CDIO 3. Kasper Leiszner, Bijan Negari, Frederik Von Scholten, Troels Lund, Helene Zgaya
Fag: Indledende programmering (02312), Udviklingsmetoder til IT-systemer (02313) og Versionsstyring og testmetoder (02315)

Projektnavn: CDIO 3
Gruppenummer: *18*
Afleveringsfrist: *Fredag den 25/11 2016 Kl. 23:59*
Fag og retning: Diplom Softwareteknologi
Denne rapport er afleveret via Campusnet (der skrives ikke under).
Denne rapport indeholder 26 sider inklusiv denne.

Kasper Leiszner s165218

Frederik Von Scholten s145005

Bijan Negari s144261

Troels Lund s161791

Helene Zgaya s104745

Resumé

Vores projekt går ud på at udvikle et spil for to til seks spillere. Spillet går ud på at spillerne slår med et par terninger på skift og rykker deres personlige brik rundt på en spilleplade bestående af 21 felter af fem forskellige slags felter: *Fleet*, *Laborcamp*, *Refuge*, *Tax* og *Territory* hver med deres egne unikke egenskaber.

Felterne *Fleet*, *Laborcamp* og *Territory* kan ejes (*Ownable*) dvs. at en spiller kan vælge at købe feltet hvis/når spilleren lander på et felt af disse typer og det ikke er ejet af en anden spiller. Køber en spiller et felt skal andre spillere der lander på feltet betale en afgift (*rent*).
Felterne *Tax* og *Refuge* kan ikke ejes, men der trækkes en afgift fra spilleren i form af enten et fast beløb eller en procentdel af spillerens pengebeholdning afhængigt af felt og spiller valg på *Tax*-feltet og modtager en bonus præmie på *Refuge*-feltet.
Hver spiller starter med 30.000 kr. Spillet fortsættes indtil der kun er en enkelt spiller tilbage, som så er vinderen. Der er brugt flere UML diagrammer til både analyse- og design delen af programmet. Desuden er programmet blevet testet.
Vi har formået at opfylde kravene og vores program er testet. Alt i alt har vi udviklet et velfungerende program.

Timeregnskab

Projekt Navn	CDIO del 3						
Time-regnskab	Ver.						

Dato*	Deltager	Design & Analyse	Impl.	Test	Dok.	Andet	Ialt
8-11-2016	Kasper	1				0,5	1,5
8-11-2016	Frederik	1				1	2
8-11-2016	Troels	1				1	2
8-11-2016	Bijan	1				1	2
9-11-2016	Kasper	1					1
9/11/16	Frederik	1,5					1,5
12/11/16	Kasper					1	1
16/11/16	Kasper		4				4
19/11/2016	Helene	0,5	2				2,5
16/11/2016	Helene		1				1
13/11/2016	Helene	1					1
19/11/2016	Frederik		2	0,5			2,5
19/11	Troels		2,5				2,5
20/11	Troels		3	1			4
20/11	Frederik		2	0,5		1	3,5
20/11	Helene		1,5	1			2,5
20/11	Bijan		4				4
20/11	Kasper						0
21/11	Kasper		1,5				1,5
21/11	Troels		2				2
21/11	Frederik	1,5	0,5		0,5		2,5
21/11	Bijan		1				1
22/11	Troels		1		0,5		1,5
22/11	Frederik	1				1	2
22/11/2016	Bijan		2			0,5	2,5
23/11/2016	Helene	1	1			1	3
23/11/2016	Frederik	2			2	1,5	5,5
23/11/2016	Kasper		3	3		0,5	6,5

23/11/2016	Bijan		3	3		0,5	6,5
23/11/2016	Troels		1	2		2	5
24/11/2016	Kasper	1	2	2		1	6
24/11/2016	Helene	1	1,5	1	1		4,5
24/11/2016	Bijan		1	1,5		3,5	6
24/11/2016	Frederik	1,5			1	2,5	5
24/11/2016	Troels		2	2	1	0,5	5,5
25/11/2016	Frederik					2	2
25/11/2016	Helene				1	1	2
25/11/2016	Troels	1			1	1	3
25/11/2016	Kasper	1					1
25/11/2016	Bijan						0
	SUM	19	44,5	17,5	8	24	113

Timer i alt pr. gruppemedlem

	Deltager	Design & Analyse	Impl.	Test	Dok.	Andet	Ialt
	Bijan	1	11	4,5		5,5	22
	Frederik	8,5	4,5	1	3,5	9	26,5
	Helene	3,5	7	2	2	2	16,5
	Kasper	4	10,5	5		4,5	24
	Troels	2	6,5	5	2,5	6,5	22,5
	SUM	17	33	12,5	5,5	21	111,5

Indhold

Indledning

Denne rapport er udarbejdet som led i undervisningen i fagene *Indledende programmering (02312), Udviklingsmetoder til IT-systemer (02313) og Versionsstyring og testmetoder (02315)* på første semester på diplom software.
Rapporten dokumenterer planlægning og udvikling af et program der skal simulere et brætspil mellem 2-6 spillere. Vi har skrevet i det objektorienteret programmeringssprog Java.
Først og fremmest har vi udfærdiget en kravspecifikation som er udført mellem os udviklere, projektleder og kunde, og som har til formål at sikre et færdigt produkt som begge parter er enige om opfylder kundens ønsker. Denne beskrives mere udførligt i hovedafsnittet. Derudover har vi benyttet flere modeller til at visualisere analyse samt dokumentation for udvikling af vores program.

Det primære mål er altså at udvikle nedenstående spil ved hjælp af objektorienteret analyse og design samt teste det.
Sekundært ønsker vi at implementere en GUI der er blevet udviklet på forhånd.

Brætspillet foregår altså mellem 2-6 spillere der skiftes til at slå med to terninger. Værdien af disse placerer spilleren på et af 21 felter der har individuel effekt for spilleren (se feltliste i bilag). Spillerens pengebeholdning starter på 30.000. Går en spiller bankerot bliver spillerens eventuelle felter ledige igen for salg såfremt der er flere end 2 spillere. Når alle spillere er bankerot på nær en, slutter spillet.

Opsummering

Følgende er altså beskrevet i denne rapports hovedafsnit:
Krav og analyse (dertilhørende modeller)
Design-dokumentation (dertilhørende modeller)
Beskrivelse af koden
Test samt dertilhørende beskrivelser
GRASP
Versionsstyring
Konfigurationsstyring
Sidst vil der forelægge en diskussion samt en konklusion som grunder i hvorvidt vi har formået at fuldføre alt beskrevet ovenfor.

