CDIO3

GRUPPE 35

s154023 - Thomsen, Oliver Elleman

s153643 - Jensen, Simon James

s153476 - Mendis, William

s153670 - Larsen, Sofie Paludan

S144730 - Werner, Mads

Denne rapport er afleveret via Campusnet, afleveringsfrist: lørdag d. 28/11 – 2015, kl. 04:59
Rapporten består af 48 sider inkl. denne side
Kurser: 02313 Udviklingsmetoder til IT-systemer, 02314 Indledende Programmering,
02315 Versionsstyrring og testmetoder

Timeregnskab

CDIO3

Deltager	Design	Impl.	Test	Dok.	I alt
Mads	11	4	2	6	23
Oliver	5	14	2	4	25
Simon	8	5	1	7	21
Sofie	6	5	4	10	25
William	7	10	5	3	25

Tabel 1: Timeregnskabet i dette projekt.

Indholdsfortegnelse

Resume

Kunden ønskede et digitalt Matador-lignende spil, hvor 2-6 spillere en efter en skal kaste to terninger og rykke deres brikker et tilsvarende antal felter frem på en 21 felters spilleplade. Undervejs skulle de have mulighed for at opkøbe ejendomme og blive opkrævet husleje, samt modtage diverse bonusser og bøder. Når en spiller mister alle sine penge, skal den vedkommende gå bankerot, og den sidste tilbageværende spiller vinder spillet. Yderligere tog gruppen nogle designbeslutninger, der ikke var specificeret af kunden, da vi vurderede, at de ville øge spillets funktionalitet.

Vi benyttede UML-diagrammerne: use-cases, domænemodeller, sekvensdiagrammer, samt klassediagrammer til at udvikle spillet, og Java til at kode det. Der har desuden været fokus på at overholde GRASP-principperne omkring creator- og ekspert-mønstrene, samt brug af low coupling og high cohesion, og klasserne er designet med arv i tankerne.

Versionering har været styret gennem Github, og flere af kodens kernekrav er blevet testet vha. JUnit i Eclipse.

Alt i alt er projektets mål nået inden for den angivne tidsramme.

Alle relevante bilag er samlet sidst i rapporten.

Indledning

I dette projekt udarbejdes spillet fra forrige projekt, hvor vi har fået til opgave at inkorporere en spilleplade samt en række andre af det udleverede "GUI" biblioteks features. Der skal nu, ifølge kundens ønske, være 2-6 spillere, som alle starter med en pengebeholdning på 30.000. Hver spiller får derudover også tildelt, som kan rykke rundt på spillepladen og indikere brugerens placering. Der vil blive tilføjet flere muligheder når man rykker rundt på de i alt 22 forskellige felter. Felterne i dette spil er opdelt i forskellige typer, hvilket betyder, at de også har forskellige virkninger på spillet. Eksempelvis kan nogle af felterne nu ejes og opkræver penge fra andre spillere, som lander på det ejede felt. Yderligere beskrivelse af felterne kan findes under design-dokumentationen. Spillet slutter først, når kun én spiller sidder tilbage med penge.

Rapporten skrives på dansk, men kodning og kommentarer i koden, samt diagrammer, skrives på engelsk. Hvis et begreb relateret til kodningen eller diagrammerne benyttes, skrives dette på engelsk.

Til kortlægning af spillets krav, er der med input fra kunden udviklet en kravspecifikationen, samt diagrammer for use cases, domænemodeller, BCE modeller, system sekvensdiagrammer, design-sekvensdiagrammer og design-klassediagrammer.

Spillet kodes er skrevet i sproget Java, i programmet Eclipse, og diagrammerne er tegnet i Magic Draw.

Analyse

Kravspecifikation

I denne kravspecifikation definerer og specificerer vi kravene til spillet. Kravene er formet ud fra kundens vision (se bilag 1). Nogen krav har vi selv måtte tage stilling til, eftersom kunden ikke har givet udtryk for egne krav på disse områder. Kravspecifikationen skal danne grundlag for vores spil, hvordan det skal fungere, udvikles osv., og samtidigt oplyse begge parter om, hvilke forventninger der er til spillet og hvad det indeholder.

Navneordsanalyse

For at skabe et bedre overblik over, hvilke klasser og genstande vi kan inddrage i spillet, har vi foretaget en navneordsanalyse af kundens vision:

- Terninger
- Spiller
- Felter
- Spil
- Spilleplade
- Slag
- Brættet
- Bankerot

Funktionelle krav

1. Følgende er krav som kunden ønsker, og vi gerne vil imødekomme:
    1. Spillere slår med to terninger
    2. Der skal være en spilleplade
    3. Spillepladen skal indeholde 21 på forhånd specificerede felter
    4. En spiller skal kunne lande på et felt
    5. Spilleren skal fortsætte fra nuværende felt i næste tur
    6. Spilleren skal gå i ring på brættet
    7. Spillet skal indeholde 2-6 spillere
    8. En spillers startbalance er 30.000
    9. Spillet slutter når alle spillere på nær én går bankerot

2. Kundens vision dækker ikke alle scenarier, der kan forekomme i spillet, hvilket betyder, at vi selv har måtte tage nogle valg. Nedenfor er nogle af de "ekstra krav", eller antagelser, som vi har gjort os:
    1. Et startfelt #1 der ikke giver point skal implementeres
    2. En spillers konto skal ikke kunne være negativ, hvis balancen er under 0 bliver den automatisk sat til 0 igen.
    3. En spiller går bankerot når spillerens konto går i 0.
    4. Når en spiller går bankerot mister han alle sine grunde
    5. Når en spiller går bankerot bliver hans bil fjernet

6. Når en spiller går bankerot kan han ikke spille længere
7. En spiller kan godt købe en ejendom selvom dette vil resultere i bankerot.

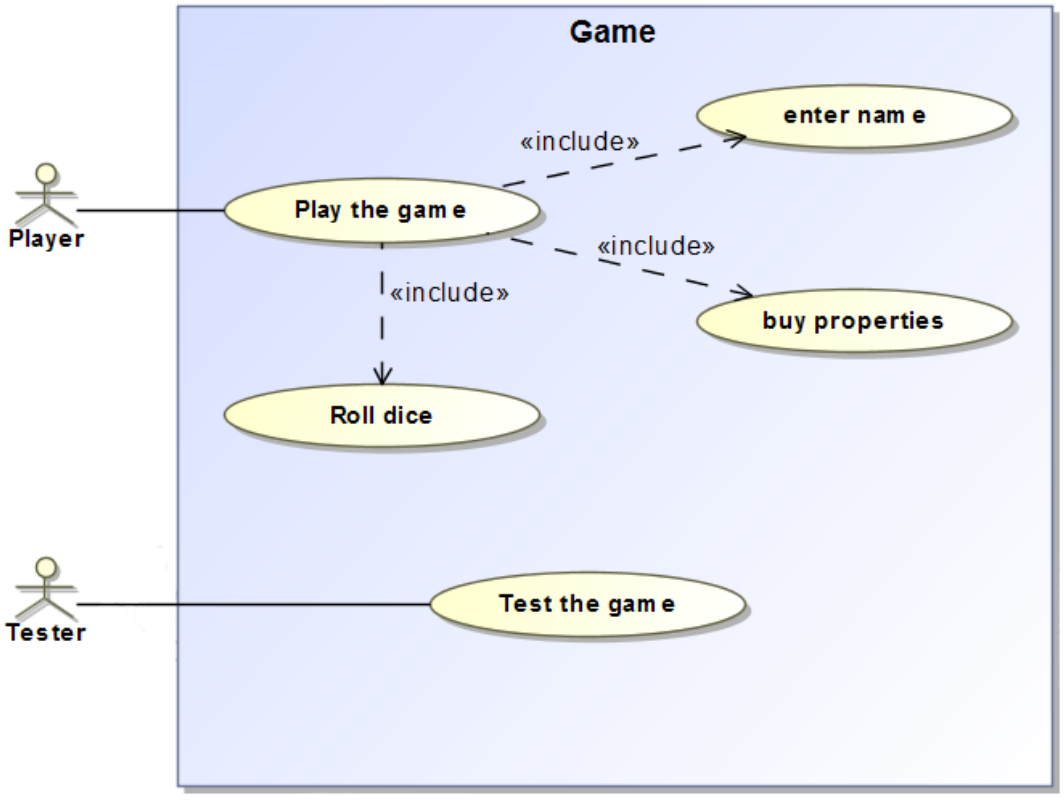
Ikke-funktionelle krav

I dette projekt er der ingen ikke-funktionelle krav.

Use-case diagram

I vores use-case indgår to aktører; Player og Tester, der henholdsvis spiller spillet og tester spillet. Player kan indtaste sit navn og slå med terninger, hvilket vil rykke spillerens avatar rundt på spillepladen, hvor han også kan købe ejendomme.

Tester tilgår spillet med det formål at køre diverse testscases, der er beskrevet under Test afsnittet længere nede.



Figur 1: Use-case diagram.

Use-case specification

Use case: Landed on fleet (ikke-ejet/køber ikke)
ID: 1
Kort beskrivelse: Spiller lander på én af 'Fleet' felterne
Primære aktører: Spiller
Sekundære aktører: Ingen
Prækondition: Spiller har: 1) oprettet spillernavn, 2) rullet med terningerne
Main flow: 1. Spiller lander på et "Fleet"-felt 2. Spillet præsenterer spiller for tekst, der tilbyder ham at købe grunden, samt to svarknapper (Ja/Nej). 3. Spiller trykker 'Nej'.
Postkondition: Ingen ændring i ejerskabsforholdene. Turen går videre til næste spiller.

