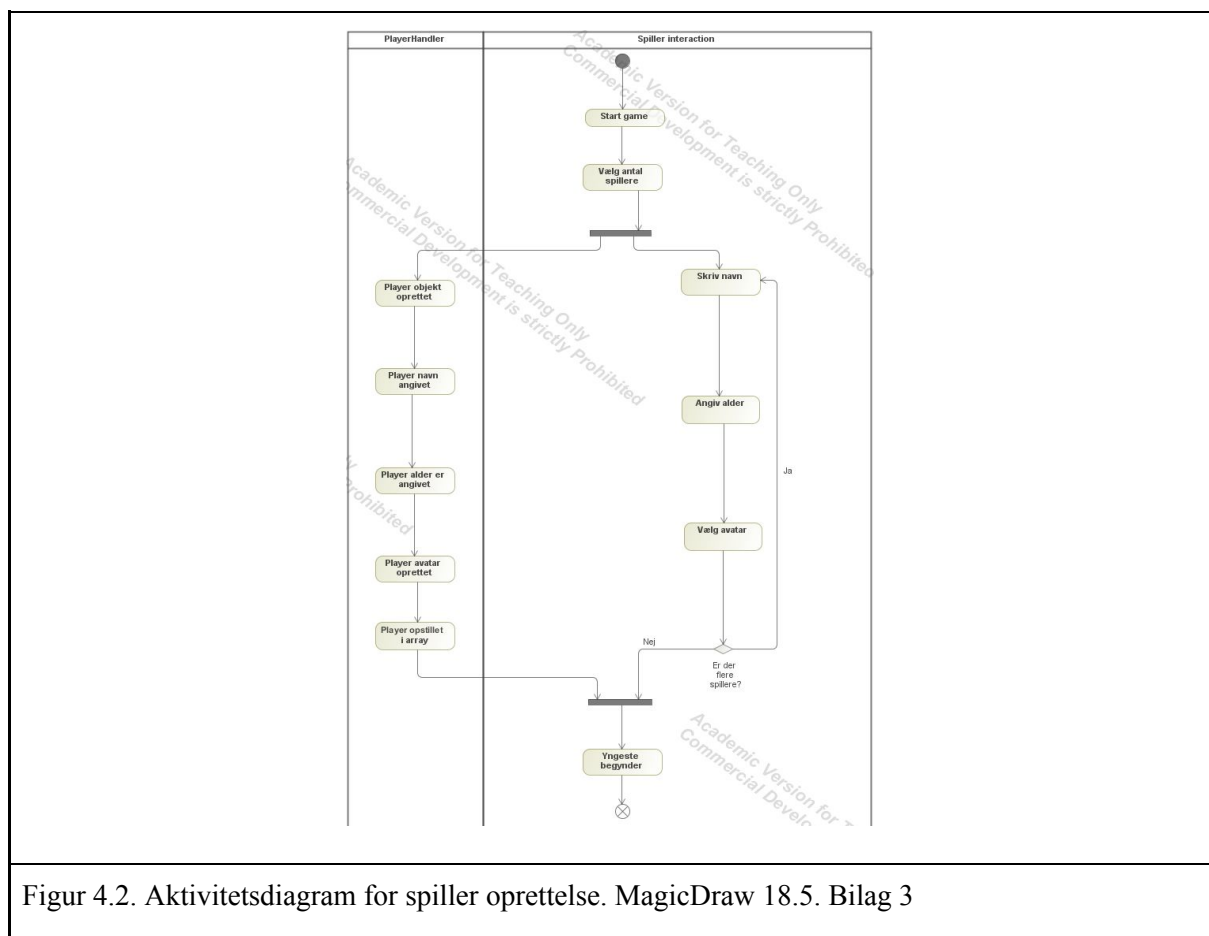
### Sekvensdiagram



GameHandler håndterer for meget. Vi undersøgte for alternative løsninger til GameHandler problemet, men ingen løsninger er logisk mulige. Vores program består af netop 5 forskellige dele. Brættet, spillerne, terningerne, kortene og reglerne. Før en spiller kan rykke felter, så skal informationen hentes fra brættet og terningerne. Når spilleren står på et felt, kræver det informationer fra regelsættet for at kunne bestemme, hvilke forudsætninger feltet består af.

I domænemodellen har vi MonopolyGame angivet som klasse. Domænemodellen viser sammensætningen af potentielle klasser. MonopolyGame-klassen er klassen som hæfter alle dele af spillet sammen, hvilket kan relateres til vores GameHandler. Visse tilfælde kræver alle dele af programmet. Et scenarie kunne være et “Ryk felt” chancekort. Spilleren er landet på et chancekort-felt, hvor både chancekort, brættet, spilleren, terningerne og reglerne bliver brugt. Selvom vi laver flere handlers, så vil arbejdsfordelingen ikke se anderledes ud.