Indledende overvejelser af udviklingsproces
I opstartsfasen af projektet har vi gjort os nogle overvejelser om den mest hensigtsmæssige fremgangsmåde og strukturering af projektet udviklingsproces. Vi taget vores erfaringer med fra løsningen af CDIO del 2 med i vores overvejelser.

Vi har tilgået softwareudviklingen ved hjælp af Agile software development. Der tager udgangspunkt i Unified Process med tidsbegrænset iterationer fordelt i tre faser. I henholdsvis etablering(elaboration), konstruktion(construction) og afslutningsvis i overdragelsen(transition) - forberedelses(inception) fasen er undladt.

Vi har benyttet os af https://trello.com til at organisere og opgavestyring. Trello gør det muligt for gruppens medlemmer se status, hvem der har hvilke opgaver og tidsfrister. Det gør det også nemmere for projektlederen at organisere arbejdet løbende.

Hovedafsnit

Analyse

Til udvikling af vores program har vi først og fremmest udfærdiget en kravspecifikation ud fra kundens vision. Dernæst har vi udarbejdet flere modeller med udgangspunkt i UML og ved hjælp af objektorienteret analyse.
Følgende modeller er udviklet til visualisering af vores overvejelser for hvordan programmet skal se ud overordnet, for derefter at kunne udvikle modeller der beskriver udviklingen mere i detaljer:

- Kravspecificering
- Navneords- og udsagsnordsanalyse
- Use-case diagram
- Use-case beskrivelser
- Traceability Matrix
- Domænemodel
- BCE
- Analyse klassediagram* gammelt

Kravspecifikation Analyse

Vi har udarbejdet en kravspecifikation ud fra kundens vision (CDIO del 3) samt kravspecifikationen fra det foregående projekt (CDIO del 2), hvilket har til formål at danne nogle specifikke og klare retningslinjer. Det gøres for at skabe en fuldstændig forståelse mellem udvikler og kunde, så det færdige produkt opfylder kundens ønske og så vi undgår misforståelser og i øvrigt får klargjort over for kunden hvad der er muligt at levere, da kunden oftest ikke har den nødvendige viden som udviklerne har. Derfor fungerer kravspecifikationen samtidig som en kontrakt mellem udvikler og kunde.

FURPS+
Vi benytter os af FURPS+ metoden vha. analyse med udgangspunkt i kundens vision fra hhv. CDIO del 3 og del 2 for at udspecificere kundens krav i en kravspecifikation. FURPS+ består af følgende krav: *functionality, usability, relaiblility, performance, supportability*. Kravspecifikation ses nedenfor.

Kravspecifikation

Functionality

- R1: Spillet skal foregå mellem to til seks spillere.
- R2: Spillerne skal skiftes med at have tur.
- R3: Spillerne skal slå med to, sekssidet terninger.
- R4: Spillerne skal kunne lande på et felt og så fortsætte derfra på næste slag.
- R5: Spillerne skal rykke det antal felter som summen af de to terninger angiver.
- R6: Spillet skal have felter nummeret 1-21 af 5 forskellige typer.
- R6: Hvert felt skal have egenskaber som defineret i feltlisten. (Se bilag 1.2)

- R8: Hvis spillerne lander på et felt der kan ejes, som ikke er ejet i forvejen, skal spilleren have mulig for at købe det.
- R9: Spilleren skal kunne lande på et felt og fortsætte fra det felt.
- R10: Hver spiller skal starte med en pengebeholdning på 30.000 Dkk.
- R11: Spillet skal slutter når der kun er en spiller som ikke er bankerot.
- R12: Der skal være en grafisk brugerflade.

Usability

- R13: Der skal findes en brugervejledning til hentning, og afvikling af spillet.

Reliability

Ingen

Performance

- R14: Spillet skal kunne køre uden forsinkelser på over 1 sek.
- R15: Spillet skal kunne køres på DTU's maskiner i data-barer.

Supportability

- R16: Spillet skal være nemt at oversætte til andre sprog.
- R17: Spiller og spillerens pengebeholdning skal uden videre kunne genbruges i andre spil.
- R18: Det skal være nemt at skifte til andre terninger.

Navneords- og Udsagnsordsanalyse

Analysen tager udgangspunkt i kundens vision, hvor vi har markeret henholdsvis navneord og udsagnsord (se bilag 1.1 og 1.2). Ordlisten fra CDIO2 kan findes i linket til rapporten i litteraturlisten.

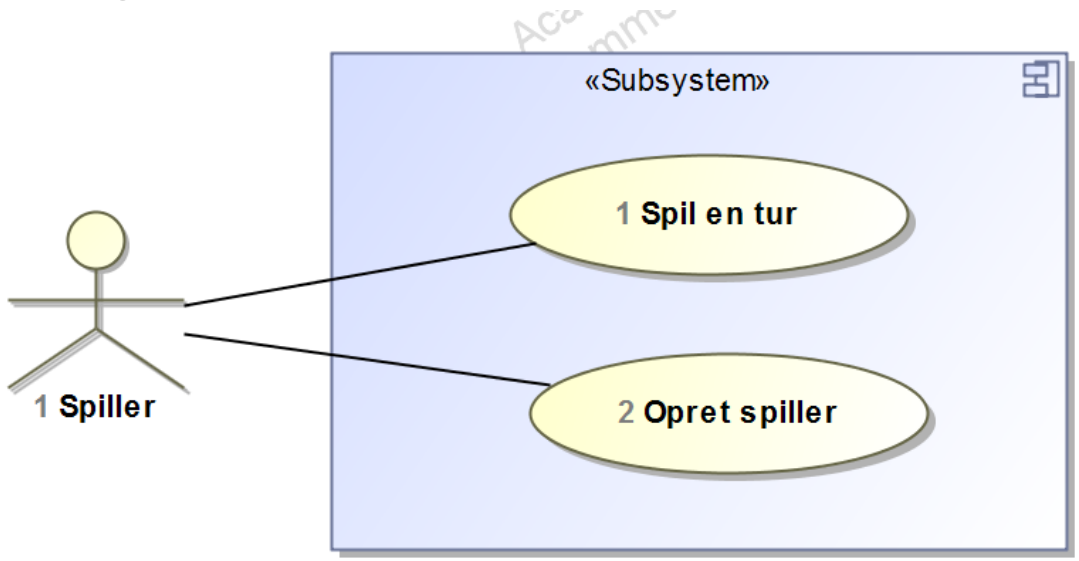
Navneord:

Spiller	Ejeren	Feltliste	Territory	Labor camp	Tax
Refuge	Afgift	Terning	Spil	Felter	Spilleplade
Bræt	Ring	Fleet			

Udsagnsord:

Arbejder	Ønsker	Udbygge	Lunde	Går	Være
Slutter					

Use-case diagram



Figur 1 Giver et simpelt visuelt overblik over hvordan spilleren interagerer med systemet og samler alle use cases i et diagram..