Tabel 2: Use-case specifikation fra landedOn fleet, ID 1.

Use case: Landed on fleet (ikke-ejet/køber)
ID: 2
Kort beskrivelse: Spiller lander på én af 'Fleet' felterne
Primære aktører: Spiller
Sekundære aktører: Ingen
Prækondition: Spiller har: 1) oprettet spillernavn, 2) rullet med terningerne
Main flow: 1. Spiller lander på et 'Fleet' felt. 2. Spillet præsenterer spiller for tekst, der tilbyder ham at købe grunden, samt to svarknapper (Ja/Nej). 3. Spiller trykker 'Ja'. 4. Spillet trækker grundens pris fra spillers bankkonto. Hvis dette resulterer i en tom bankkonto, går spiller bankerot og elimineres fra spillet. Alle spillerens ejendomme, inklusiv den der lige er købt, skifter nu status til 'ikke-ejet'.
Postkondition: Spilleren ejer nu feltet, med mindre han gik bankerot. Turen går videre til næste spiller. Hvis næste spiller er den eneste spiller, der ikke er gået bankerot vinder denne spillet.
Alternative flows: ID1: Landed on fleet (ikke-ejet/køber ikke) ID3: Landed on fleet (ejet)

Tabel 3: Use-case specifikation for landedOn fleet, ID 2.

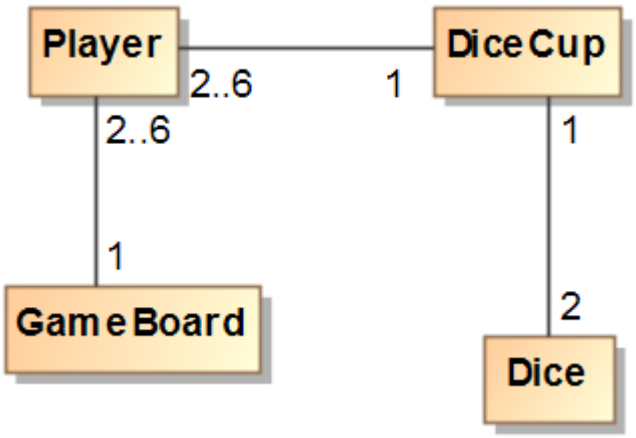
Use case: Landed on fleet (ejet)
ID: 3
Kort beskrivelse: Spiller lander på én af 'Fleet' felterne
Primære aktører: Spiller
Sekundære aktører: Ingen
Prækondition: Spiller har: 1) oprettet spillernavn, 2) rullet terninger
Main flow: 1. Spiller lander på et 'Fleet' felt. 2. Spillet informerer spilleren om, at feltet er ejet, hvem der ejer feltet, samt hvad spilleren skal betale. 3. Spillet trækker pengene fra spillers bankkonto og indsætter tilsvarende på ejers bankkonto. Hvis dette resulterer i en negativ bankkonto går spiller bankerot og elimineres fra spillet. Alle spillerens ejendomme skifter status til 'ikke-ejet'.
Postcondition: Turen går videre til næste spiller. Hvis næste spiller er den eneste spiller, der ikke er gået bankerot vinder denne spillet.
Alternative flows: ID1: Landed on fleet (ikke-ejet/køber ikke) ID2: Landed on fleet (ikke-ejet/køber)

Tabel 4. Use-case specifikation for landedOn fleet, ID 3.

Domænemodel

Domænemodellen repræsenterer spil-elementerne og sammenhængen mellem disse i et "real-world"-scenarie. For at lave en domænemodel skal man altså stille sig selv spørgsmålet; "Hvad ville disse objekter repræsenterer hvis vi spillede spillet i den virkelige verden"?

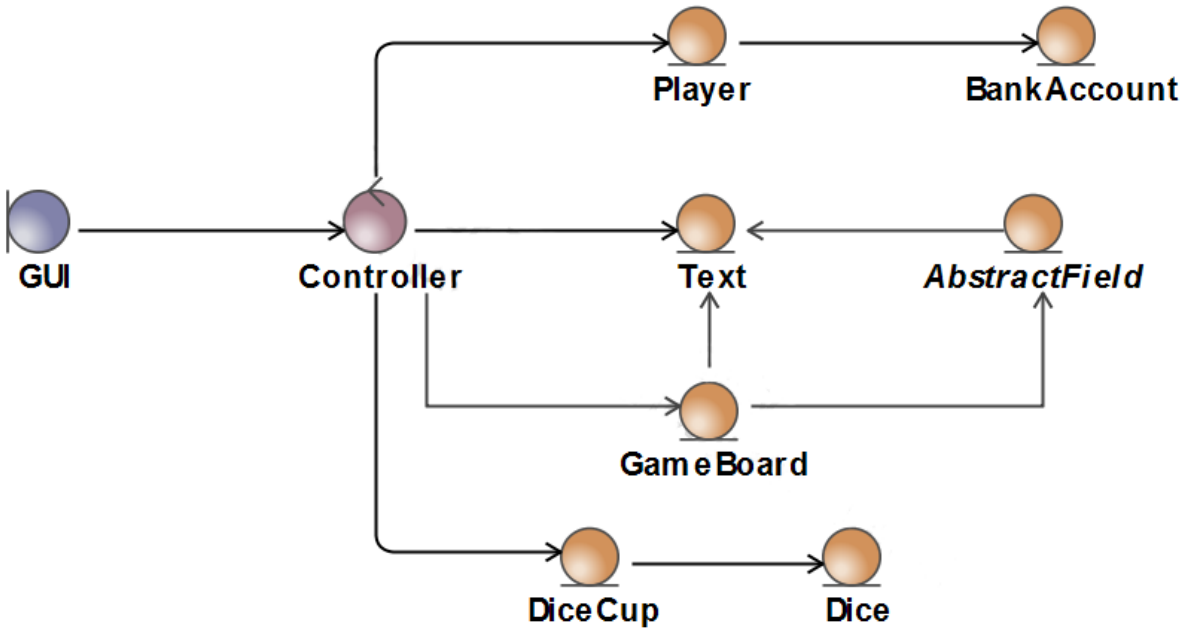
Diagrammet fortæller os, at der kan være 2-6 spillere, en spilleplade "gameboard" og et raflebægere "dicecup" som indeholder to terninger "dice".



Figur 2: Domænemodel.

BCE

I BCE-modellen beskriver vi GUI'en som vores boundary. Den styrer nemlig interaktionen mellem aktøren og og systemet, hvor den holder disse adskilt. Det kan vi se, når aktøren benytter sig af spillet og kun ser brugerfladen. Videre har vi controlleren, som styrer klasserne. Den benytter sig af metoder og attributter fra de entities vi ser på diagrammet. Nogle af disse entities bruger andre entities, for eksempel DiceCup'en, der bruger Dice.



Figur 3: BCE-model.

Design-dokumentation

Felter

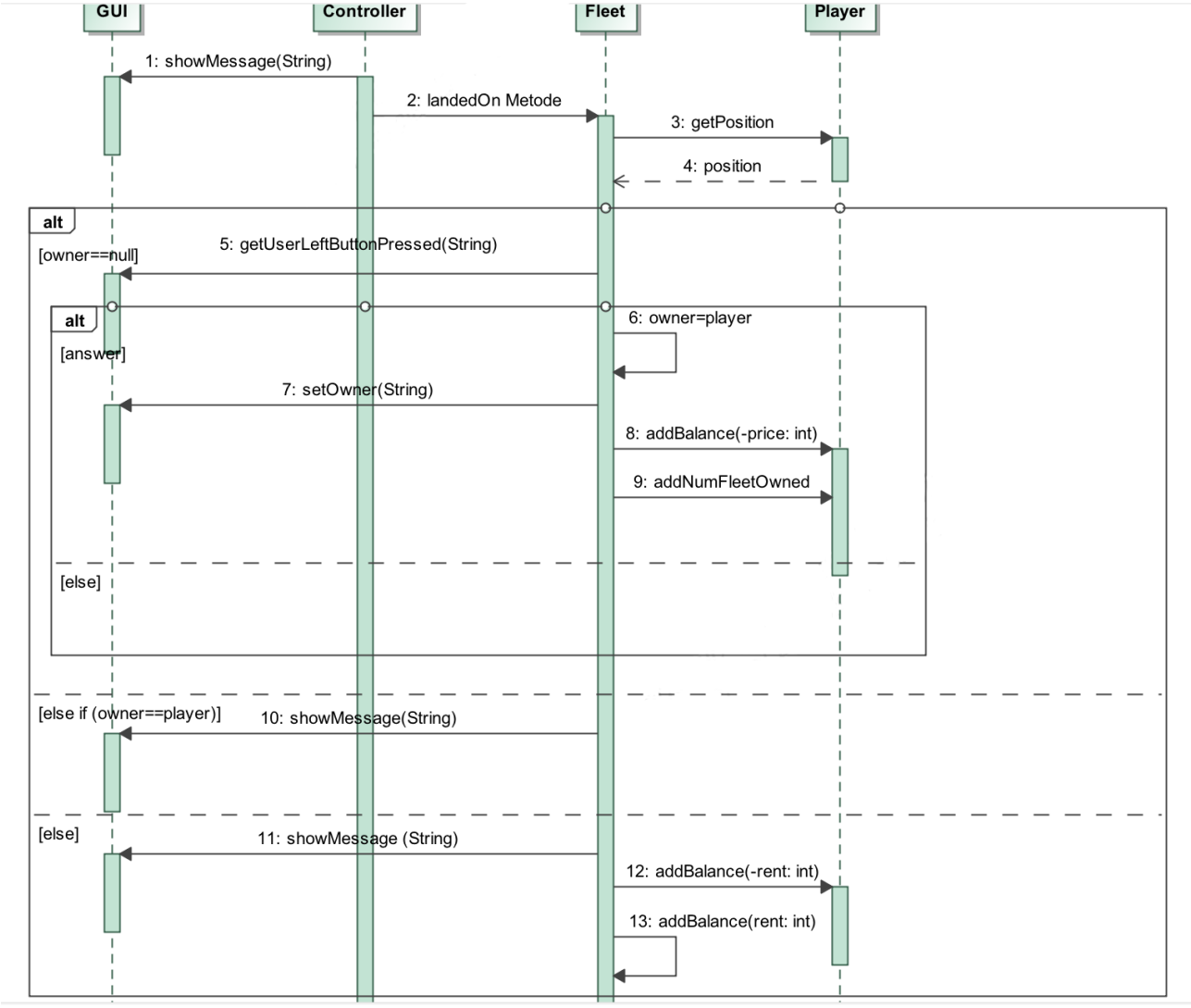
I nedenstående tabel er en oversigt over de felter, som indgår i spillet. Tabellen er ordnet efter felternes rækkefølge som vi har besluttet den skal være. Vi har valgt at blande rækkefølgen, så de forskellige felttyper er fordelt ligeligt over pladen.