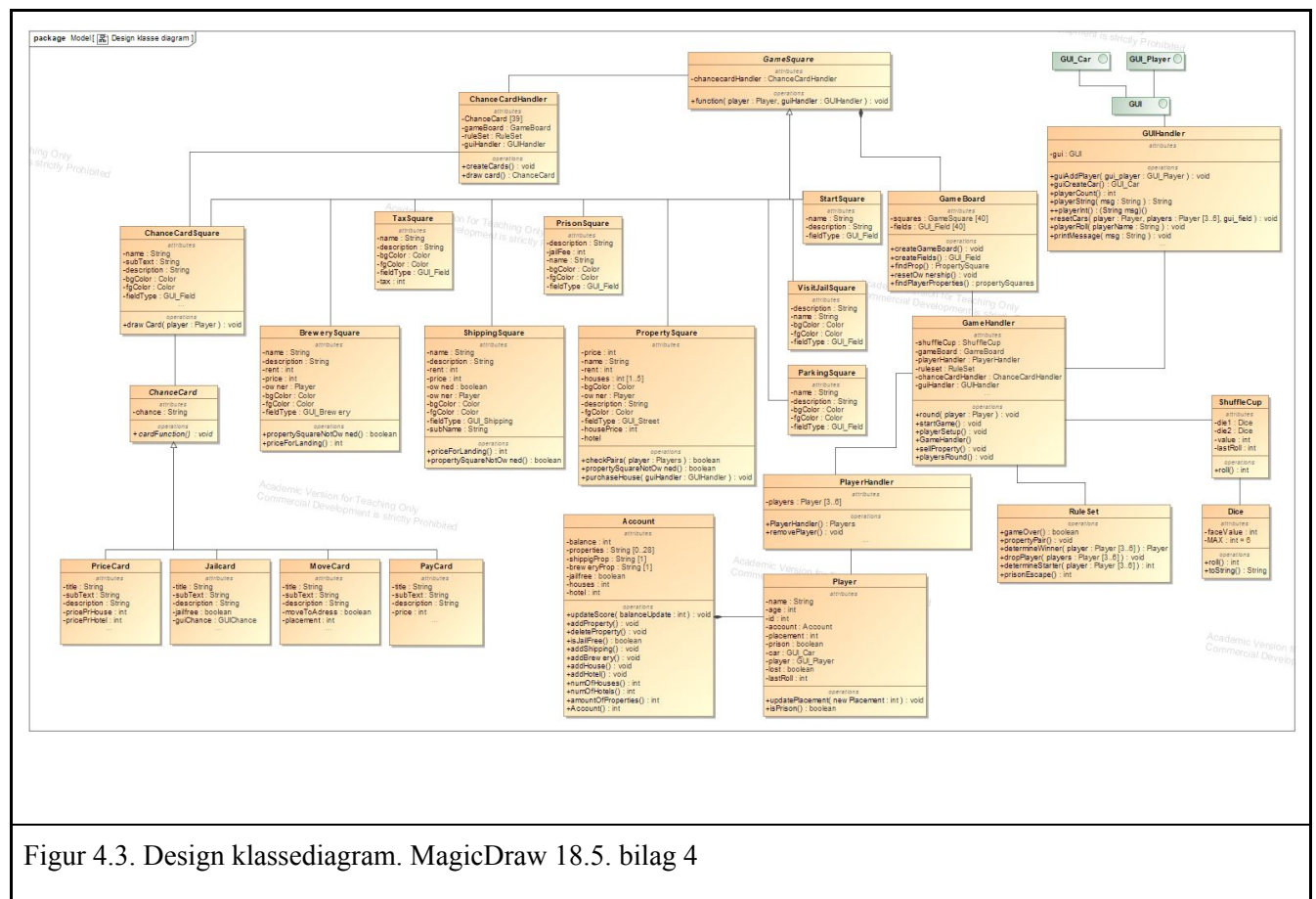
## Spiller oprettelse



Figur 4.2. Aktivitetsdiagram for spiller oprettelse. MagicDraw 18.5. Bilag 3

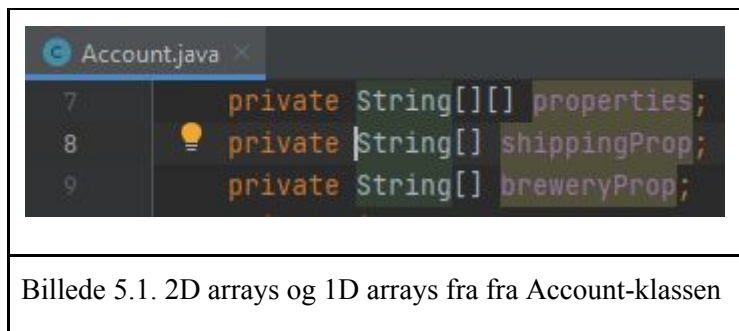
Spiller oprettelsen er lidt speciel i forhold til den resterende kode. I begyndelsen opretter vi data, hvor vi senere kun kan redigere i den. Et aktivitetsdiagram kan vise processen, men vi har undladt at vise hvordan informationen bliver delt mellem diagrammets swimlane fork, da det ville se forvirrende ud. Swimlanes er en metode man bruger i aktivitetsdiagrammer til at vise hvilke objekter der har ansvar over hvilke aktiviteter.

Når PlayerHandler-klassen skal instantieres, kræver klassen spiller-antallet som en parameter. Denne information bruges til at bestemme hvilken spillers tur det er, samt hvilket id hver enkelt spiller har.



## 5. Implementering

Den startende spiller bliver valgt ud fra alder, den yngste spiller starter. Hvis to spillere er lige gamle starter den spiller der er nævnt først.



Vi har brugt arrays til at lagre information i objekterne. Her holder spillerens “account” styr på spillerens ejendomme, siden huslejen har indflydelse på spillerens indkomst. Billede 5.1 viser “String[][]”, som er et 2-dimensionelt array. De farvekodet ejendomme har ekstra nødvendig information, der beskriver deres funktion i det overordnede program. Når en spiller ejer alle ejendommene af samme farvekode, så stiger huslejen for den alle ejendommen af samme farvekode til det dobbelte af den normale husleje. Farverne har derfor også indflydelse på spillerens indtægt og dermed spillerens account.

Et 2-dimensionelt array kan anses som en tabel. Tabellen forneden viser et 2D array.

Ejendoms navn	Ejendoms navn	Ejendoms navn
Farve	Farve	Farve

0	1	2
1	1	1

Logikken minder lidt om et excel dokument. Her arbejder programmøren med rækker og kolonner. Billede 5.1 viser hvordan man arbejder med 2-dimensionelle arrays i Java. Se arrayet som [kolonne][række]. Vi starter med index fra første aksen, derefter andenaksen.





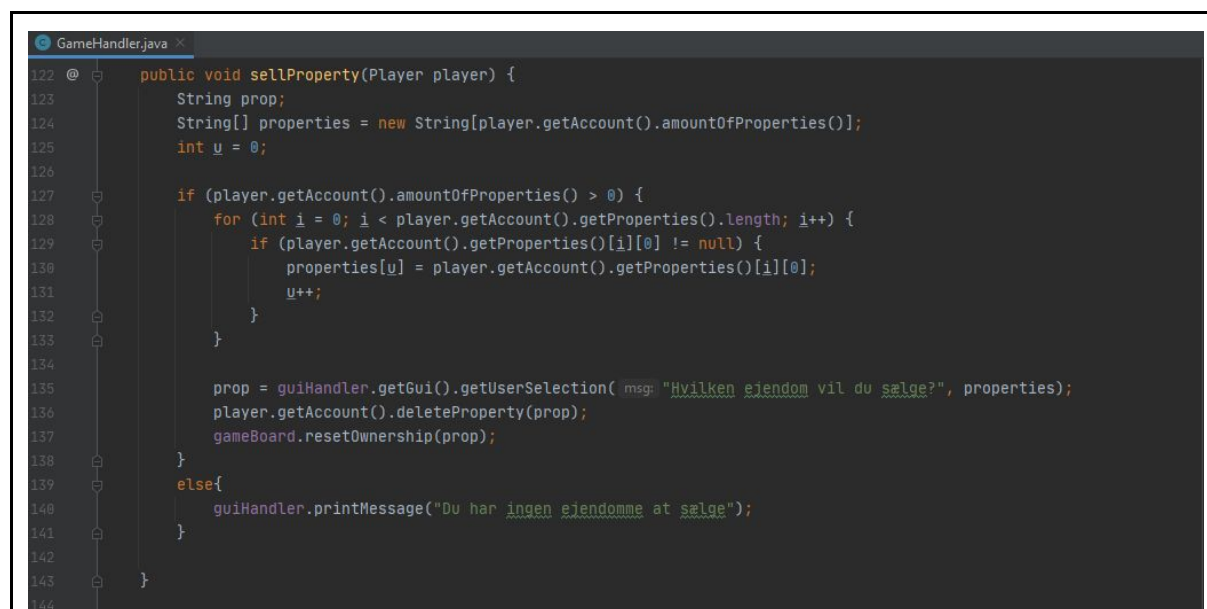
Metoden addProperty viser adderingen af en ejendom til en spiller. Her gemmer metoden ejendommens navn og farve i det 2-dimensionelle array.



I CDIO 3 dokumenterede vi nogle overvejelser til programmets struktur for dette projekt. Ved realisering af disse overvejelser fandt vi dem ikke overvældende. Et godt eksempel kunne være ejendommene, som indeholder deres ejere. Dette var ikke den bedste ide, men gjorde det lettere at håndtere transaktion mellem to spillere. Hvis en spiller lander på et "Property"-felt, som er ejet. Så har feltet al den information der skal til, for at kunne udføre transaktionen. Dette kan designes på en anderledes måde. En transaktion klasse kan være brugbar, hvis objektet har direkte association til alle spillerne (playerHandleren).

Vi implementerede også andre overvejelser i dette projekt. Felterne styrer funktionerne for spilleren, hvilket har sine fordele og ulemper. Fordelene er at vores felter kan give beskeder til spilleren. Ulempen er at vores objekter ikke kender til andre objekter af samme type. Denne form for kode er ikke GRASP venlig, siden det fører til forkert fordeling af ansvar mellem klasserne.