Use-case Beskrivelser

Use casen “Spil en tur” er generaliseret for en spiller, men gentages for hver oprettet spiller indtil spillet er slut.

Use case: Spil en tur
ID: UC1
Beskrivelse: Kort beskrivelse af use casen En tur spilles. Spiller kaster med terningerne. Terningens værdi svarer til det antal felter spilleren skal rykke fra spillerens pågældende plads på spillepladen. Feltet afgøre hvorvidt spilleren skal betale til ejeren af feltet eller har mulighed for at købe det pågældende felt. Hvis spillers balance går i nul smides man ud af spillet.
Primær aktør: En spiller
Sekundær aktør: Ingen
Precondition: 1. Spil påbegyndt 2. Spillere oprettet 3. En spillers tur
Main flow: 1. Kast terningerne, spilleren slår 7 2. Spiller rykker 7 felter frem fra sit pågældende felt. Spilleren står på felt 1 og rykker hermed til felt 8. 3. Felt 8 er af typen Territory, spilleren vælger at købe feltet. 4. Felt 8 har en pris på 5000, de 5000 bliver trukket fra den pågældendes spillers bankkonto 5. Spilleren ejer nu feltet og det kan ikke købes af andre. Hvis en anden spiller lander på felt 8 skal spilleren betale afgiften på 2000.
Alternativ flow: 1. Kast terningerne, spilleren slår 5 2. Spiller rykker 5 felter frem fra sit pågældende felt. Spilleren står på felt 7 og rykker hermed til felt 13. 3. Felt 13 er af typen Refuge 4. Dermed modtager spilleren 500. 5. De 500 bliver lagt oven i spillerens nuværende saldo.
Alternativ flow: 1. Kast terningerne, spilleren slår 2 2. Spiller rykker 2 felter frem fra sit pågældende felt. Spilleren står på felt 13 og rykker hermed til felt 15. 3. Felt 15 er af typen Laborcamp 4. Labor camp er ejet af en anden spiller 5. Du kaster terningerne for at regne ud hvad du skal give i afgift til ejeren. 6. Du slår to 5’ere, summen er 10. Summen bliver ganget med 100 og ganget med det antal laborcamps ejeren har i forvejen. Ejeren har 2 laborcamps derfor skal spilleren betale ejeren 2000 7. Spilleren bliver fratrukket 2000 fra hans saldo

Alternativ flow: 1. Kast terningerne, spilleren slår 2 2. Spiller rykker 2 felter frem fra sit pågældende felt. Spilleren står på felt 15 og rykker hermed til felt 17. 3. Felt 17 er af typen tax 4. Du skal nu vælge at betale en fast afgift på 4000 eller 10% af din saldo 5. Spilleren vælger at betale 10% med en saldo på 10000 6. Spilleren betaler 1000 7. Spillerens nye saldo bliver derfor 9000
Alternativ flow: 1. Kaster terningerne, spilleren slår 2 2. Spiller rykker 2 felter frem fra sit pågældende felt. Spilleren står på felt 17 og rykker hermed til felt 19. 3. Felt 19 er af typen fleet 4. Feltet er ejet i forvejen 5. Ejeren ejer 3 fleet felter 6. Spilleren skal dermed betale 2000 til ejeren 7. Spilleren havde en saldo på 10000, derfor er hans nye saldo på 8000
Postcondition: 1. Gå videre til næste tur

Use case: Opret Spillere
ID: UC2
Beskrivelse: Opretter 2-6 spillere og sæt balance
Primær aktør: En spillere
Sekundær aktør: Ingen
Precondition: 1. Spil påbegyndt
Main flow: 1. Spørg hvor mange spillere der skal oprettes 2. Opret antallet af spillere 3. Spillers tilhørende balance sættes til 30.000
Secondary flow: Ingen
Postcondition: 1. Alle spillerne er oprettet 2. Spillet fortsætter 3. Alle spillere har 30.000 i balance
Alternativ flows: None

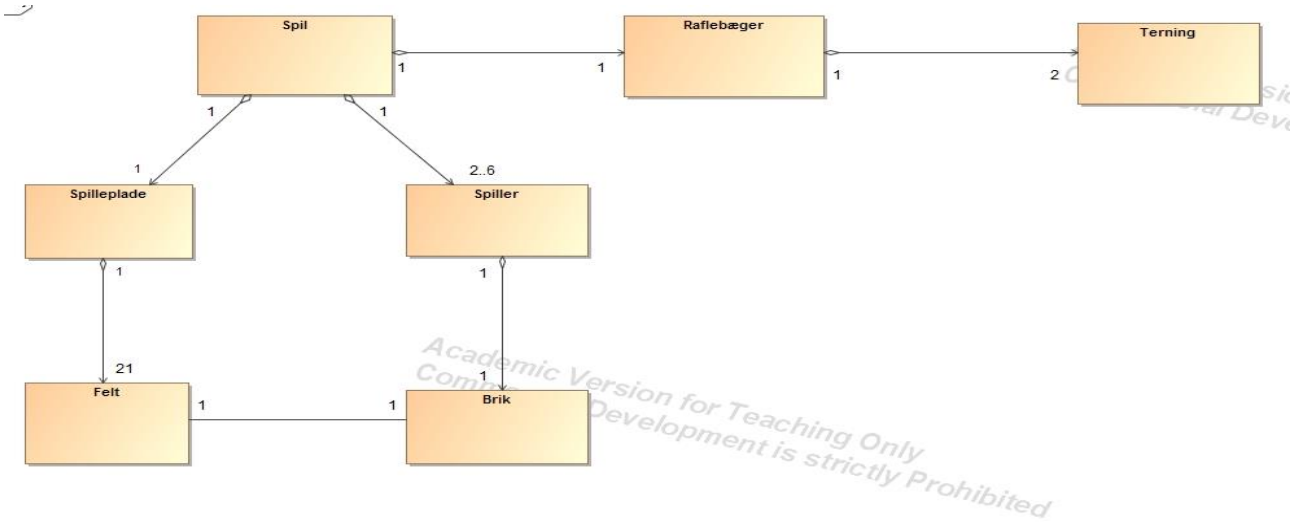
Beskrivelse af use-cases

Traceability Matrix

	R1	R2	R3	R4	R5	R6	R7	R8
UC1		x	x	x		x	x	
UC2	x				x			

Use-casen ”Spil en tur” er speciel på den måde at hvert flow generaliserer de manglende flows, f.eks. når det ses i main flowet at feltet ikke bliver købt, så er det generelt for alle de andre typer felter, at bliver det tilfældige felt ikke købt, så sker det samme som i main flowet. På samme måde tager de alternative flows også udgangspunkt i hinanden. Dette er mest relevant under de ”Ownable” felter, altså de felter der kan ejes.