Field#	Name	Type	Rent	Price	Receive
1	START				
2	Tribe Encampment	Territory	100	1000	
3	Goldmine	Tax			-2000
4	Crater	Territory	300	1500	
5	Second Sail	Fleet	500-4000	4000	
6	Mountain	Territory	500	2000	
7	Monastery	Refuge			500
8	Cold Dessert	Territory	700	3000	
9	Sea Grover	Fleet	500-4000	4000	
10	Black Cave	Territory	1000	4000	
11	Huts in the Mountain	Labor Camp	100 x dice	2500	
12	The Warewall	Territory	1300	4300	
13	Caravan	Tax			-4000 or -10%
14	Mountain Village	Territory	1600	4750	
15	The Buccaneers	Fleet	500-4000	4000	
16	South Citadel	Territory	2000	5000	
17	Walled City	Refuge			5000
18	Palace Gates	Territory	2600	5500	
19	The Pit	labor Camp	100 x dice	2500	
20	Tower	Territory	3200	6000	
21	Privateer Armade	Fleet	500-4000	4000	
22	Castle	Territory	4000	8000	

Tabel 5: Feltoversigt.

Design sekvensdiagram for landedOn Fleet

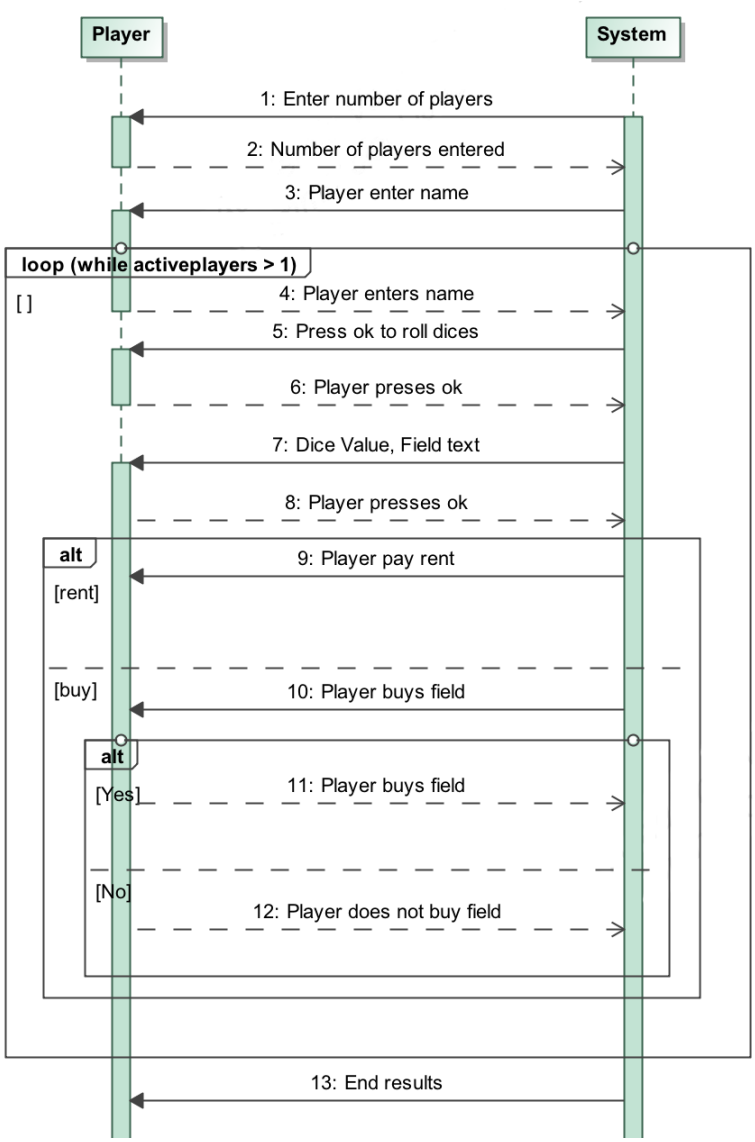
Designsekvensdiagrammet tillader os at se interaktionen mellem spillets objekter i den sekvens de forekommer i. Øverst i de firkantede parenteser er kodens klasser og deres interaktioner. Dette design sekvensdiagram påviser, hvad der sker, når spilleren lander på et "Fleet" felt. Efter controlleren har registreret, at spilleren er landet på fleet, så viser diagrammet, at der er tre alternativer/mulige handlinger. Hvis [owner=null] så kan spilleren vælge at købe grunden. Hvis [owner==player], så fremkommer der kun en besked. Og hvis ingen af de ovenstående er gældende, så skal spilleren betale en afgift til ejeren. Her viser diagrammet, at metoden "addBalance" bliver benyttet, hvori spilleren mister leje beløbet (-rent), og ejeren får leje beløbet (rent). Der er ikke nogen livslinje for "AbstractOwnables", da Fleet klassen er en extension af "AbstractOwnables".



Figur 4: Design sekvens-diagram.

System sekvensdiagram

Et sekvensdiagram forklarer hele spillets flow fra start til slut. Her i system sekvens-diagrammet ser vi interaktionen mellem brugeren og systemet. Når brugeren starter spillet, sender systemet en besked, som spørger om antallet af spillere, der skal spille spillet. Brugeren kan ikke komme videre i spillet uden at indtaste det systemet spørger om. Det er her ikke muligt at indtaste værdier højere end 6, eller lavere end 2. Herefter skal alle spillere indtaste deres navne. Når spillerne ruller med terningerne, så kommer der et output med terningens værdi og det pågældende felts tekst. Det kommer an på feltet mht., hvad det næste skridt er for spilleren. Hvis spilleren for eksempel kan købe grunden, skal han tage action (vælge ja eller nej). Der er lavet et loop omkring terningekastet, da spillerne skal blive ved med at kaste terningerne så længe der er mere end en aktiv spiller. Når spillet kører, så fremkommer der nogle "alternatives" når man lander på et felt. "[alt]" påviser, at hvis spilleren ikke skal betale rent (hvis feltet ikke er ejet), så har han mulighed for at købe det. Når der kun er én aktiv spiller tilbage, vil systemet udskrive slut resultaterne.



Figur 5: System sekvens-diagram.

Arv

Ved at bruge arv i java, eller inherit på engelsk, kan man nemt og hurtigt fordele metoder på sine klasser, uden at skulle gå dem alle sammen slavisk igennem. På den måde kan man spare en masse tid og have et bedre overblik over sine klasser.

Den klasse, som indeholder metoden, kaldes for super. I denne klasse ligger metoden, som de andre klasser arver. Skal man referere til super-klassen fra sin sub-klasse, bruges "super". Er der eksempelvis ændringer, kan man nemt og hurtigt gå ind i sin super-klasse og ændre i metoden, hvilket vil træde i kraft for alle under-klasserne. Underklasserne, eller sub-klasserne, er dem der arver.

I java skriver man "extends" for at indikere, at man vil arve. Derefter skrives hvilken klasse, man ønsker at arve fra. Dette illustreres i eksemplet nedenfor, hvor vi har en klasse kaldet "AbstractOwnables", som arver fra klassen "AbstractField".

```
4
5 public abstract class AbstractOwnables extends AbstractField{
6
```

Figur 6. Screenshot af vores abstrakte klasse, her som et eksempel.

Man kan også skabe flere underklasser af sine underklasser, og på den måde oprette en form for systematisk klassehierarki.(se eksempel på dette længere nede i Arvehierarki)

Ønsker man, at enkelte sub-klasser skal indeholde nogle ændrede værdier, men stadig den samme metode, gøres dette ved at skrive hele metoden op i den pågældende sub-klasse. Vi kan anbefale at benytte sig af copy/paste fra super-klassen, så man undgår syntaks-fejl eller andre unødvendige fejl. Man laver derefter sine ønskede ændringer og lader metoden stå i sub-klassen. Sub-klassen vil da "override", hvad super-klassen ellers har tildelt metoden.