I dette projekt har vi valgt at bygge runden op således, at der er en while-løkke hvori spilleren har to valgmuligheder når runden starter, *rul* og *sælg*. Dette var ikke hvad vi havde i tankerne, da vi først udtænkte vores struktur til spillet. Vi havde regnet med at vi kunne genbruge vores runde-opbygning fra CDIO 3 projektet, men vi indså at dette ville blive et problem da vi skulle implementere salg af ejendomme. I while-løkken har vi en switch-case, som aflæser spillerens valg, enten "*rul*" eller "*sælg*", hvis *rul* vælges, fortsætter runden ved at terningerne bliver kastet og spilleren skal rykke det viste antal øjne, hvorefter funktionen af det felt som spilleren lander på udføres. Hvis *sælg* vælges går programmet ned og finder metoden *sellProperty*.



Billede 5.4: "*sellProperty*" metoden fra GameHandler klassen

Denne metode starter med at tjekke om spilleren overhovedet har nogle ejendomme med et if-statement. Hvis dette ikke er tilfældet bliver spilleren sendt tilbage til starten af while-løkken. Hvis spilleren derimod har ejendomme i sit *properties*- array, så har vi en

for-løkke, som kører hele spilleres 2D-property array igennem, og hvis pladsen i arrayet ikke er *null*, sætter vi navnet på ejendommen ind i et nyt array, *properties[]*, der kun er ligeså langt som antallet af ejendomme spilleren ejer. Dette array kan vi nu præsentere for spilleren ved hjælp af GUIens *getUserSelection*-metode. *getUserSelection* kan nemlig tage imod et *String*-array som valgmuligheder. Når spilleren har valgt hvilken ejendom der skal sælges, har vi en metode *resetOwnership* i *GameBoard*-klassen og metoden *deleteProperty* i *Account*-klassen, der står for at fjerne huse og kant-farve fra feltet, samt opdatere spillerens penge og property-array. Herefter bliver spilleren returneret til toppen af while-løkken igen, og kan da vælge om der skal sælges igen, eller om spilleren er klar til at kaste med terningerne.

Vi har valgt at man sælger ejendommen med eventuelle huse på en og samme tid. Dette valgte vi, da vi havde store problemer med at få trukket ejendomme med huse og ejendomme uden huse ud af spillerens property-array.

## Test driven development

Matador er et langt spil, som kan tage op til en time at gennemføre. Hermed følger mange specielle scenarier. Bryggerierne i Matador er interessant, siden terningens øjne har konsekvenser for huslejen spilleren skal af med. Scenariet er besværligt at teste ved gennemgang af spillet. Her opstiller vi tests, der udfordrer metodernes logik. Unit tests kan give kode en sandsynlighed for succes.

De fleste scenarier i slutningen af spillet er blevet unit testet. Alle er beskrevet i test afsnittet. Fremgangsmåden består i at skrive simple metoder. Unit tests gennemgår metoderne, og returnerer resultatet. Vi har forventninger til, hvordan resultatet vil se ud. Hvis vores forventning og resultatet stemmer overens, så er testen en succes. Hvis testen derimod fejler, så har vi lært fra den og kan approximere metoden yderligere.

## 6. Dokumentation

I vores projekt kom vi ud for situationer hvor vi ikke havde en fungerende version af programmet, hvor alle vores seneste implementerede funktioner indgik. Dette har været et resultat af at vi ikke har meget vores fungerende feature branches ind i vores development

branch. Nedenunder kan det ses at vi ikke har kunnet starte spillet og teste om det virker, da nogle andre gruppemedlemmer har været i gang med at implementere en ny feature.

```
C:\Users\henri\IdeaProjects\23_final\src\main\java\Controller\GameHandler.java:144:40  
java: method sellHouses in class Squaretype.PropertySquare cannot be applied to given types;  
  required: int  
  found: no arguments  
  reason: actual and formal argument lists differ in length
```

Billede 6.1. Dokumentation for en fejl der er opstået af en metode der ikke blevet lavet færdigt.

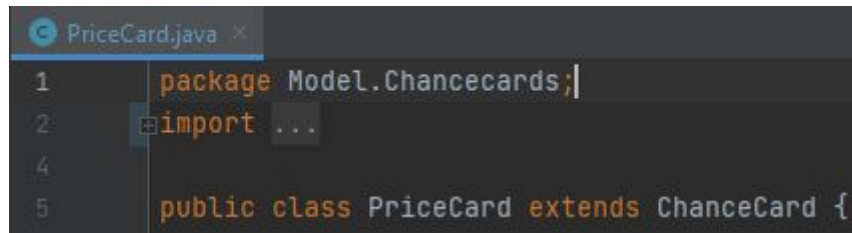
I vores program indgår der to superklasser, der har hver deres metoder, der bliver nedarvet til deres underklasser. Programmets to superklasser er *GameSquare* og *ChanceCard*. Vi har superklassen *GameSquare*, da vores program har ni forskellige typer af felter, der alle sammen har nogle af de samme metoder. Det giver derfor mening at have en abstrakt klasse der indeholder metoder, der nedarves af underklasserne.

Superklassen *ChanceCard*, der er behov for, da vores chancekort allesammen både har en beskrivelse og en given funktion. Det kan ses nedenunder hvordan vi har opstillet *ChanceCard*-klassen i IntelliJ IDEA.

```
ChanceCard.java x  
1 package Model;  
2 import Controller.GUIHandler;  
3  
4 public abstract class ChanceCard {  
5  
6     private String chance = "Chance kort!";  
7  
8     public abstract void cardFunction(Player player);  
9  
10    public abstract String getDesc();  
11  
12 }
```

Billede 6.2. Dokumentation for hvordan vi har opstillet vores ene superklasse *ChanceCard*

Nedenfor kan det ses hvordan vi har nedarvet *ChanceCard*-klassens metoder, til dens underklasser.

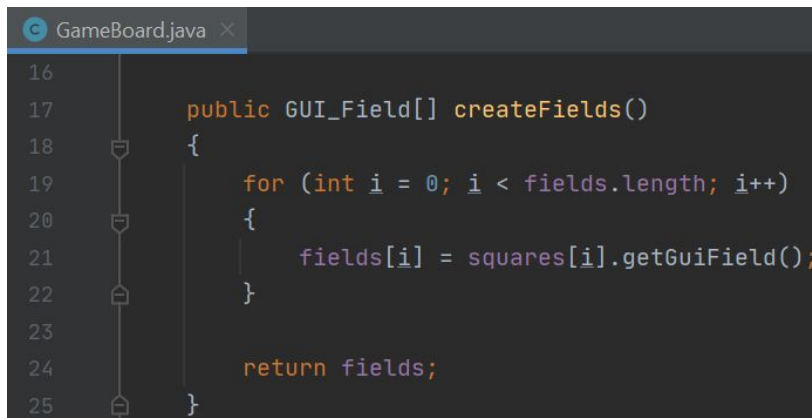


```
1 package Model.Chancecards;
2 import ...
4
5 public class PriceCard extends ChanceCard {
```

Billede 6.3. Et eksempel på en af *ChanceCard* klassens underklasser.

## Dokumentation for brug af GRASP principper

### Creator



```
16
17 public GUI_Field[] createFields()
18 {
19     for (int i = 0; i < fields.length; i++)
20     {
21         fields[i] = squares[i].getGuiField();
22     }
23
24     return fields;
25 }
```

Billede 6.4. Dokumentation for at *GameBoard*-klassen er en Creator-klasse.