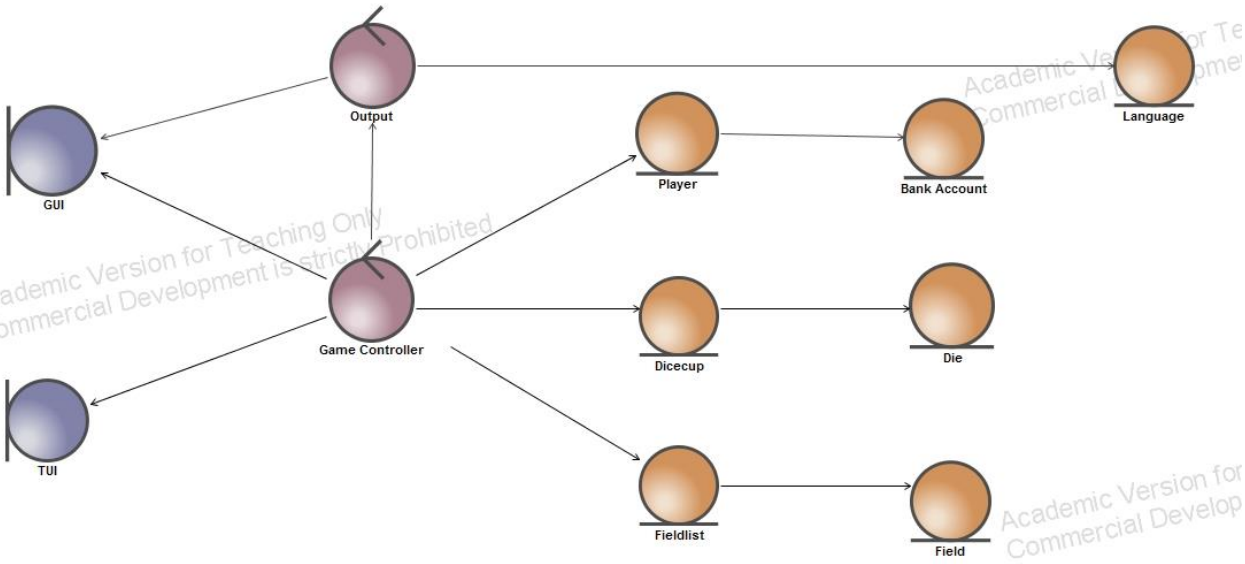
Domænemodel



Figur 2 Domænemodellen er en konceptuel model der viser hvordan de forskellige dele hænger sammen samt multipliciteter. Selvom "Spil" ikke som sådan er med, giver det mening for at se en sammenhæng.

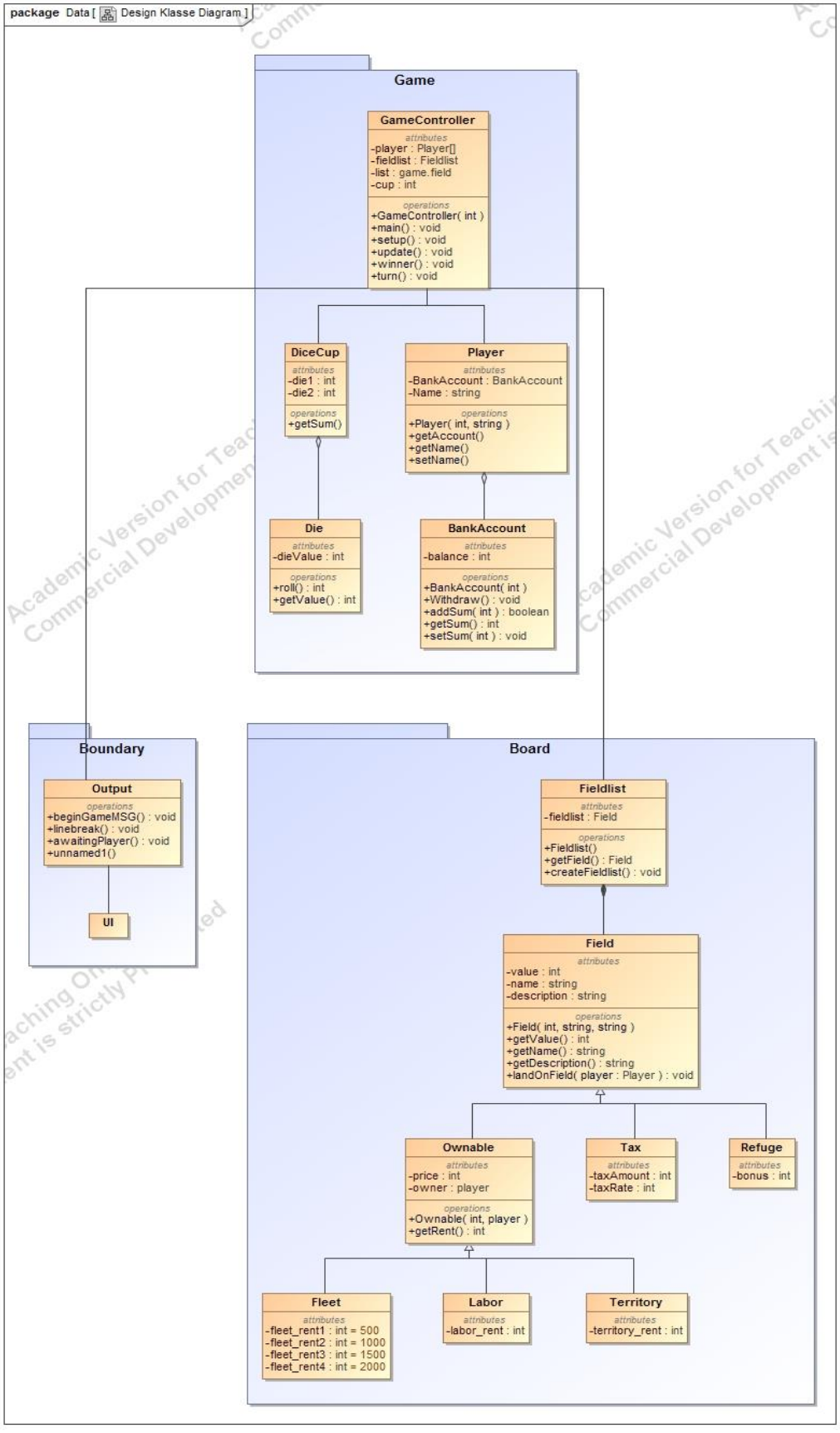
BCE model

Figur 3 Viser sammenspil mellem boundaries, controllers og entities.