Abstract

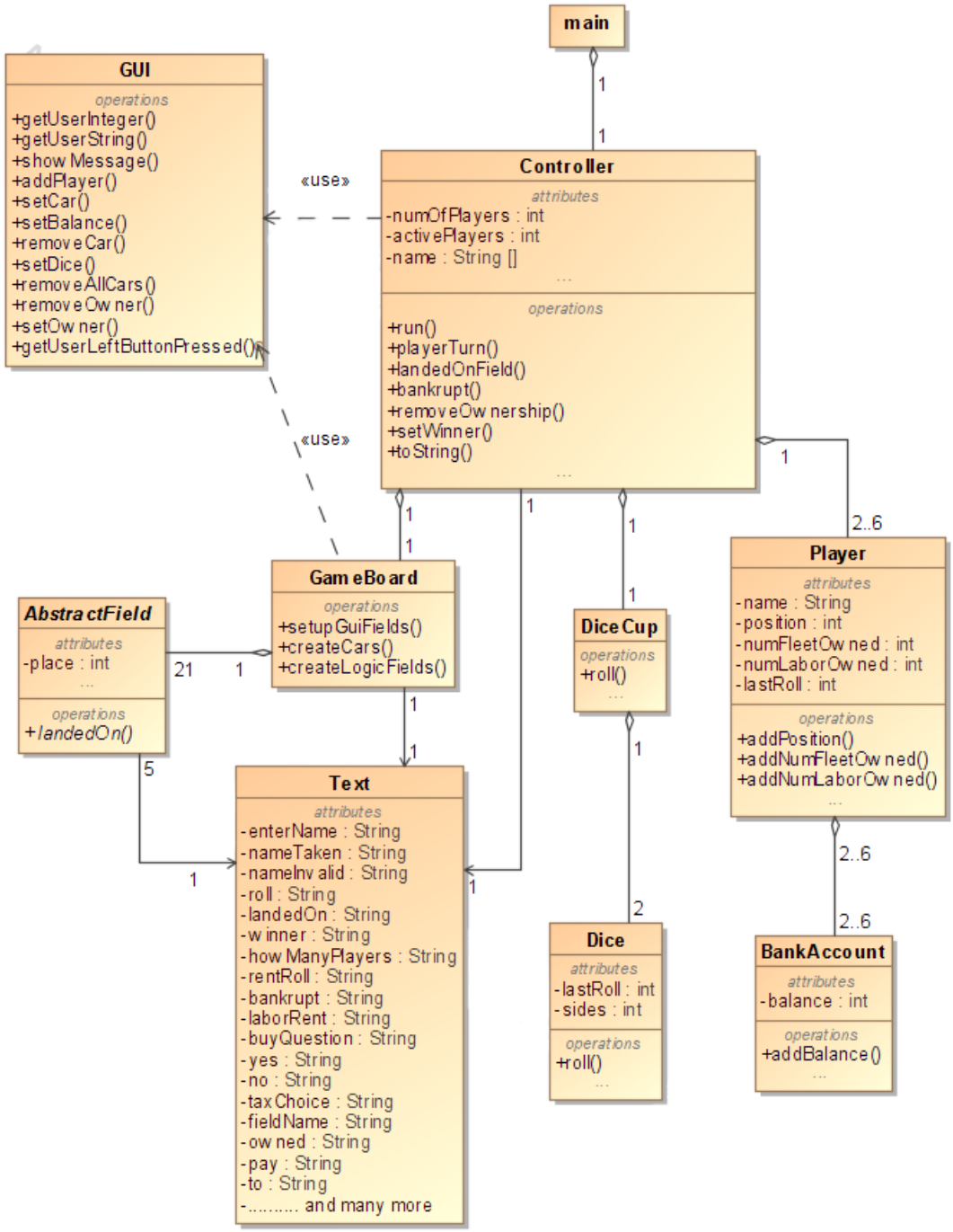
Når en klasse er abstrakt betyder det, at den indeholder en eller flere metoder, som er abstrakte. Abstrakte metoder er metoder, som endnu ikke er definerede. Disse abstrakte metoder skal dog defineres på et tids-punkt. Dette kan for eksempel ske i en underklasse, der har arvet fra den abstrakte klasse. Man definerer metoden ved at "override" den originale abstrakte metode, og derefter lave en krop til metoden, der be-stemmer, hvad den gør.

Klassediagram

Klassediagrammet viser alle klasserne i spillet, hvad de indeholder og hvordan de interagerer med hinanden. Under hver klasse kan man se hvilke metoder og attributter de indeholder.

Den stiplede <<use>>-pil der går fra Controller til GUI, hvilket betyder, at Controlleren bruger de nævnte metoder, der ligger i GUI'en.

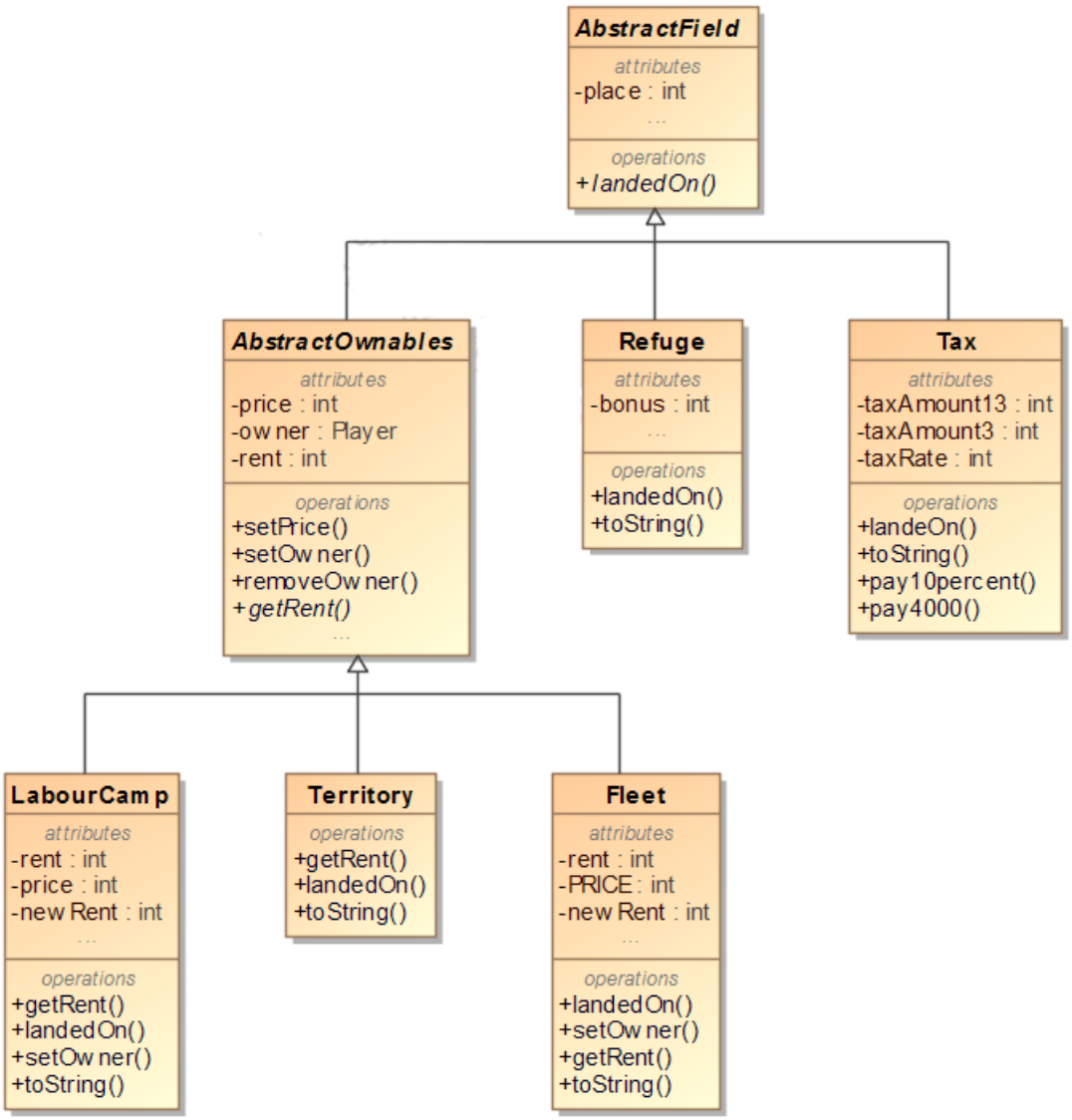
Det diamant-formede symbol, som binder for eksempel “Dicecup” og “Dice”, er en aggregation. Den symboliserer, eksempelvis, at “DiceCup”-klassen bruger “Dice”-klassen. Det kan ses, at controlleren bruger 21 instanser af klassen “AbstractFields”. “AbstractFields” har en masse underklasser, som arver fra “AbstractFields”. Dette kan ses i arvehierarkiet længere nede i rapporten.



Figur 7: Klassediagram.

Arvehierarki

Dette diagram viser alle underklasserne til "AbstractFields". Disse underklasser er extensions af klassen "AbstractField". Dette kan ses ved de karakteristiske pile, der er brugt i diagrammet. Dette diagram indeholder, på samme måde som klassediagrammet, en masse metoder og attributter for alle klasserne.



Figur 8: Arvehierarki.

Dokumentation for overholdt GRASP

Controller-klassen

Vores controller fungerer som hub for alle hændelser i spillet. Alle kommandoer, der registreres fra brugeren, sendes fra GUI'en til controller-klassen, som så sørger for at videregive informationen til relevante system-objekter. Eksempelvis anmoder GUI'en brugeren om at indtaste antallet af spillere. Tallet der returneres danner grundlag for et nyt array, der efterfølgende fyldes med et tilsvarende antal Player-objekter, som instantieres ud fra Player-klassen. Parametre dannes delvis med yderligere bruger-input, eller fra constructorens prædefinerede parametre.Derudover sørger controlleren for at håndtere alle systemoperationer, så som at holde styr på alle spillernes ture, eliminere spillere når de går bankerot og erklære en vinder. På denne måde sænkes coupling mellem GUI og klasserne, samt indbyrdes mellem klasserne. Dette tillader os at tilpasse enten klasserne eller GUI'en løbende.