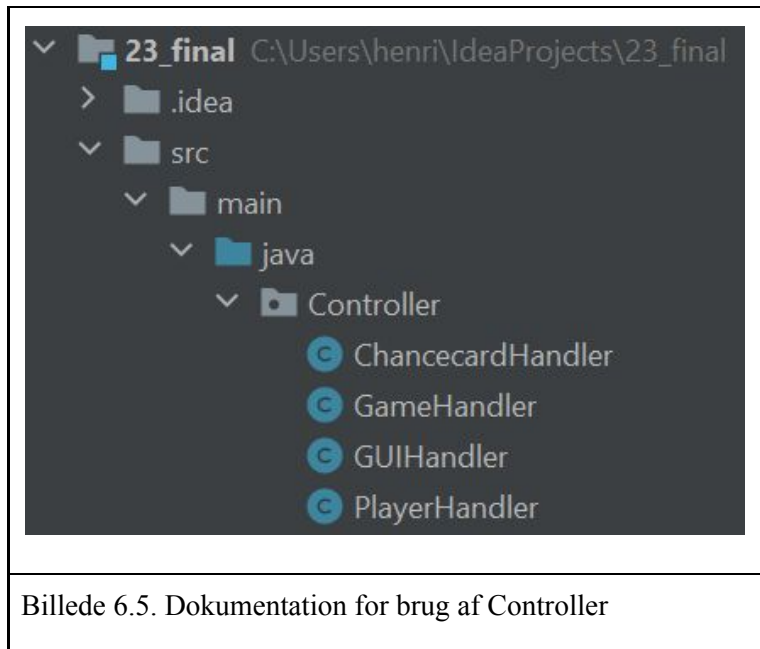
På ovenstående billede vises en af vores creator-klasser. *GameBoard* har metoden *createFields*, denne metode, er til for at kunne lave felterne i vores spil, og dette betyder at *GameBoard* er en creator-klasse, der har ansvaret for at lave fields.

### Information expert

I vores program har vi prøvet at overholde information expert princippet, ved kun at tildele information om klasser til andre klasser som har behov for den information. Vi har dog ikke

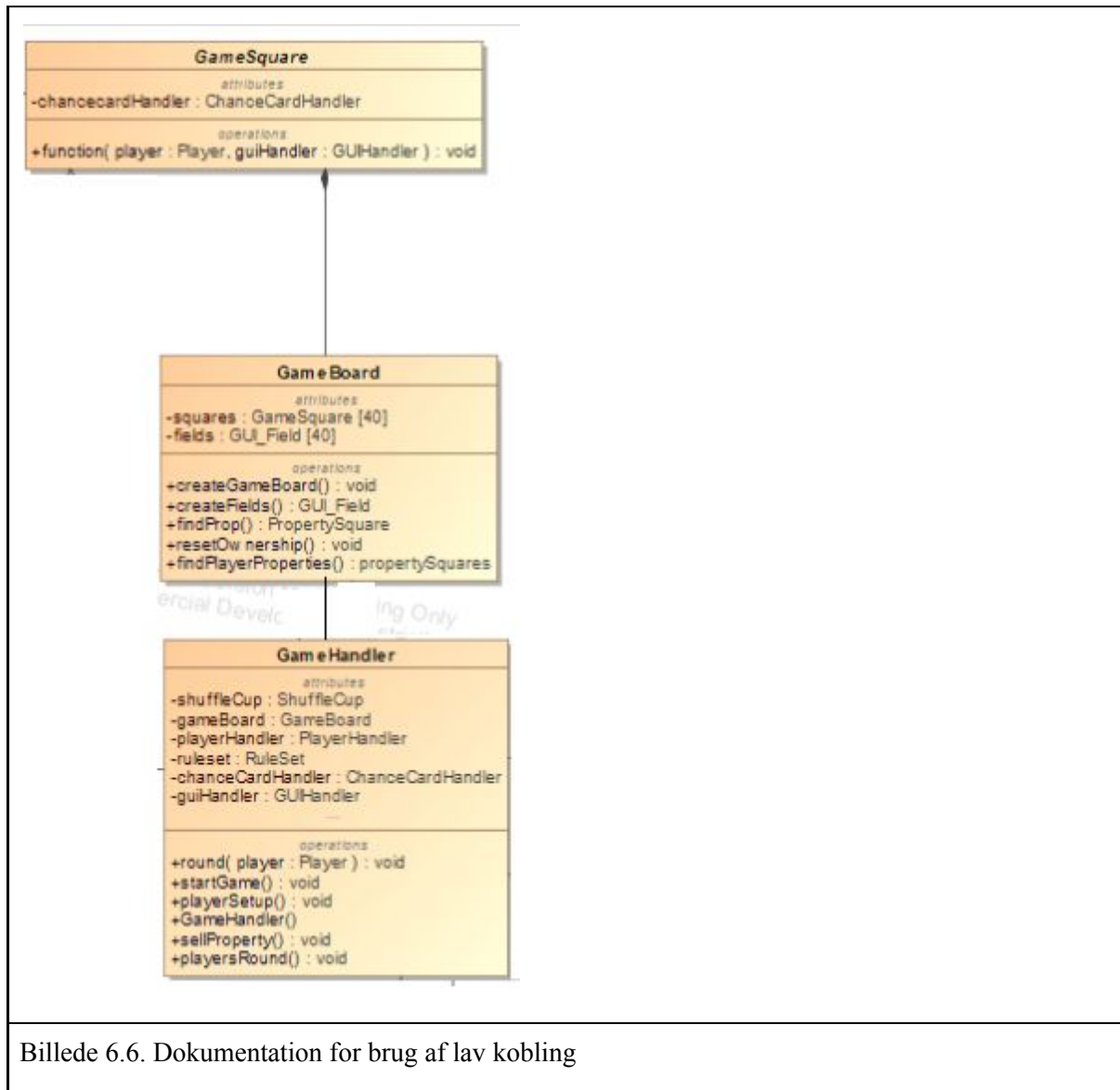
overholdt princippet i vores account klasse, da den er blevet tildelt mere information end en spillers konto burde have adgang til.

## Controller



Vi har i vores program benyttet Model-View-Controller (MVC) design mønstret, og det er derfor meget nemt at identificere vores programs controllere. Vi har i IntelliJ, opdelt vores klasser i mapper *Controller* og *Model*. Vi har en mappe med alle vores Controllere i IntelliJ.