Figur 4 Viser sammenspil mellem boundaries, controllers og entities. Vi har normalt to use cases, spil en tur og opret spiller, men da opret spiller er så lille en use-case, gav det bedre mening at plotte begge ind i en klasse kaldet GameController. Da output klassen indeholder har forbindelse til language klassen og kommunikerer med GUI'en har vi valgt at give den rollen som controller.

Analyse klassediagram



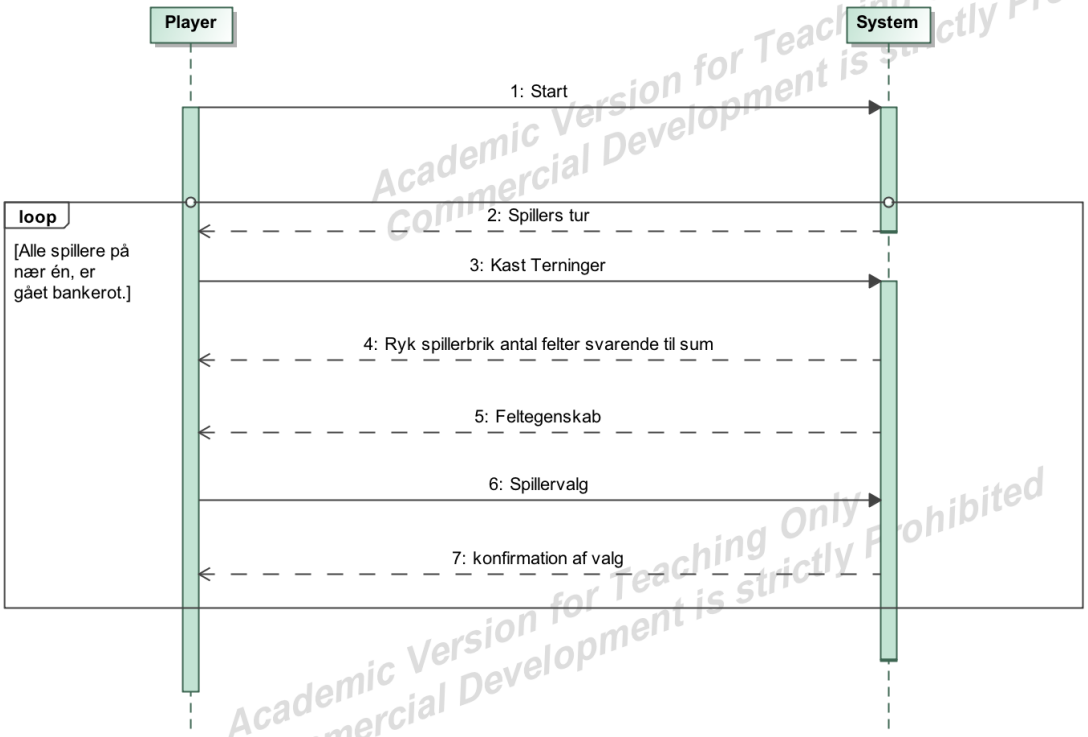
Figur 5 Analyse diagrammet viser hvordan vi har tænkt vores kode skal se ud.

Design

Følgende afsnit omhandler designet af vores program hvor vi har udarbejder diagrammer med UML notation ud fra objektorienteret analyse og design metoderne. Vi har her udarbejdet følgende diagrammer.

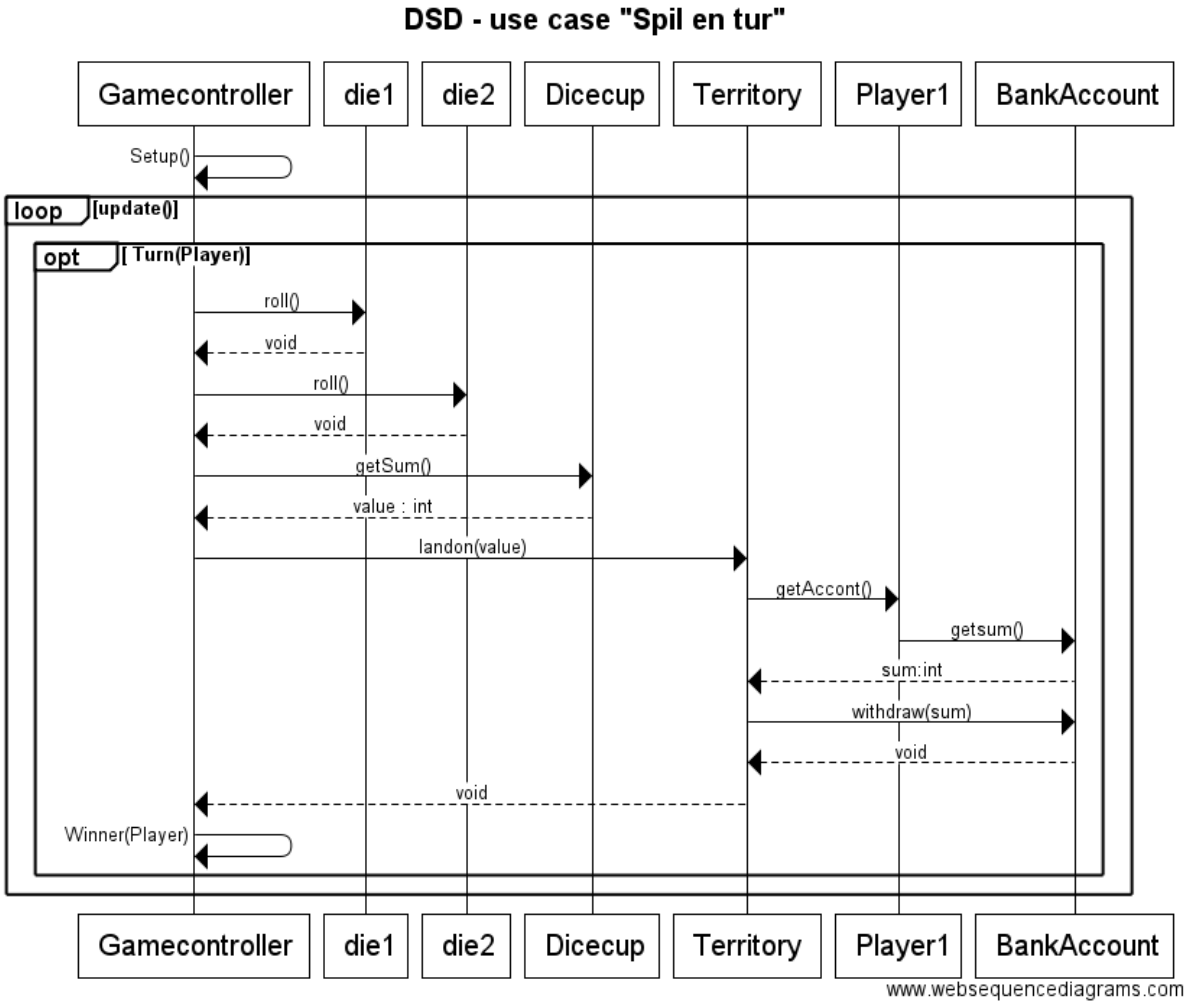
- System sekvens diagram
- Design sekvensdiagram
- Design klassediagram
- Implementering
- Test
- Dokumentation for overholdt GRASP