Ekspert-mønsteret

Klasser, der instantierer objekter, indeholder naturligvis al information som disse instanser skal bruge for at fungere.

Polymorfisme

I spillet er felterne, som spilleren kan lande på (de såkaldte AbstractField-klasser), opdelt i felter, der kan ejes, der giver spilleren en bonus og der giver spilleren en bøde. Felterne, der kan ejes, er yderligere opdelt i tre undertyper. Disse koster andre spillere penge, såfremt de lander på dem. For at tillade denne dynamik, er felterne konstrueret ud fra et polymorfisk hierarki, hvor de fem forskellige typer Field-klasser kan arve og override attributter og metoder fra de overliggende klasser på deres respektive grene. Eksempelvis det mest almindelige felt i spillet: Territory-klassen. Som alle felterne i spillet har feltet i Territory-klassen en nummerværdi, der identificerer dens position, og en metode, der bliver kaldt hvis man lander på det. Klassens "bedstefar" er derfor AbstractField-klassen. Da Territory derudover kan have en ejer, en værdi og en husleje udvides AbstractField med AbstractOwnables-klassen, der indeholder disse attributer, samt settere og gettere til at bruge dem. Da Territory-klassen har to søskende, der også kan ejes, men hvis husleje er dynamisk og ikke statisk som Territorys, udvides AbstractOwnables til tre yderligere klasser, der indeholder attributter og metoder til at differentiere de tre klasser.

Denne tilgang mindsker vores afhængighed af statiske conditional-statements til at holde styr på hvad der skal ske, når en spiller lander på et felt. Dette gør det lettere at tilføje, fjerne og modificere spillets felter i senere versioner af spillet.

Low coupling

Brugen af polymorfisme sænker kodens coupling og tillader os, som nævnt tidligere, en højere grad af fleksibilitet når det kommer til at rette og udvide koden. Naturligvis betyder en ændring i en klasse, der nedarver til en anden klasse, at denne også bliver berørt af ændringen, men hvis forholdet mellem disse klasser er fornuftigt planlagt, burde problemerne være minimale. Vores brug af en enkelt Controller-klasse sørger desuden for, at de fleste klasser opererer uafhængigt af hinanden, og eventuelle konflikter, der kunne opstå ved ændring i koden, ligeledes burde være minimale.

High cohesion

Alle vores objekter har et meget fokuseret ansvarsområde. Hvis en klasse skal kunne håndtere flere ting, dannes der enten abstrakte underklasser, til at håndtere disse ansvar, eller også fungerer klassen som creator for et objekt, der kan varetage dette ansvarsområde (for eksempel adskillelsen af Player- og Account-klasser).

GRASPed

Yderligere information om ovenstående kan ses i relevante designdokumenter. Disse skulle gerne, set sammen med koden, understrege, at spillet er designet med henblik på fleksibilitet og mulig udbygning i fremtiden.

Hvad er landedOn

LandedOn er først og fremmest en abstrakt metode, der bliver lavet i klassen "AbstractFields". Metoden bliver senere tildelt en krop med noget kode, som bestemmer, hvad metoden skal gøre. Dette sker i hver af underklasserne: Territory, LaborCamp, Fleet, Refuge og Tax. Hver klasse har så sin egen landedOn-metode. Når man opretter en instans af en af disse klasser, vil de automatisk blive tildelt deres egen landedOn-metode, der hører specifik til dette felt. LandedOn-metoden bestemmer, hvad der skal ske, når man lander på det felt. Det kan for eksempel være, at man kan købe det, betale en leje, eller få udbetalt en bonus. Det eneste felt, der ikke har en landedOn-metode, er Start.

Konfigurationsstyring

Udviklings- og produktionsplatformen

Under udvikling af vores spil, er følgende platform blevet brugt:

- OS: Windows 8 og 10, samt OSX.
- IDE: Eclipse, Version: Mars Release (4.5.0)
- Java (Java Runtime Environment(JRE) & Java Development Kit(JDK)
- GUI.jar
- GitHub

Import af spillet til Eclipse via archive file

1. Tryk på “File” i værktøjslinjen
2. Tryk på “Import”
3. Vælg herefter “Existing Projects into Workspace” og tryk “Next”
4. Markér “Select archive file” og tryk på “Browse” for at finde den file du gerne vil importere.
5. Når du har valgt din fil, tryk da på “Finish”
6. Projektet er nu importeret

Sådan køres programmet

Når projektet er importeret, køres det fra klassen “Main” der ligger i mappen “main”. Den indeholder nemlig kun en main metode. For at kører main metoden, højre klik da på klassen og tryk på “Run As” og vælg “Java Application”.

Versionsstyring

Link til vore online git repository: https://github.com/Flubber13/35_CDIO3.git

Import af spillet til Eclipse via Github.com

For at importere vores projekt gennem et Github-link, skal man blot bruge Eclipses import-funktion, givet at EGit er korrekt opdateret.

1. Vælg "Import" under den første menu i værktøjslinjen: "File".
2. Et pop-up vindue dukker nu op, hvor du skal specificere, hvad du vil importere. Under mappen "Git" vælges "Project from Git". Klik derefter "Next >".
3. Vælg "Clone URI" og klik videre.
4. Har du kopieret linket til vores repository, altså vores URI, vil Eclipse pr. automatik udfylde felterne under "Location". Har du en bruger på Github.com, vil den også indsætte din oplysninger under "Authentication".
5. Hvis alt ser rigtigt ud, klikkes der "Next >".
6. Vælg "master" i Branch Selection. Dette er vores aktive kode.

Test

Testrapporter

For at teste om koden overholder de krav, vi har stillet, laver vi nogle negative og positive tests. De negative tests indebærer, at vi ændrer værdier, navne eller andet til noget man ikke må, og ser, hvordan programmet reagerer. Vi forventer, at programmet på trods af de "ulovlige" input stadig kan fungere korrekt. I de positive undersøger vi blot, om koden fungerer efter vores hensigt.

Testrapport 1.1: (Negativ) Tester om balancen kan gå i minus	
Pre: 1. Opretter en spiller med et navn og en positiv balance (+3000)	X
Test: 1. Ligger -4000 til spillerens balance	X
Post: 1. Forventer, at spillerens balance bliver sat til 0	X

Tabel 6: Testrapport 1.1, negativ test over spillerens balance.

Testrapport 1.2: (Positiv) Tester om balancen bliver fratrukket korrekt	
Pre: 1. Opretter en spiller med et navn og en positiv balance (+3000)	X
Test: 1. Ligger -1000 til spillerens balance	X
Post: 1. Forventer, at spillerens balance bliver sat til 2000	X

Tabel 7: Testrapport 1.2, positiv test over spillerens balance.

Testrapport 2 : Tester owner	
Pre: 1. Opretter en spiller 2. Opretter et felt af typen ownable 3. sætter spilleren til at eje feltet	X
Test: 1. Tester om spilleren er den samme som ejeren 2. Pre: removeOwner() tester om ejeren er blevet fjernet	X
Post: 1. Forventer, at spiller og ejer er den samme. 2. forventer at spilleren er blevet fjernet som ejer.	X

Tabel 8: Testrapport 2, tester ejeren.

Testrapport 3: Tester terningens sandsynligheder	
Pre: 1. Opretter et raflebæger 2. laver et array der indeholder alle teoretiske sandsynligheder 3. Ruller terningen 1 million gange, og gemmer data i et array til beregnede sandsynligheder	X
Test: 1. Sammenligner alle værdierne i de to arrays (teoretiske med beregnede) med en usikkerhed på 0.001	X
Post: 1. Forventer, at alle værdier stemmer overens inden for usikkerheden, og testen kører uden fejl.	X

Tabel 9: Testrapport 3, test over terningens sandsynlighed.

I testrapport 1.1 vil vi se, om spilleren balance kan blive negativ. Det vil vi nemlig ikke have sker, men i stedet, at spilleren balance bliver sat til 0, uanset hvor meget man trækker fra. Testen bestod, og vores kode holder dette krav. Vi laver derefter en positiv test på samme område, testrapport 1.2, hvor vi undersøger, om kontoen trækker penge fra balancen korrekt. Også denne test bestod.

I testrapport 2 tester vi, om spillerne, der ejer et felt, bliver fjernet som owners på den rigtige måde. Testen bestod.

I testrapport 3 tester vi terningernes sandsynlighed, da vi gerne vil have et sæt terninger, der slår med samme sandsynlighed som virkelige terninger ville rent teoretisk.

Refuge test

Refuge er et felt, som uddeler en bonus, når man lander på det. Vi tester derfor, om der bliver trukket point fra spillerens konto, når man indfører en refuge med negativ værdi, hvilket der naturligvis ikke må ske. Vi gennemgår denne JUnit-test grundigt, så fremgangsmåden står klart. Vi laver ikke en lige så uddybende tekst vedrørende de andre tests. For at se de andre tests, henviser vi til dokumentation for test i bilag 2.