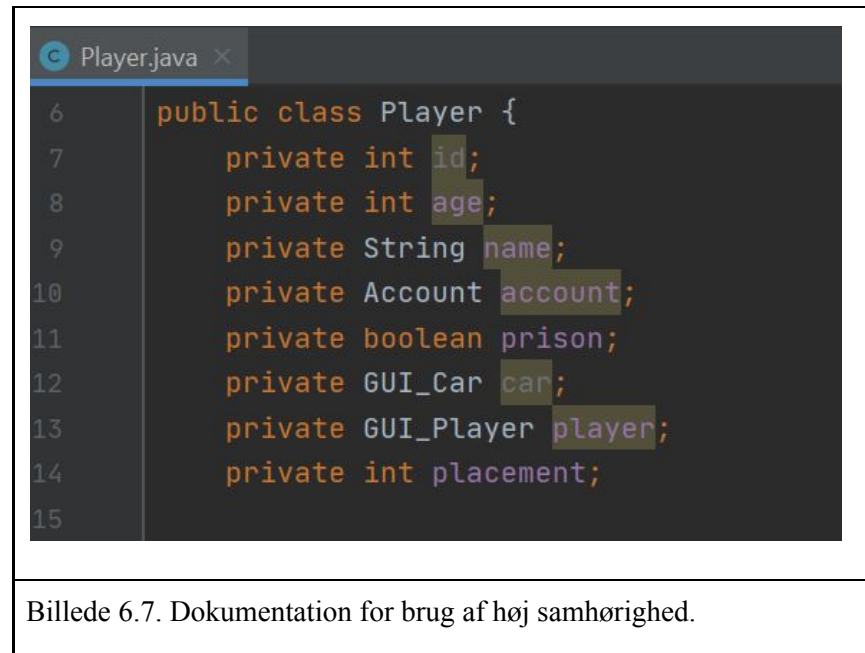
## Low coupling



Billede 6.6. Dokumentation for brug af lav kobling

Vi har prøvet på at have så lav kobling som muligt i vores program, dette har vi gjort ved at undgå unødvendige associationer mellem vores klasser. Det kan ses i vores klassediagram at vi har sørget for at det kun er meget få af vores klasser der har mere end to associationer. De enkelte klasser der har fået tildelt mange associationer, har været nødvendigt, da de er vigtige for programmets funktionalitet.

## High Cohesion



Vi har så vidt som muligt, prøvet på at overholde High Cohesion princippet, i vores projekt. Dette har vi gjort ved at have veldefinerede klasse, og så godt som muligt at adskille klassernes ansvarsområder. Vi har dog klassen *Account* som har fået tildelt en del mere ansvar end en konto klasse burde have. Dette vil altså sige at der ikke er høj samhørighed i *Account*-klasse. Et eksempel på at vi har brugt høj samhørighed, er vores *PropertySquare*, hvor klassen er veldefineret og har sit ansvarsområde som den overholder.





## 7. Test

### Code Coverage



Billede 7.1: Her ses hvilke af vores tests der er succesfulde, og hvilke tests der ikke er

The screenshot shows the IntelliJ IDEA code coverage report. It displays a table with columns for Element, Class, %, Method, %, and Line, %. The report shows coverage for various packages and classes, with the Controller package highlighted.

Element	Class, %	Method, %	Line, %
com			
Controller	50% (2/4)	24% (6/25)	26% (52/193)
gui_codebehind			
gui_fields			
gui_main			
gui_resources			
gui_tests			
images			
java			
javax			
jdk			
META-INF			
Model	76% (10/13)	44% (33/74)	49% (145/295)
netscape			
org			
Squaretype	100% (9/9)	20% (15/74)	38% (114/298)
sun			
toolbarButtonGraphics			
Main	0% (0/1)	0% (0/1)	0% (0/3)

77% classes, 39% lines covered in 'all classes in scope'

Billede 7.2: Dokumentation for vores code coverage ud fra vores 10 tests

Som det fremgår af billede 7.2, så har vi en samlet line coverage på 39% med vores tests. Da vi mener at vi har testet på de centrale dele af spillet, mener vi at dette medfører at vores program er godt gennemtestet, og derfor bør opfører sig som vi har tiltænkt det. Vi har også en class coverage på 77%, så vi kommer godt rundt i hele programmet ved hjælp af vores tests. Vi har en test der fejler, og vi har valgt at beholde den, da den er et eksempel på test driven development.

## Unit test

Testcase ID	TC 01 - testUpdate
Summary	Test der tjekker om en spillers konto kan miste og få penge.
Requirements	<i>updateBalance</i> metoden bliver eksekveret - F4
Preconditions	Der findes en integer <i>balanceUpdate</i>
Postconditions	Spillerens account balance er opdateret.
Test procedure	<ol style="list-style-type: none"><li>1. Opretter en ny spiller</li><li>2. Klargøre en integer</li><li>3. Tilføjer integer til <i>updateBalance</i>.</li><li>4. Tjekker for fejlagtig ændring ved at bruge metoden <i>getBalance</i> fra <i>Account</i></li></ol>
Test data	NaN
Expected result	Balance bliver 300
Actual result	300
Status	Succes
Tested by	Kasper Falch Skov
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 02 - propArrayTest
Summary	Test der tjekker om et <i>Account</i> kan modtage en ejendom med farvekode
Requirements	F4
Preconditions	Der er en ejendom og en spiller
Postconditions	Spilleren ejer ejendommen
Test procedure	<ol style="list-style-type: none"><li>1. Der bliver oprettet en spiller</li><li>2. Der bliver oprettet ejendommen "Lyse Stagen" med farven "black"</li><li>3. Spilleren får ejendommen</li></ol>
Test data	NaN
Expected result	black
Actual result	java.awt.Color[r=0,g=0,b=0]
Status	Failed
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Vi har her fået en mislykket test, da metoden *getProperties* forventer en rgb kode, og vores test spørger efter en String. Vi har altså her brugt test-driven development til at finde ud af at vores metode returnerer noget andet end vi forventer.

Testcase ID	TC 03 - PayCardFunctionTest
Summary	Test der tjekker om man kan kalde på superklassen for at bruge metoder fra underklassen
Requirements	F6
Preconditions	player1 trækker et chancekort igennem <i>ChancecardHandler</i> metoden <i>drawcard()</i> .
Postconditions	player2 har 3000 i sin balance og player1 har 2000 i sin balance
Test procedure	<ol style="list-style-type: none"><li>1. Opretter 2 spillere</li><li>2. Opretter boolean condition</li><li>3. Henter chancekort nr 10 fra <i>ChancecardHandler</i> og giver det til superklassen <i>ChanceCard</i></li><li>4. Giver chance kort nr 10 til player1 igennem <i>ChancecardHandler</i></li><li>5. Giver chance kort nr 10 til player2 igennem superklassen <i>ChanceCard</i></li><li>6. Tjekker om player2 har en pengebeholdning på 3000 og om player1 har en pengebeholdning på 2000</li></ol>
Test data	NaN
Expected result	true
Actual result	true
Status	Succes
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 04 - rentFromDiceEyes
Summary	Test der tjekker om <i>BrewerySquare</i> virker, som den skal og giver en rente på $100 \cdot \text{det antal øjne man slog i sit slag}$ .
Requirements	F31
Preconditions	Der er en spiller der har kastet en terning og er landet på en <i>BrewerySquare</i> som er ejet af en anden spiller
Postconditions	Spilleren der landede på <i>BrewerySquare</i> betaler $100 \cdot \text{LastRoll}$ , til ejeren af <i>BrewerySquare</i> .
Test procedure	<ol style="list-style-type: none"><li>1. GameBoard bliver oprettet</li><li>2. Der bliver oprettet to spillere</li><li>3. player bliver sat til at være ejeren af <i>BrewerySquare</i></li><li>4. player1 ruller 7 for at lande på <i>BrewerySquare</i></li><li>5. player1 får 700 trukket fra sin balance.</li></ol>
Test data	player1 point = 3000 player1 nye point = 2300
Expected result	2300
Actual result	2300
Status	Succes
Tested by	Henrik Lynggaard Skindhøj
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 05 - testRentIfAllPairsOwned
Summary	Test der tjekker om huslejen ændrer sig når du har alle <i>PropertySquare</i> af samme farve.
Requirements	F14
Preconditions	Alle <i>PropertySquare</i> der er cyan, er ejet af den samme spiller
Postconditions	Alle <i>PropertySquares</i> der er cyan, har dobbelt husleje.
Test procedure	<ol style="list-style-type: none"><li>1. Der oprettes et spillebræt</li><li>2. Der oprettes en spiller</li><li>3. Der oprettes 3 ejendomme med farven cyan</li><li>4. Spilleren får ejerskab, og de 3 ejendomme</li><li>5. Huslejen på de 3 ejendomme bliver fordoblet</li></ol>
Test data	Valby Langgade rent = 100 Valby Langgade ny rent = 200
Expected result	200
Actual result	200
Status	Success
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 06 - priceCardFunctionTest
Summary	Test der tjekker om priceCard klassens funktion virker. Der er kun 2 kort af denne type, og derfor nødvendig at teste.
Requirements	F6
Preconditions	player trækker et chancekort igennem <i>ChancecardHandler</i> metoden <i>drawcard()</i> .
Postconditions	At penge er trukket fra gælden spiller.
Test procedure	<ol style="list-style-type: none"><li>1. Spiller bliver oprettet</li><li>2. boolean sat false</li><li>3. Spiller objekt får huse under sin account</li><li>4. chance kort bliver trukket og funktion eksekveret</li><li>5. tjekker om penge er trukket (3500 kr)</li></ol>
Test data	Spiller start pengebeholdning: 3501 kr Spiller har 3 huse Spiller har et hotel
Expected result	1 kr
Actual result	1 kr
Status	Success
Tested by	Filip Igor Meyer Clausen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 07 - testRentIfNotAllPairsOwned
Summary	Er i virkeligheden det omvendte af TC05. Hvis ikke alle ejendomme er ejet, hæver den så renten.
Requirements	F14
Preconditions	2 ejendomme af samme farve er ejet.
Postconditions	prisen ikke ændret siden farven er hverken blå eller magenta
Test procedure	<ol style="list-style-type: none"><li>1. Spiller får to ejendomme af samme farve</li><li>2. Ejendom med given farve bliver oprettet.</li><li>3. Ejet ejendoms farver bliver sammenlignet</li><li>4. Tjek om renten er blevet ændret</li></ol>
Test data	Renten er 100 kr Spiller med to ejendomme
Expected result	100
Actual result	100
Status	Success
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit



Testcase ID	TC 08 - testRentIfTwoColorPairOwned
Summary	Farven blå og magenta er unik siden renten skal fordobles ved kun 2 ejet ejendomme. (Ikke 3 som normalt)
Requirements	F14
Preconditions	2 ejendomme af samme farve er ejet.
Postconditions	prisen er ændret da farven er enten blå eller magenta
Test procedure	<ul style="list-style-type: none"><li>5. Spiller får to ejendomme af samme farve</li><li>6. Ejendom med given farve bliver oprettet.</li><li>7. Ejet ejendoms farver bliver sammenlignet</li><li>8. Tjek om renten er blevet ændret</li></ul>
Test data	Renten er 50 kr Spiller med to ejendomme
Expected result	100
Actual result	100
Status	Success
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 09 - multipleShippingSquaresRent
Summary	Tester om man betaler det rigtige alt efter hvor mange shipping squares ejeren ejer.
Requirements	F31
Preconditions	Ejeren af rederiet der bliver landet på ejer 3 rederier
Postconditions	prisen har ændret sig fra 500 til 2000
Test procedure	<ol style="list-style-type: none"><li>1. En spiller får tildelt 3 rederier</li><li>2. Et felt objekt bliver oprettet</li><li>3. Spilleren bliver sat som ejer af rederi objektet</li><li>4. Spillerens pengebeholdning bliver ændret</li><li>5. Tjekker ændringerne af spillerens pengebeholdning</li></ol>
Test data	spilleren ejer 3 rederier prisen for at lande på et af rederierne er 2000 spilleren der lander på en af rederierne har 3000 kroner
Expected result	1000
Actual result	1000
Status	Success
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

Testcase ID	TC 10 - determineWinnerTest
Summary	Tester om vinderen bliver defineret
Requirements	F18
Preconditions	Alle andre spillere er gået fallit
Postconditions	Der er en spiller tilbage, hvis "playerLost" boolean er false
Test procedure	<ol style="list-style-type: none"><li>1. 2 spiller objekter bliver oprettet</li><li>2. Spiller objekter placeres i array</li><li>3. Regelsæt objektet bliver oprettet</li><li>4. En spillers playerLost boolean bliver ændret til true</li><li>5. Tjek om vinderens navn er fundet og returneret.</li></ol>
Test data	player name = HC Player1 name = jan removeplayer metode fra playerhandler bliver kørt på player
Expected result	Jan
Actual result	Jan
Status	Success
Tested by	Hans Christian Leth-Nissen
Date	17-01-2021
Test environment	Intellij IDEA 2020.2.3 (Ultimate Edition) Build #UI-202.7660.26 built on October 6, 2020 JUnit 4.13.1 Windows 10 Home 64 bit

## 8. Projektplanlægning

### Gantt-kort

Projektets fremgang og planlægning er dokumenteret på et gantt-kort (se bilag 1). Gantt-kortet viser opgaver, og tidsestimeringer til opgaverne. Tiderne kan også anses som deadlines. Opgaverne er opdelt efter Rational Unified Process (RUP) discipliner. Artefakterne der bliver udarbejdet i iterationerne er beskrevet nederst. I bunden af gantt-kortet ser man faserne. Ansvar kan også gives til gruppemedlemmer.

Gennem perioden kommenterer gruppen på deres egen fremgang. Kommentaren kan beskrive, hvad vi har nået og hvad der kunne forbedres. Så efter hver iteration beskriver gruppen deres arbejde og sætter planer for næste iteration. Siden projektet kun varede 2 uger, så har hver fase kun en iteration. I længere projekter vil der være flere iterationer, og højst sandsynligt flere artefakter.

Vi har benyttet gantt-kortet som en tidsplan, for at holde styr på hvor langt vi var nået med de forskellige dele af projektet. Vi har igennem meget af projektet være hurtigere færdige med nogle af vores artefakter end forventet, og vi har derfor været foran vores tidsplan på nogle punkter.

### Feature Branching

Vi har i vores projekt taget brug af Git, til at versionsstyre vores kode, dette har gjort det muligt at holde styr på vores kodes udvikling. Git har også gjort det muligt at dele kode sammen i gruppen, og det er blevet gjort på en overskuelig måde ved at bruge Feature Branching strategien. Feature Branching går ud på at oprette en ny feature branch for hver funktion i programmet som vi har udarbejdet. Når en funktion er kodet færdig, har vi meget den omtalte feature branch ind i vores Development branch.

Feature Branching strategien kræver at man committer sin kode ofte, så der opstår så få problemer som muligt, og så man kan identificere og løse problemer så snart de opstår. Dette har vi overholdt i vores projekt ved at tage udgangspunkt i kundens krav, der siger at vi helst skulle committe en gang i timen. Da vi har committet så ofte som vi har gjort, er det muligt

for os at holde står på udviklingsprocessen, og det giver også kunden mulighed for at se processen.

Vores branching strategi gik dog galt da vi skulle til at kode salg af huse og ejendomme på *PropertySquare*. Vi lavede et par commits i *BugFixes* branchen og kom så i en situation hvor vi ikke kunne få det til at virke og lavede en masse flere commits i *BugFixes* branchen. Vi har altså lavet rigtig meget kode i en branch der ikke var en feature branch.

## Konfigurationsstyring

I projektet har vi taget brug af forskellige værktøjer og programmer til at udarbejde et gennearbejdet projekt. Vi har benyttet Google Drev til at skrive vores rapport, da Google drev giver os mulighed for at skrive i det samme dokument samtidigt, og følge med i hvad andre gruppemedlemmer skriver. Brug af Google Drev har også gjort det muligt for os at versionsstyre vores rapport.