System Sekvens Diagram (SSD)



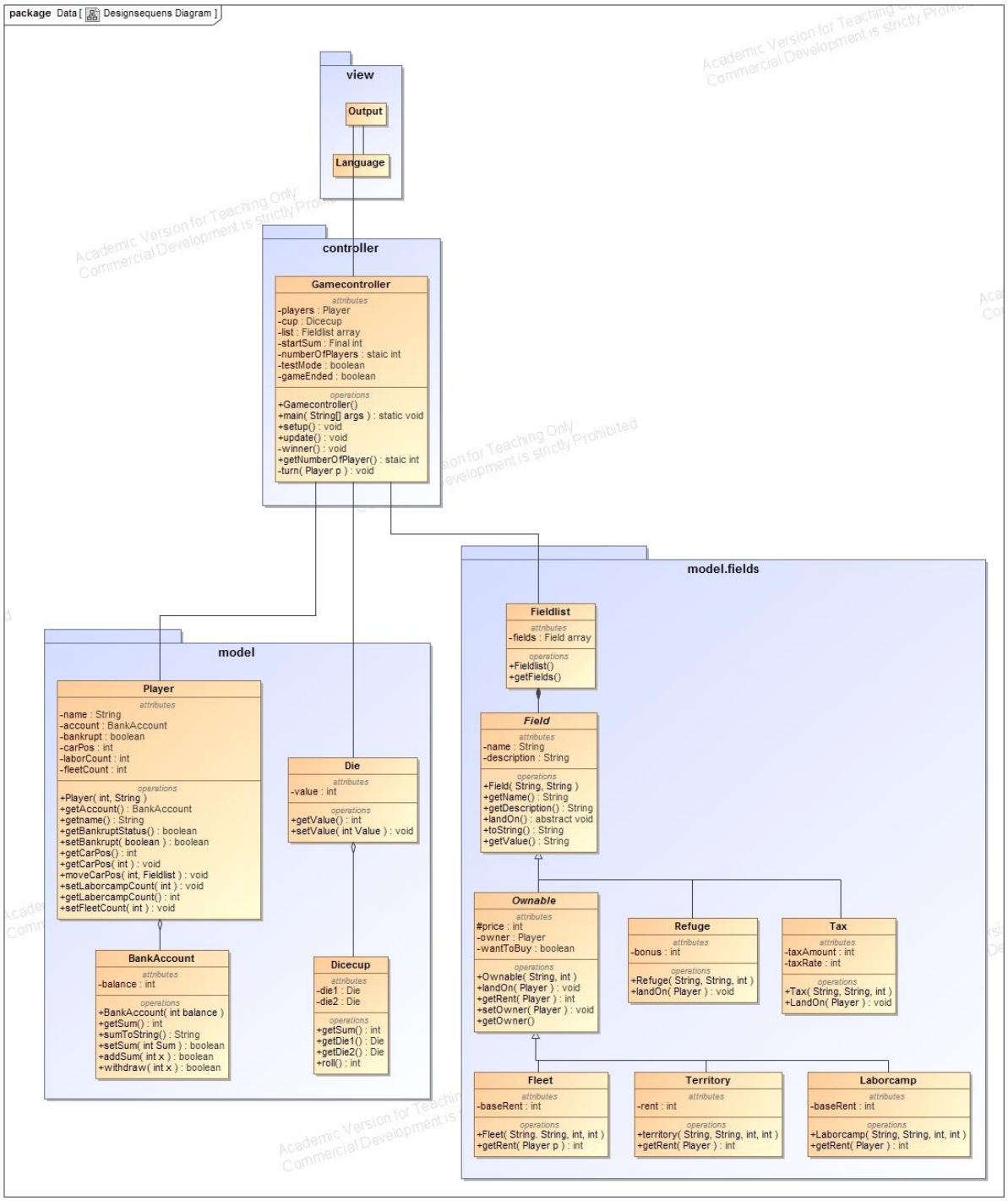
Figur 6 SSD. Viser sammenspillet mellem aktøren og systemet.

Design Sekvens Diagram (DSD)



Figur 7 Viser sammenspillet mellem vores klasser over use-casen "Spil en tur".

Design Klasse Diagram (DKD)



Figur 8 Design klassediagrammet dokumenterer vores kode. Det er en modificeret udgave af vores analyse klassediagram (Se figur 4)

Implementering

landOn() metoden indeholdes af alle felter nedarvet fra Field klassen. Det er denne metode som står for logikken ved hvert enkelt felt. Dette vil altså sige at ethvert type felt overskriver denne metode med en specialiseret *landOn()* til denne felt-type.

I klassen Fleet findes metoden LandOn(). LandOn metoden er et godt eksempel på eksekvering af et felt. Koden i fleet består af fem forgreninger som hver især håndterer et muligt udfald når spiller lander på et Fleet felt.

```
@Override
public void landOn(Player p){
    if(p.getAccount().getSum()>=price && getOwner() == null) // tjekker om player
    {   // can buy
        wantToBuy = Output.shop(price, p); // Spørg spilleren i GUIen om feltet s

        if(wantToBuy) // hvis playeren sagde ja til købet køres denne kode
        {
            Output.setColor(p);
            setOwner(p);

            p.setFleetCount(p.getFleetCount() + 1);

            p.getAccount().withdraw(price);
            Output.verificationOfPurchase();
        }
    }
    else if(p.getAccount().getSum() < price && getOwner() == null)
    {   // cant affort
        Output.deniedPurchase();
        System.out.println("You can't affort");
    }
    else if(p.getAccount().getSum()>=price && getOwner()  == null && !wantToBuy)
    {   //Player don't want to buy
        Output.deniedPurchase();
    }
    else if(getOwner()  != null && getOwner()  != p)// is owned
    {   // Pay rent
        p.getAccount().withdraw(getRent(p));
        getOwner().getAccount().addSum(getRent(p));
        Output.payedRent(p, getRent(p));
    }
    else
    {
        //It's your own field
        Language.ownField();
    }
```