Vi opretter testkoden i en "JUnit Test Case" i Eclipse. Her starter vi med at oprette tre forskellige refuges, samt vores spiller. Som en pre-condition beskriver vi vores objekter yderligere. Det første refuge giver 200 point og har placeringen 1. Næste refuge giver 0 point og har placeringen 2. Sidste refuge trækker 200 point fra og har placeringen 3. Vores testspiller er givet 1000 point. Meningen er, at vi placerer ham på de tre felter og ser, hvad der sker.

```
private Player Donald;
private Refuge refuge200;
private Refuge refuge0;
private Refuge refugeNeg200;

@Before
public void setUp() throws Exception {
    this.Donald = new Player(1000, "Donald Duck");
    this.refuge200 = new Refuge(1, 200);
    this.refuge0 = new Refuge(2, 0);
    this.refugeNeg200 = new Refuge(3, -200);
}
```

Figur 9: Screenshot fra vores Refuge test.

I den første test opretter vi en int kaldt "expected", som er den forventede værdi vi forventer at få efter koden er kørt. Vi tilføjer en anden int "actual" som er den faktiske værdi. Vi håber at disse stemmer overens. I testen med refuge200 forventer vi at spilleren har 1200 point, da refuge200 lægger 200 point til spillerens account. Vi tester først, om forholdene stemmer overens, og derefter om det forventede er lig det faktiske. Begge dele viste sig korrekte. Spilleren fik altså 200 point ved at lande på dette felt.

```
@Test //Tests if you get points by landing on a Refuge field
public void testLandOnField() {
    int expected = 1000;
    int actual = this.Donald.getAccount().getBalance();
    Assert.assertEquals(expected, actual);
    //Perform the action to be tested
    this.refuge200.landedOn(this.Donald);
    expected = 1000 + 200;
    actual = this.Donald.getAccount().getBalance();
    Assert.assertEquals(expected, actual);
}
```

Figur 10: Screenshot fra vores Refuge test.

Vi tester refuge200 igen, dog lander vores spiller på feltet to gange i træk. Også denne test bestod; spilleren fik, hvad der svarer til 1400 point.

I den anden test regner vi med, at spillerens beholdning er uberørt efter at have landet på refuge. Vores refuge har nemlig en værdi på 0, og giver derfor ingen bonus. Vi forventer derfor at "expected" er 1000, hvilket er lig spillerens nuværende beholdning. Denne test bestod også, og det samme, hvis spilleren lander på dette felt to gange i træk.

```
@Test //Tests what happens when you land on a Refuge that awards 0 points
public void testLandOnField0() {
    int expected = 1000;
    int actual = this.Donald.getAccount().getBalance();
    Assert.assertEquals(expected, actual);

    //Perform the action to be tested
    this.refuge0.landedOn(this.Donald);

    expected = 1000;
    actual = this.Donald.getAccount().getBalance();
    Assert.assertEquals(expected, actual);
}
```

Videre til den sidste test; her undersøger vi, om spilleren får fratrukket point ved at lande på refugeNeg200. Dette felt trækker nemlig 200 point fra spilleren, set ud fra feltets værdi. Vi forventer dog, som nævnt tidligere, at det ikke vil ske, eftersom en refuge kun kan give point. Vi forklarer, at "expected" er 1000, som er det samme som spillerens nuværende beholdning. Efter at have kørt testen kan vi se, at spilleren havde det forventede beløb på kontoen. Med andre ord, så bestod refuge denne test, og vi er nu sikre på, at der ikke kan blive trukket penge fra spilleren ved at lande på disse felter i spillet.

Konklusion

Dette projekt er forløbet rigtig godt, da vi er nået i mål med både rapport og kodning inden for den givne tidsramme. Vi har også opfyldt alle kravene fra kravspecifikation, både med kundens og vores egne krav til spillet. Koden blev skrevet i Eclipse med brug af GUI, hvor vi fik lavet et flot, velfungerende stykke kode. Vi har taget ved lære af de forrige projekter, og har været gode til at fordele arbejdet, skabe overblik og mødes i fritiden for at arbejde videre. UML-diagrammerne har været lettere at lave, sandsynligvis grundet øget erfaring, i forhold til forrige projekt. I dette projekt har vi gjort meget ud af at overholde GRASP, ved brug af mange klasser og klasser med underklasser, sådan at metoder og attributter er delt ud på en hensigtsmæssig måde. Dette er lykkedes med succes

Der har dog også været nogle bump på vejen, i form af bl.a. problemer med EGit. Flere af os har gentagne gange haft fejl med enten import eller eksport af projektet, hvilket har resulteret i meget tidsspild, og at vi har måtte lave et nyt repository og kopiere koden over til dette. Dette skete i slutningen af projektet, hvilket har medført at den første del af arbejdsprocessen i vores repository historie er gået tabt. Alt i alt er projektet blevet en succes, og lever op til vores forventninger.

Bilag

1. Kundens vision

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.

Kundens krav til 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

2. Dokumentation for test

I dette afsnit findes en række screenshots af vores tests.

```
1 package test;
2
3 import org.junit.After;
4 import org.junit.Assert;
5 import org.junit.Before;
6 import org.junit.Test;
7
8 import game.Player;
9
10 public class TestBalance {
11
12     private Player DonaldDuck;
13
14
15     // Preconditions
16     @Before
17     public void setUp() throws Exception {
18         this.DonaldDuck = new Player(3000, "Donald Duck");
19     }
20
21     @After
22     public void tearDown()  {
23         this.DonaldDuck = new Player(3000, "Donald Duck");
24     }
25
26     @Test
27     public void negativTest(){
28         // Performs the action to be tested
29         DonaldDuck.getAccount().addBalance(-4000);
30         int expected = 0;
31         int actual = DonaldDuck.getAccount().getBalance();
32         Assert.assertEquals(expected,actual);
33     }
34
35
36     @Test
37     public void positiveTest(){
38         // Performs the action to be tested
39         DonaldDuck.getAccount().addBalance(-1000);
40         int expected = 2000;
41         int actual = DonaldDuck.getAccount().getBalance();
42         Assert.assertEquals(expected, actual);
43     }
44 }
45
```

Figur 11: Negativ og positiv test af spillerens balance.

```
 1 package test;
 2
 3 import static org.junit.Assert.*;
11
12 public class TestOwner {
13
14     Player goofy;
15     AbstractField field;
16
17     @Before
18     public void setUp() throws Exception {
19         goofy = new Player(1000, "Goofy");
20         field = new Territory(1, 50, 500);
21         field.setOwner(goofy);
22     }
23
24     @Test
25     public void testOwner() {
26         Player expected = goofy;
27         Player actual = field.getOwner();
28         assertEquals(expected, actual); // tests if Goofy is the owner
29
30     }
31     @Test
32     public void testRemoveOwner() {
33         field.removeOwner();
34         Player expected = null;
35         Player actual = field.getOwner();
36         assertEquals(expected, actual); // tests if the owner is removed
37
38     }
39
40 }
41
```

Figur 12: Test for ejeren af felt.

```
 1 package test;
 2
 3 import org.junit.Assert;
 8
 9 public class DiceTest {
10
11     DiceCup dicecup;
12     // Index 1 and 2 will not be used throughout; because then the index
13     //number is the same as the dice number
14     double[] calProb = new double[13];
15     double[] theoreticalProb = {0, 0 ,1.0/36, 2.0/36, 3.0/36, 4.0/36,
16                                 5.0/36, 6.0/36, 5.0/36, 4.0/36, 3.0/36,
17                                 2.0/36, 1.0/36};;
18
19
20     @Before
21     public void setUp() throws Exception {
22         dicecup = new DiceCup();
23
24         /* This loop rolls the dice 1,000,000 times, and adds one millions
25         /* of a point to the corresponding array index in calProb[] after
26          * every roll */
27         for(int i=0;i<1000000;i++){
28             int value = dicecup.roll();
29             calProb[value]+=(1.0/1000000);
30         }
31
32         // This loop prints the calculated and expected probabilities for
33         // easy comparison
34                 for (int i=2; i<13; i++){
35                     System.out.println("C"+i+" = "+calProb[i]);
36                     System.out.println("T"+i+" = "+theoreticalProb[i]);
37                     System.out.println();
38                 }
39     }
40
41     @Test
42     public void test() {
43         for(int j=2; j<13; j++){
44             // Tests if probability for every instance matches with the
45             // theoretical probability, with an uncertainty of 0.001
46             Assert.assertEquals(theoreticalProb[j], calProb[j], 0.001);
47         }
48     }
49 }
50
```