I vores projekt har vi brugt programmet MagicDraw til at lave mange af vores artefakter, der er med i denne rapport. Vi har brugt MagicDraw, da det er nemt at lave pæne diagrammer, uden meget besvær. Vi har brugt version 18.5 af MagicDraw

Til at skrive vores kode har vi brugt programmet IntelliJ Idea. Vi har brugt dette program, fordi alle i gruppen har kendskab til det, og vores undervisning har foregået i IntelliJ IDEA. Vi har skrevet koden i versionen - IntelliJ IDEA 2020.2.3 (Ultimate Edition). Vi har skrevet koden med JDK 14.01

Til versionsstyring af vores kode har vi taget brug af Git. Git virker godt sammen med IntelliJ, da det er meget nemt at oprette nye branches, skifte imellem branches og merge dem sammen. Vi har brugt Maven til at hente biblioteket matadorgui.jar. Vi har også brugt Maven til at hente JUnit, som vi har brugt til at teste. Vi har brugt version 4.13.1 af JUnit.

Den nyeste version af vores projekt, er den version der senest er blevet uploadet. Denne version er vores færdige projekt, hvor alle artefakter fra rapporten stemmer overens med vores kode.

## Konklusion

I denne opgave har vi programmeret spillet, Dansk Monopoly. Vi har lavet et gantt kort, for at holde styr på spillet udvikling, og vi har også haft 2 møder med vores kunde, for at give kunden indblik i hvordan vores spil ville komme til at se ud, og hvilket dele af programmet vi ikke ville kunne nå at lave. Vi har udarbejdet spillet ud fra et sæt regler som vi har fået givet af en kunde. Vi er så kommet frem til et hvor man kan spille imellem 3-6 personer, der rykker rundt på vores spillebræt, ved skiftevis at slå med 2 terninger. På vores spillebræt har vi lavet 40 felter med hver deres funktion som har en effekt på spillernes pengebeholdning. Vi er kommet frem til dette ved at lave regelsættet om til en række krav, og herfra har vi lavet use case, og disse har vi brugt til at lave vores klasser.

Ud fra vores klasser har vi så lavet artefakter, som vi har brugt til at gøre kodning processen så overskuelig som muligt. Vi har også genbrugt dele fra vores forrige projekt, CDIO 3, hvor vi skulle lave et Monopoly Junior spil. Vi har så benyttet en GUI, som vi har fået udleveret, så vores program er blevet visuelt gennemført. For at sikre os at vores spil er et gennemført produkt, har vi så lavet nogle unit test der tester forskellige funktioner i vores spil.

Vi har altså fået lavet et gennemført Dansk Monopoly spil, som følger de krav vi fik givet i regelsættet.

## Bibliografi

### Programmeringsbogen

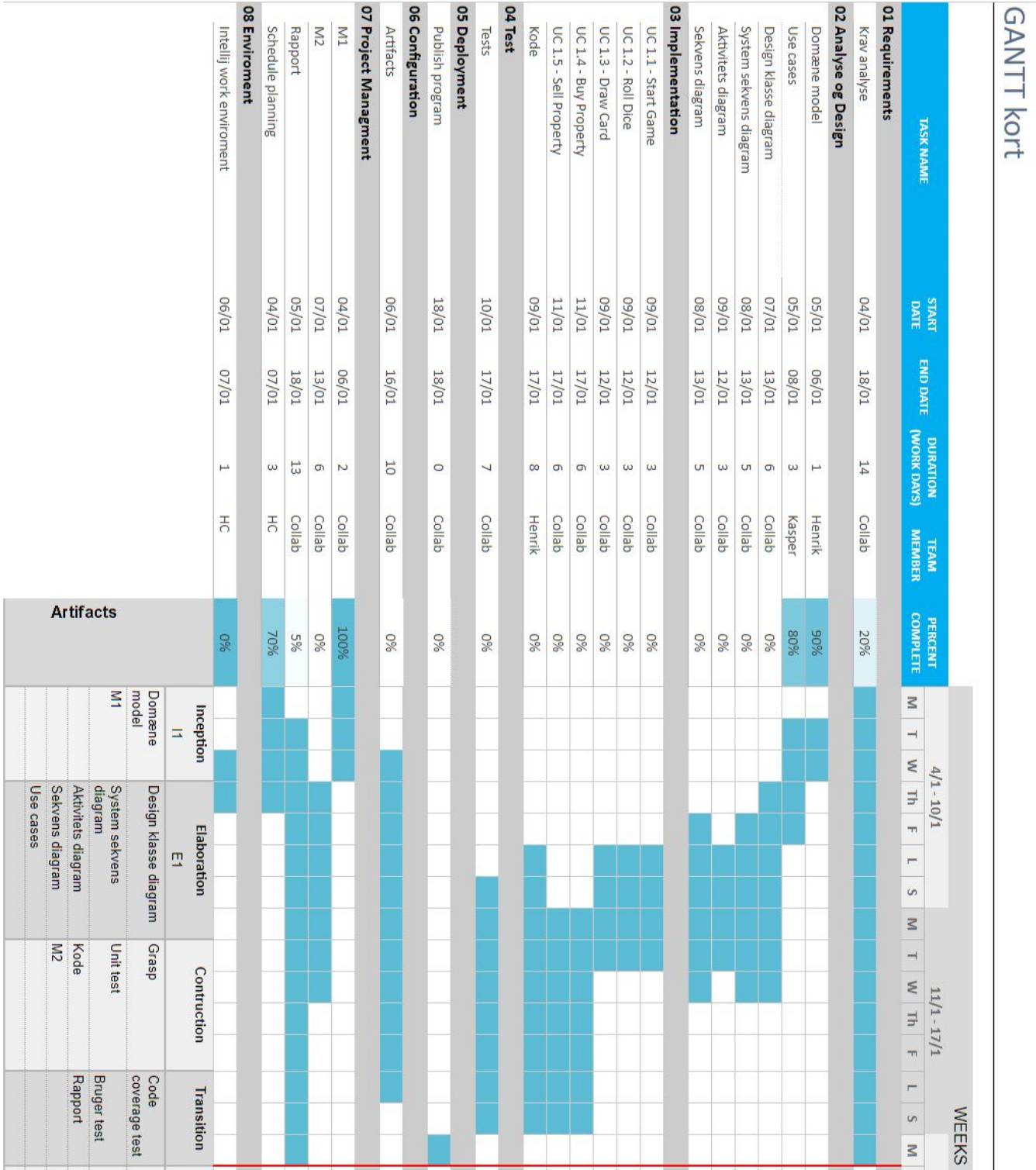
Lewis, J.. & Loftus, W.. (2017). *Java Software Solutions: Foundations of Program Design* (9.. udg.). Pearson Education Limited.

### UML bogen

Larman, C.. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3.. udg.). Pearson Education.