Første if statement kontrollerer hvorvidt spiller har penge nok til at købe det respektive felt samt tjekker at der ikke allerede er en ejer af feltet. Hvis begge disse evalueres til true, bliver spilleren spurgt om spilleren vil købe dette felt for x antal penge. Et true vil udløse at koden inde i if statementet vil blive eksekveret. Feltet vil blive farvet i spillerens farve, spilleren bliver skrevet på som ejer af feltet, der vil blive lagt 1 til det antal af fleet felter som spilleren ejer, penge bliver trukket fra spillerens konto og spilleren vil i GUI'en blive mødt af en besked om at købet er gået igennem.

Hvis der ikke er nogen ejer af feltet, men spilleren ikke har penge nok får spilleren det at vide. Tilsvarende vises en besked hvis spilleren bare ikke vil købe feltet.

Hvis feltet allerede ejes af en anden spiller betales en afgift til ejeren. Lander man på et felt man selv ejer, sker der ikke noget.

De andre landOn metoder under ownable er meget lig denne, og refuge samt tax er ganske simple og bliver derfor ikke gennemgået i detaljer. Refuge og tax kan ikke ejes.

Arv

Det kan være smart at generalisere klasser så man kan nedarve og senere specialisere dem.

Skal man bruge metoder og data fra en klasse, kan man i stedet for at kopiere koden fra en klasse til andre klasser, bruge nedarvning. På denne måde kan man hente alle metoder og variabler fra en såkaldt superklasse ved at bruge extends og superklassenavnet. Det gør det også langt nemmere at vedligeholde én kopi af en metode i stedet for at skulle rette den samme metode i flere klasser. Man kan desuden overskrive og tilføje nye metoder i subklasserne, samt tilføje nye variabler. Subklasserne er altså en specialiseret udgave at superklassen.

Ønsker man at bruge default metoden, kan man bruge super i underklassen

Det er vigtigt man bruger public som acces type i superklassen, eller protected for at opretholde indkapsling, da subklasserne ellers ikke kan tilgå metoderne. Man kan stadig godt have private variabler, men så skal man bruge en get metode for at kunne bruge dem i subklasserne. Access type protected indkapsler ikke helt som private, men gør at det kun er klasser i den pågældende package der har adgang, samt subklasser.
Kontruktøren nedarves ikke, men man kan tilgå metoden og variablerne ved at bruge super.

Bruger man final, kan metoden ikke overskrives.

Bruger man abstact, kan man ikke oprette objekter af denne klasse. Abstract markerer man med kursiv når man skriver den ind i et klassediagram.

Test

Gennem softwareudviklingen har taget udgangspunkt i agile development tilgang med iterationer som udgangspunkt i UP modellen, derfor har det været smart at have fokus på Test Driven Development, som et led i risiko formindskelses process.

Dette er i praksis gjort ved at teste hver enkelt klasse, og dens tilhørende metoder, via en dedikeret JUnit test til hver klasse og vha. automatisk test. Alle JUnit testene er samlet i et test suite. Testene findes i test package.

Denne tilgang har vi valgt for, hurtigt og nemt at kunne identificere problemer i systemets forskellige enheder, samt overskueliggøre hvor og fange de fejl der måtte opstå undervejs.

Vores test dækker følgende:

- JUnit - af alle logiske enheder, klasser test af alle enheder (klasser)
- Automatisk test - general automatisk test vi bruger ved versionsstyring, hurtig test hver gang der laves en ny version.
- Integration - tester det samlede system
- White box test - debug på koden en linje af gangen.
- Black box test - user test af GUI (inkl. positiv og negativ test)
- Accepttest - tester om kravspecifikationen overholdes.

JUnit test

Testene kunne være foretaget ved at skrive en normal java klasse men JUnit har en fordel at attributterne bliver redigeret individuelt i selve testen, man skal derfor ikke tage højde for hvorvidt forud for testen er blevet ændret i attributterne. En anden fordel er at alle test er samlet i et JUnit test suite, og derved giver et hurtigt overblik om der er opstået nogle fejl efter modificering af koden.

Dette gør testingen nemmere og lægger op til en større mængde test under udviklingen, med et minimalt tidsforbrug, hvilket i sidste ende vil resultere i færre kritiske problemer undervejs og især mod slutningen af projektet, hvor større design fejl vil kunne være katastrofal, med denne metode er disse fejl forhåbentlig er blevet opdaget i den tidligt i udviklingen.

Dokumentation for alle JUnit test kan findes for hver enkelt specifik JUnit i kildekoden (i test-package).

Automatisk test – dedikeret testprogram

Ud over testning af de logiske enheder med JUnit test har vi også inkorporeret et testprogram. Testprogrammet er en automatisk test. Der tester hele systemet igennem med præfabrikeret testdata læst fra en ekstern fil, så resultatet kan tjekkes om det stemmer overens med det forventet resultat. Dette er en god måde at vide at det overordnet system returnere de rigtige data tilbage når de skal arbejde sammen. Dette er hurtigt og nem løsning, da testen kan genbruges, så man ikke ved hver iteration i skal udviklinge en ny test. Det kunne fx. Være en fordel ved testning af et specifikt felt på spillepladen, sådan at man ikke skal køre spillet og kaste terninger tilfældigt indtil man landede på det specifikke felt, hver gang man foretager en test.
Vores automatiske test bruger en falsk *Dicecup* med snyde terninger som erstatter de oprindelige tilfældige slag med nogle forhåndsbestemte værdier. *FakeDicecup*-klassen arbejder sammen med *Testdata*-klassen har ansvaret for at læsse den eksterne tekstfil og hente data ud af filen således at de kan bruge som terning-værdier. Dette gøres i praksis via et scanner objekt. Dette scanner objekt bliver derefter brugt af *readNextLine()* metoden som læser en linje ad gangen, og splitter dataen op ved "," således at de kan parsses til ind i et *int array*, så værdierne af disse int kan bruges som terning-værdier.

Eventuelle fejl automatisk test kan f.eks. skyldes fejl i testprogrammet eller hvis der er den samme fejl(f.eks. samme algoritme) i testprogrammet som selve programmet. Det kan svært at opdage og løse når testen ikke giver fejlmeddelelser.