Figur 13: Terningetest.

```java
package test;

import org.junit.*;

import game.Player;
import fields.Refuge;

public class RefugeTest {

    private Player Donald;
    private Refuge refuge200;
    private Refuge refuge0;
    private Refuge refugeNeg200;

    @Before
    public void setUp() throws Exception {
        this.Donald = new Player(1000, "Donald Duck");
        this.refuge200 = new Refuge(1, 200);
        this.refuge0 = new Refuge(2, 0);
        this.refugeNeg200 = new Refuge(3, -200);
    }

    @After
    public void tearDown() throws Exception {
        this.Donald = new Player(1000, "Donald Duck");
        //The fields are unaltered
    }

    @Test
    public void testEntities() {
        Assert.assertNotNull(this.Donald);

        Assert.assertNotNull(this.refuge200);
        Assert.assertNotNull(this.refuge0);
        Assert.assertNotNull(this.refugeNeg200);

        Assert.assertTrue(this.refuge200 instanceof Refuge);
        Assert.assertTrue(this.refuge0 instanceof Refuge);
        Assert.assertTrue(this.refugeNeg200 instanceof Refuge);
    }

    @Test //Tests if you get points by landing on a Refuge field
    public void testLandOnField() {
        int expected = 1000;
        int actual = this.Donald.getAccount().getBalance();
        Assert.assertEquals(expected, actual);
        //Perform the action to be tested
        this.refuge200.landedOn(this.Donald);
        expected = 1000 + 200;
        actual = this.Donald.getAccount().getBalance();
        Assert.assertEquals(expected, actual);
    }

    @Test //Tests if you get points by landing on a Refuge field twice in a row
    public void testLandOnFieldTwice() {
        int expected = 1000;
        int actual = this.Donald.getAccount().getBalance();
        Assert.assertEquals(expected, actual);

        //Perform the action to be tested
        this.refuge200.landedOn(this.Donald);
        this.refuge200.landedOn(this.Donald);
        expected = 1000 + 200 + 200;
        actual = this.Donald.getAccount().getBalance();
```

```
65          Assert.assertEquals(expected, actual);
66      }
67
68      @Test //Tests what happens when you land on a Refuge that awards 0 points
69      public void testLandOnField0() {
70          int expected = 1000;
71          int actual = this.Donald.getAccount().getBalance();
72          Assert.assertEquals(expected, actual);
73
74          //Perform the action to be tested
75          this.refuge0.landedOn(this.Donald);
76
77          expected = 1000;
78          actual = this.Donald.getAccount().getBalance();
79          Assert.assertEquals(expected, actual);
80      }
81
82      @Test //Tests what happens when you land on a Refuge that awards 0 points
83      public void testLandOnField0Twice() {
84          int expected = 1000;
85          int actual = this.Donald.getAccount().getBalance();
86          Assert.assertEquals(expected, actual);
87
88          //Perform the action to be tested
89          this.refuge0.landedOn(this.Donald);
90          this.refuge0.landedOn(this.Donald);
91
92          expected = 1000;
93          actual = this.Donald.getAccount().getBalance();
94          Assert.assertEquals(expected, actual);
95      }
96
97      @Test //Tests whether or not a negative amount of points can be awarded when
    you land on a Refuge
98      public void testLandOnFieldNeg() {
99          int expected = 1000;
100         int actual = this.Donald.getAccount().getBalance();
101         Assert.assertEquals(expected, actual);
102
103         //Perform the action to be tested
104         this.refugeNeg200.landedOn(this.Donald);
105
106         //It is not possible to deposit a negative amount
107         expected = 1000; //We expect balance to be unchanged
108         actual = this.Donald.getAccount().getBalance();
109         Assert.assertEquals(expected, actual);
110     }
111
112     @Test //Tests whether or not a negative amount of points can be awarded when
    you land on a Refuge twice in a row
113     public void testLandOnFieldNegTwice() {
114         int expected = 1000;
115         int actual = this.Donald.getAccount().getBalance();
116         Assert.assertEquals(expected, actual);
117
118         //Perform the action to be tested.
119         this.refugeNeg200.landedOn(this.Donald);
120         this.refugeNeg200.landedOn(this.Donald);
121
122         //It is still not possible to deposit a negative amount
123         expected = 1000; //We expect balance to be unchanged
124         actual = this.Donald.getAccount().getBalance();
125         Assert.assertEquals(expected, actual);
126     }
```

```java
package test;

import org.junit.*;
import game.Player;
import fields.LaborCamp;

public class LaborCampTest {

    private Player Donald;
    private Player Mickey;
    private LaborCamp LaborCamp;

    @Before
    public void setUp() throws Exception {
        this.Donald = new Player(5000, "Donald Duck");
        this.Mickey = new Player(2000, "Mickey Mouse");
        this.LaborCamp = new LaborCamp(1);
    }

    @After
    public void tearDown() throws Exception {
        this.Donald = new Player(5000, "Donald Duck");
        this.Mickey = new Player(2000, "Mickey Mouse");
        this.LaborCamp.removeOwner();
        //The fields are unaltered
    }

    @Test
    public void testEntities() {
        Assert.assertNotNull(this.Donald);
        Assert.assertNotNull(this.Mickey);
        Assert.assertNotNull(this.LaborCamp);
        Assert.assertTrue(this.LaborCamp instanceof LaborCamp);
    }

    @Test //Tests if a Fleet that IS owned actually IS owned
    public void testIfOwned() {

        //Set Mickey as owner
        this.LaborCamp.setOwner(this.Mickey);

        //Have Donald land on field that is owned
        this.LaborCamp.landedOn(this.Donald);

        Object expected = this.Mickey;
        Object actual = this.LaborCamp.getOwner();
        Assert.assertEquals(expected, actual);

    }
    @Test //Tests whether or not points are deducted from player that lands on a field whose owner owns ONE Labor Camp.
    public void testLandOnFieldOwned() {

        int actual = this.Donald.getAccount().getBalance();
        int expected = 5000;
        Assert.assertEquals(expected, actual);

        //Have Mickey own a LaborCamp
        this.LaborCamp.setOwner(Mickey);

        //Have Donald land on field not owned
        this.LaborCamp.landedOn(this.Donald);

        actual = this.Donald.getAccount().getBalance();
```

```
64          expected = 5000 - (this.Donald.getLastRoll() * 100 *
  this.Mickey.getNumLaborOwned());
65          Assert.assertEquals(expected, actual);
66
67      }
68
69      @Test //Tests whether or not points are deducted from player that lands on a
  field whose owner owns ONE Labor Camp. Twice in a row.
70      public void testLandOnFieldOwnedTwice() {
71
72          int expected = 5000;
73          int actual = this.Donald.getAccount().getBalance();
74          Assert.assertEquals(expected, actual);
75
76          //Have Mickey own a LaborCamp
77          this.LaborCamp.setOwner(Mickey);
78
79          //Have Donald land on field not owned
80          this.LaborCamp.landedOn(this.Donald);
81          int firstroll = this.Donald.getLastRoll();
82          this.LaborCamp.landedOn(this.Donald);
83
84          actual = this.Donald.getAccount().getBalance();
85          expected = 5000 - (this.Donald.getLastRoll() * 100 *
  this.Mickey.getNumLaborOwned()) - (firstroll * 100 *
  this.Mickey.getNumLaborOwned());
86          Assert.assertEquals(expected, actual);
87
88      }
89
90      @Test //Tests whether or not points are deducted from player that lands on a
  field whose owner owns TWO Labor Camps.
91      public void testLandOnFieldOwnerOwnsTwo() {
92
93          int expected = 5000;
94          int actual = this.Donald.getAccount().getBalance();
95          Assert.assertEquals(expected, actual);
96
97          //Have Mickey own a LaborCamp
98          this.LaborCamp.setOwner(Mickey);
99          this.Mickey.setNumFleetOwned(2);
100
101         //Have Donald land on a field owned
102         this.LaborCamp.landedOn(this.Donald);
103
104
105         actual = this.Donald.getAccount().getBalance();
106         expected = 5000 - (this.Donald.getLastRoll() * 100 *
  this.Mickey.getNumLaborOwned());
107         Assert.assertEquals(expected, actual);
108
109     }
110
111     @Test //Tests whether or not points are paid to owner who owns ONE Labor Camp
  when another player lands on his field.
112     public void testGetMoneyOnFieldOwned() {
113
114         int actual = this.Mickey.getAccount().getBalance();
115         int expected = 2000;
116         Assert.assertEquals(expected, actual);
117
118         //Have Mickey own a LaborCamp
119         this.LaborCamp.setOwner(Mickey);
120
```

```
121         //Have Donald land on field not owned
122         this.LaborCamp.landedOn(this.Donald);
123 
124         actual = this.Mickey.getAccount().getBalance();
125         expected = 2000 + (this.Donald.getLastRoll() * 100 *
    this.Mickey.getNumLaborOwned());
126         Assert.assertEquals(expected, actual);
127 
128     }
129 
130     @Test //Tests whether or not points are paid to owner who owns TWO Labor Camps
    when another player lands on one of his fields.
131     public void testGetMoneyOnTwoFieldsOwned() {
132 
133         int actual = this.Mickey.getAccount().getBalance();
134         int expected = 2000;
135         Assert.assertEquals(expected, actual);
136 
137         //Have Mickey own a LaborCamp.
138         this.LaborCamp.setOwner(Mickey);
139         this.Mickey.setNumFleetOwned(2);
140 
141         //Have Donald land on this field
142         this.LaborCamp.landedOn(this.Donald);
143 
144         actual = this.Mickey.getAccount().getBalance();
145         expected = 2000 + (this.Donald.getLastRoll() * 100 *
    this.Mickey.getNumLaborOwned());
146         Assert.assertEquals(expected, actual);
147 
148     }
149 
150 }
```

Figur 14: Test for LaborCamp.

```java
package test;

import org.junit.*;
import game.Player;
import fields.Fleet;

public class FleetTest { //*** IN PROGRESS ***

    private Player Donald;
    private Player Mickey;
    private Fleet Fleet;
    private Fleet Fleet2;
    private Fleet Fleet3;
    private Fleet Fleet4;

    @Before
    public void setUp() throws Exception {
        this.Donald = new Player(5000, "Donald Duck");
        this.Mickey = new Player(2000, "Mickey Mouse");
        this.Fleet = new Fleet(1);
        this.Fleet2 = new Fleet(2);
        this.Fleet3 = new Fleet(3);
        this.Fleet4 = new Fleet(4);
    }

    @After
    public void tearDown() throws Exception {
        this.Donald = new Player(5000, "Donald Duck");
        this.Mickey = new Player(2000, "Mickey Mouse");
        this.Fleet.removeOwner();
        this.Fleet2.removeOwner();
        this.Fleet3.removeOwner();
        this.Fleet4.removeOwner();
        //The fields are unaltered
    }

    @Test
    public void testEntities() {
        Assert.assertNotNull(this.Donald);
        Assert.assertNotNull(this.Fleet);
        Assert.assertTrue(this.Fleet instanceof Fleet);
    }

    @Test //Tests if a Fleet that is NOT owned actually is NOT owned
    public void testLandOnFieldNotOwned() {

        // Since no player has been set to own this field, we expect owner to be
  null

        Object expected = null;
        Object actual = this.Fleet.getOwner();
        Assert.assertEquals(expected, actual);

    }

    @Test //Tests if a Fleet that IS owned actually IS owned
    public void testLandOnFieldOwned() {

        //Set Mickey as owner
        this.Fleet.setOwner(this.Mickey);

        //Have Donald land on field that is owned
        this.Fleet.landedOn(this.Donald);