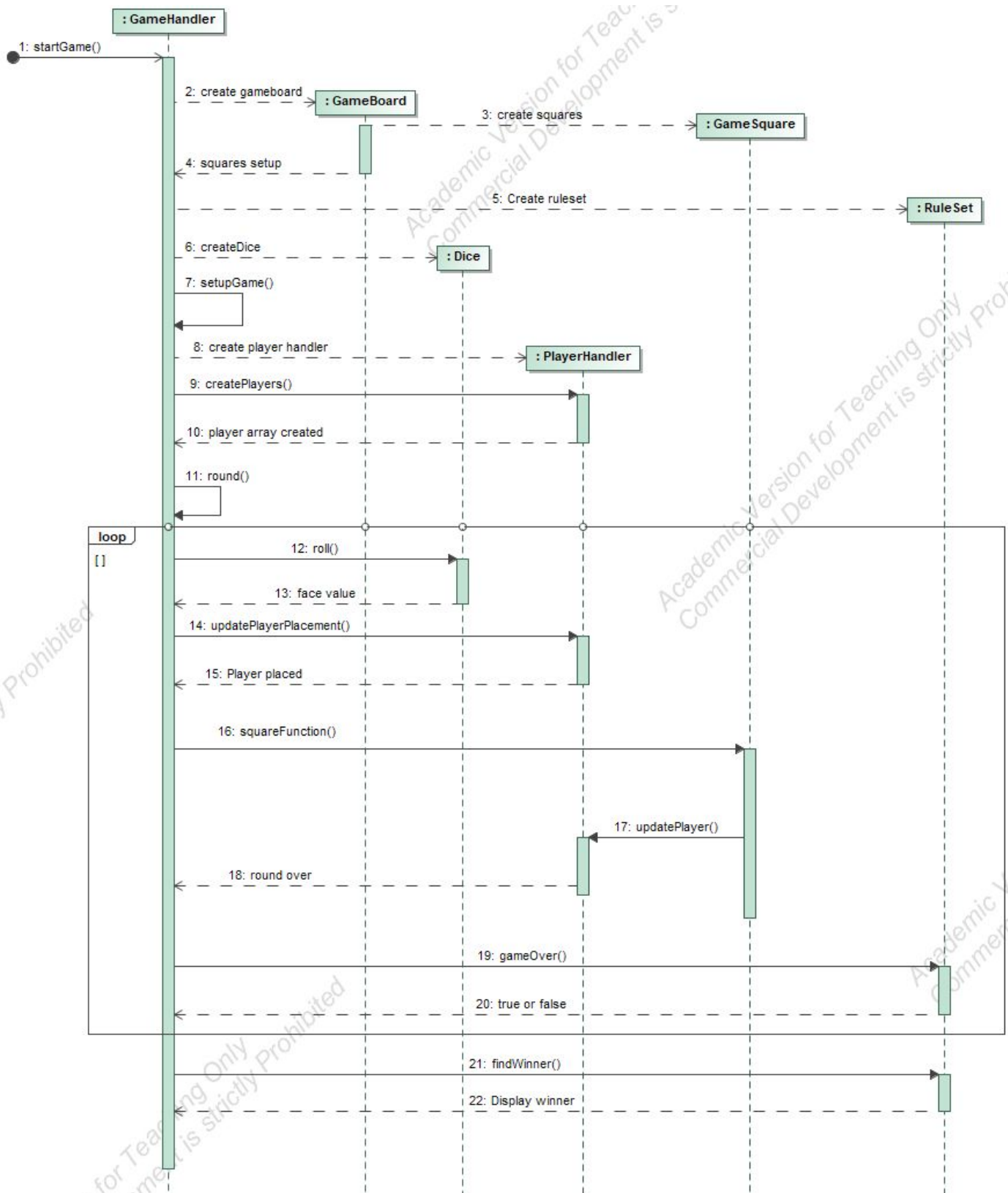
Vi har i vores projekt genbrugt nogle dele af rapporten og koden fra CDIO 2 og CDIO 3.

Bilag 1: Gantt Kort

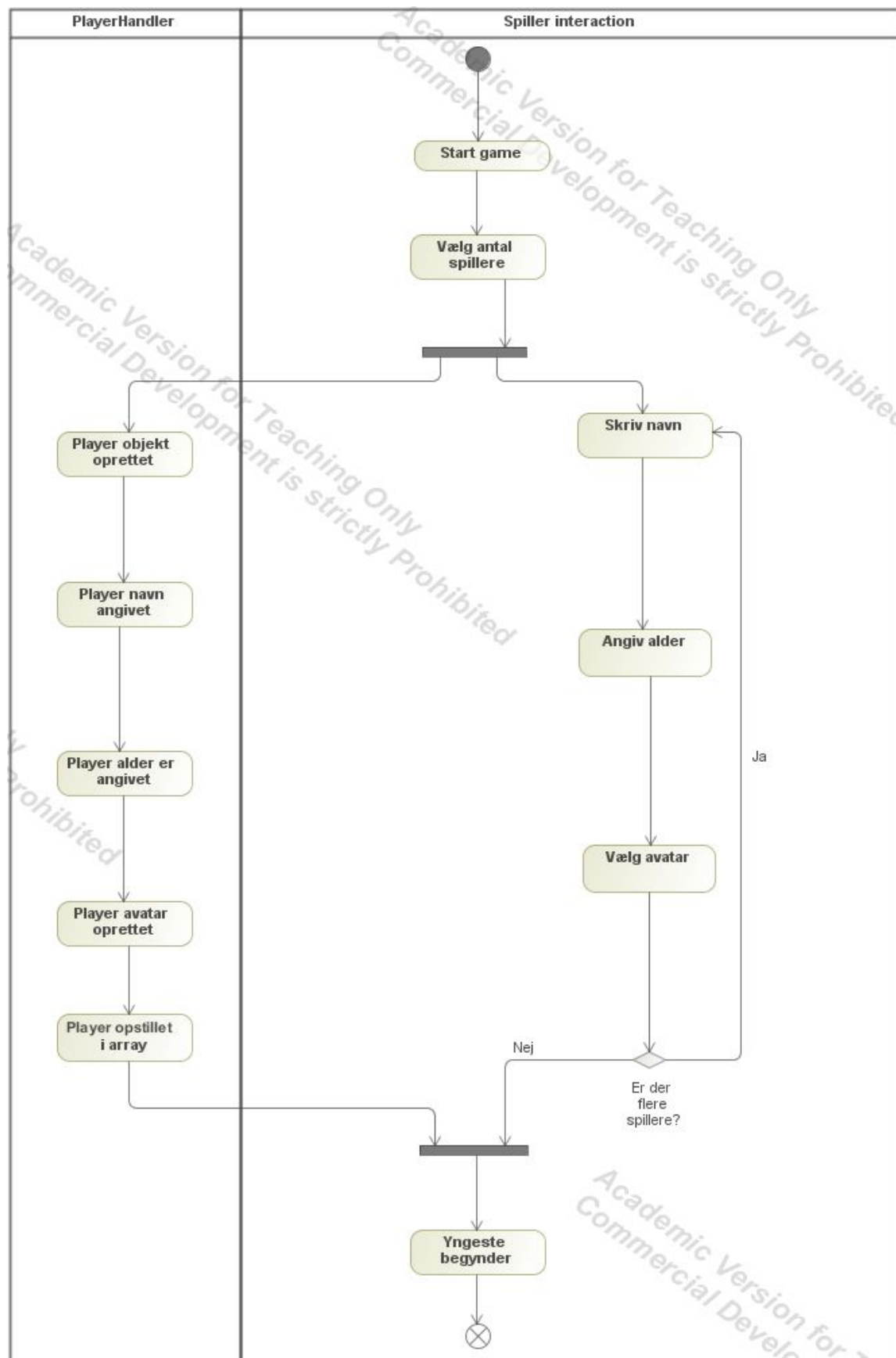




## Bilag 2: Sekvensdiagram



## Bilag 3: Aktivitetsdiagram



## Bilag 4: Design klassediagram

For en større version, der faktisk af læselig, kan dette link bruges:

<https://prnt.sc/x37evr>