Konfiguration

Bruger- og systemkrav

Velfungerende testet krav:
Windows 7-10/Vista/XP, Intel Core i7-3630QM, 16 GB Memory, NVIDIA Quadro K2000M. 12 KB Ledig lagerplads.

Anbefalet:
Mus eller trackpad.

Kildekode

Kildekoden kan køres i Eclipse. Der findes en værktøjslinje i toppen af programmet, med en grøn afspilningsknap. Programmet køres ved at vælge denne knap, fra klassen med main metoden, køre programmet. I vores program findes main metoden i klassen Gamecontroller.
programmet kan også køres uden for Eclipse. Eksport Funktionen i Eclipse findes under fanen filer og derefter 'Export' eksporter nu til en runnable jar-fil, som kan køres på alle maskiner med Java.

Konfigurationsstyring

For at kunne køre programmets kildekode er det krævet at flere stykker software er installeret på den anvendte computer. Det er vigtigt at *Java* og den tilhørende *Java SE Development Kit* er downloadet og installeret. Derudover er Eclipse nødvendigt for at kunne køre kildekoden - der er benyttet til udviklingen og produktionen af vores spil.

Udviklingsplatformen & Produktionsplatformen
I dette projekt er udviklingsplatformen og produktionsplatformen ens. Følgende er den nødvendige software versioner. Der ud over er biblioteket GUI.jar også nødvendigt.

Software versioner:
Java 8 Update 111 (build 1.8.0_111-b14), Java SE Development Kit 8 Update 101, Eclipse IDE for Java Developers Version: Neon Release (4.6.0) (build 20160613-1800).

Import i Eclipse fra git
For at importere kildekoden i "Eclipse", fra et såkaldt git repository, navigeres der til programmets import funktion som findes i fanen file. Nu skulle der gerne dukke et nyt vindue op, her udfoldes mappen "git". Tryk derefter på knappen git projekt. Et nyt vindue vil dukke op, i URL feltet indskrives en kopieret URL af det eksisterende repository.

CDIO del3 gruppe 18 git URI: https://github.com/trolund/CDIO18_del3.git

Konklusion:

Vi har formået at udvikle et program der simulerer et brætspil mellem 2-6 spillere. Produktet overholder alle kundens funktionelle krav. Programmets funktionalitet er dokumenteret og med test i JUnit test samt automatisk test. Der lægges vægt på at vores kode gør det muligt at både spilleren og hans bankbeholdning skal kunne bruges i andre spil, og at det er let at skifte til andre terninger samt ændrer sprog. Samtlige 21 felter er etableret og er funktionelle.

Vi har tilgået softwareudviklingen ved hjælp af *Agile software development*. Der er taget udgangspunkt i *Unified process* med tidsbegrænset iterationer.

Versionsstyring via github har fungeret optimalt med ganske få udfordringer.

Bilag 1.1. navneords- og udsagnsordsanalyse

Kundens vision: Rød er navneord og blå er udsagnsord.

Nu har vi terninger og spillere på plads, men felterne mangler stadig en del arbejde. I dette tredje spil ønsker vi derfor at forrige del bliver udbygget med forskellige typer af felter, samt en decideret spilleplade.
Spillerne skal altså kunne lande på et felt og så fortsætte derfra på næste slag. Man går i ring på brættet.
Der skal nu være 2-6 spillere.
Man starter med 30.000.
Spillet slutter når alle, på nær én spiller, er bankerot.

I bilag kan I se en oversigt over de felter vi ønsker, samt en beskrivelse af de forskellige typer.

Typer af felter:

- **Territory**
  - Et territory kan købes og når man lander på et Territory som er ejet af en anden spiller skal man betale en afgift til ejeren.
- **Refuge**
  - Når man lander på et Refuge får man udbetalt en bonus.
- **Tax**
  - Her fratrækkes enten et fast beløb eller 10% af spillerens formue. Spilleren vælger selv mellem disse to muligheder.
- **Labor camp**
  - Her skal man også betale en afgift til ejeren. Beløbet bestemmes ved at slå med terningerne og gange resultatet med 100. Dette tal skal så ganges med antallet af Labor camps med den samme ejer.
- **Fleet**
  - Endnu et felt hvor der skal betales en afgift til ejeren. Denne gang bestemmes beløbet ud fra antallet af Fleets med den samme ejer, beløbene er fastsat således:
    1. Fleet: 500
    2. Fleet: 1000
    3. Fleet: 2000
    4. Fleet: 4000

Bilag 1.2 Feltliste

Feltliste:

1.	Tribe Encampment	Territory	Rent 100 Price 1000
2.	Crater	Territory	Rent 300 Price 1500
3.	Mountain	Territory	Rent 500 Price 2000
4.	Cold Desert	Territory	Rent 700 Price 3000
5.	Black cave	Territory	Rent 1000 Price 4000
6.	The Werewall	Territory	Rent 1300 Price 4300
7.	Mountain village	Territory	Rent 1600 Price 4750
8.	South Citadel	Territory	Rent 2000 Price 5000
9.	Palace gates	Territory	Rent 2600 Price 5500
10.	Tower	Territory	Rent 3200 Price 6000
11.	Castle	Territory	Rent 4000 Price 8000
12.	Walled city	Refuge	Receive 5000
13.	Monastery	Refuge	Receive 500
14.	Huts in the mountain	Labor camp	Pay 100 x dice Price 2500
15.	The pit	Labor camp	Pay 100 x dice Price 2500
16.	Goldmine	Tax	Pay 2000
17.	Caravan	Tax	Pay 4000 or 10% of total Assets
18.	Second Sail	Fleet	Pay 500-4000 Price 4000
19.	Sea Grover	Fleet	Pay 500-4000 Price 4000
20.	The Buccaneers	Fleet	Pay 500-4000 Price 4000
21.	Privateer armade	Fleet	Pay 500-4000 Price 4000

Bilag 2. Litteraturliste

CDIO del 2 gruppe 18 URI: *https://github.com/trolund/CDIO18_del2.git*

CDIO delopgave 1 filer / links: 25/11-2016
https://drive.google.com/open?id=1oq5eSz1CzN2iuimXlzcOvg_xe-KVj68SNk3YaQBRiU8
CDIO delopgave 2 filer / links: 25/11-2016
https://drive.google.com/open?id=1LfQo3O_GjCfKtwcd85uvGgeEpQAat8DRjPXjma-JQL0
CDIO delopgave 3 filer / links: 25/ 11-2016
https://drive.google.com/open?id=1UsCw9jeOvZlthzUrxIg0sF8ez1xwk-64oh6KivQvzHI

Applying UML and Patterns An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3. Udgave 2004 af Craig Larman