```

```
64          Object expected = this.Mickey;
65          Object actual = this.Fleet.getOwner();
66          Assert.assertEquals(expected, actual);
67
68      }
69
70      @Test //Tests whether or not points are deducted from a player landing on a
    field that is already owned
71      public void testLandOnField() {
72          int expected = 5000;
73          int actual = this.Donald.getAccount().getBalance();
74          Assert.assertEquals(expected, actual);
75
76          //Perform the action to be tested
77
78          //First set Mickey as owner
79          this.Fleet.setOwner(this.Mickey);
80
81          //Then have Donald land on this field
82          this.Fleet.landedOn(this.Donald);
83          expected = 5000 - 500;
84          actual = this.Donald.getAccount().getBalance();
85          Assert.assertEquals(expected, actual);
86      }
87
88      @Test //Tests whether or not points are deducted from a player landing on a
    field that is already owned. Twice in a row.
89      public void testLandOnFieldTwice() {
90          int expected = 5000;
91          int actual = this.Donald.getAccount().getBalance();
92          Assert.assertEquals(expected, actual);
93
94          //Perform the action to be tested
95
96          //First set Mickey as owner
97          this.Fleet.setOwner(this.Mickey);
98
99          //Then have Donald land on this field (TWICE IN A ROW! AWWW MAN THATS BAD
    LUCK!)
100         this.Fleet.landedOn(this.Donald);
101         this.Fleet.landedOn(this.Donald);
102         expected = 5000 - 500 - 500;
103         actual = this.Donald.getAccount().getBalance();
104         Assert.assertEquals(expected, actual);
105     }
106
107     @Test //Tests whether or not points are awarded to owner of field after another
    player lands on this field.
108     public void testGetPointsField() {
109         int expected = 2000;
110         int actual = this.Mickey.getAccount().getBalance();
111         Assert.assertEquals(expected, actual);
112
113         //Perform the action to be tested
114
115         //First set Mickey as owner
116         this.Fleet.setOwner(this.Mickey);
117
118         //Then have Donald land on this field
119         this.Fleet.landedOn(this.Donald);
120         expected = 2000 + 500;
121         actual = this.Mickey.getAccount().getBalance();
122         Assert.assertEquals(expected, actual);
123     }
```

```
124
125     @Test //Tests whether or not points are awarded to owner of field after another
    player lands on this field. Twice in a row.
126     public void testGetPointsFieldTwice() {
127         int expected = 2000;
128         int actual = this.Mickey.getAccount().getBalance();
129         Assert.assertEquals(expected, actual);
130 
131         //Perform the action to be tested.
132 
133         //First set Mickey as owner
134         this.Fleet.setOwner(this.Mickey);
135 
136         //Then have Donald land on this field (TWICE IN A ROW! AWWW MAN THATS BAD
    LUCK!)
137         this.Fleet.landedOn(this.Donald);
138         this.Fleet.landedOn(this.Donald);
139 
140         expected = 2000 + 500 + 500;
141         actual = this.Mickey.getAccount().getBalance();
142         Assert.assertEquals(expected, actual);
143     }
144 
145     @Test //Tests whether or not points are deducted from player that lands on a
    field whose owner owns TWO Fleets.
146     public void testLandOnFieldOwnerOwns2() {
147         int expected = 5000;
148         int actual = this.Donald.getAccount().getBalance();
149         Assert.assertEquals(expected, actual);
150 
151         //Perform the action to be tested
152 
153         //First set Mickey as owner
154         this.Fleet.setOwner(this.Mickey);
155         this.Mickey.setNumFleetOwned(2);
156 
157         //Then have Donald land on one of the fields
158         this.Fleet.landedOn(this.Donald);
159 
160         expected = 5000 - 1000;
161         actual = this.Donald.getAccount().getBalance();
162         Assert.assertEquals(expected, actual);
163     }
164 
165     @Test //Tests whether or not points are deducted from player that lands on a
    field whose owner owns THREE Fleets.
166     public void testLandOnFieldOwnerOwns3() {
167         int expected = 5000;
168         int actual = this.Donald.getAccount().getBalance();
169         Assert.assertEquals(expected, actual);
170 
171         //Perform the action to be tested
172 
173         //First set Mickey as owner
174         this.Fleet.setOwner(this.Mickey);
175         this.Mickey.setNumFleetOwned(3);
176 
177         //Then have Donald land on one of the fields
178         this.Fleet.landedOn(this.Donald);
179 
180         expected = 5000 - 2000;
181         actual = this.Donald.getAccount().getBalance();
182         Assert.assertEquals(expected, actual);
183     }
```

```
184
185     @Test //Tests whether or not points are deducted from player that lands on a
    field whose owner owns FOUR Fleets.
186     public void testLandOnFieldOwnerOwns4() {
187         int expected = 5000;
188         int actual = this.Donald.getAccount().getBalance();
189         Assert.assertEquals(expected, actual);
190
191         //Perform the action to be tested
192
193         //First set Mickey as owner
194         this.Fleet.setOwner(this.Mickey);
195         this.Mickey.setNumFleetOwned(4);
196
197         //Then have Donald land on one of the fields
198         this.Fleet.landedOn(this.Donald);
199
200         expected = 5000 - 4000;
201         actual = this.Donald.getAccount().getBalance();
202         Assert.assertEquals(expected, actual);
203     }
204
205     @Test //Tests whether or not points are paid to owner who owns TWO Fleets when
    a player lands on his field.
206     public void testGetMoneyOnFieldOwnerOwns2() {
207         int expected = 2000;
208         int actual = this.Mickey.getAccount().getBalance();
209         Assert.assertEquals(expected, actual);
210
211         //Perform the action to be tested
212
213         //First set Mickey as owner
214         this.Fleet.setOwner(this.Mickey);
215         this.Mickey.setNumFleetOwned(2);
216
217         //Then have Donald land on one of the fields
218         this.Fleet.landedOn(this.Donald);
219
220         expected = 2000 + 1000;
221         actual = this.Mickey.getAccount().getBalance();
222         Assert.assertEquals(expected, actual);
223     }
224
225     @Test //Tests whether or not points are paid to owner who owns THREE Fleets
    when a player lands on his field.
226     public void testGetMoneyOnFieldOwnerOwns3() {
227         int expected = 2000;
228         int actual = this.Mickey.getAccount().getBalance();
229         Assert.assertEquals(expected, actual);
230
231         //Perform the action to be tested
232
233         //First set Mickey as owner
234         this.Fleet.setOwner(this.Mickey);
235         this.Mickey.setNumFleetOwned(3);
236
237         //Then have Donald land on one of the fields
238         this.Fleet.landedOn(this.Donald);
239
240         expected = 2000 + 2000;
241         actual = this.Mickey.getAccount().getBalance();
242         Assert.assertEquals(expected, actual);
243     }
244
```

```
245     @Test //Tests whether or not points are paid to owner who owns FOUR Fleets when
    a player lands on his field.
246     public void testGetMoneyOnFieldOwnerOwns4() {
247         int expected = 2000;
248         int actual = this.Mickey.getAccount().getBalance();
249         Assert.assertEquals(expected, actual);
250 
251         //Perform the action to be tested
252 
253         //First set Mickey as owner
254         this.Fleet.setOwner(this.Mickey);
255         this.Mickey.setNumFleetOwned(4);
256 
257         //Then have Donald land on one of the fields
258         this.Fleet.landedOn(this.Donald);
259 
260         expected = 2000 + 4000;
261         actual = this.Mickey.getAccount().getBalance();
262         Assert.assertEquals(expected, actual);
263     }
264 
265 }
266 
```

Figur 15: Fleet test.

```
 1 package test;
 2
 3 import static org.junit.Assert.*;
18
19 public class TaxTest {
20
21     private Player minieMouse;
22     private Tax tax3;
23     private Tax tax13;
24
25     @Before
26     public void setUp() throws Exception {
27         this.minieMouse = new Player(10000,"Mini");
28         this.tax3 = new Tax(3);
29         this.tax13 = new Tax(13);
30     }
31
32     @After
33     public void tearDown() throws Exception {
34
35     }
36
37     @Test
38     public void testEntities() {
39         Assert.assertNotNull(this.minieMouse);
40
41         Assert.assertNotNull(this.tax3);
42         Assert.assertNotNull(this.tax13);
43
44         Assert.assertTrue(this.tax3 instanceof Tax);
45         Assert.assertTrue(this.tax13 instanceof Tax);
46     }
47
48     @Test
49     public void testPosition3() {
50         // Performs the action to be tested
51         this.minieMouse.setPosition(3);
52
53         int expected = 3;
54         int actual = minieMouse.getPosition();
55         assertEquals(expected, actual);
56     }
57
58     @Test
59     public void testPosition13() {
60         // Performs the action to be tested
61         this.minieMouse.setPosition(13);
62
63         int expected = 13;
64         int actual = minieMouse.getPosition();
65         assertEquals(expected, actual);
66     }
67
68     @Test
69     public void testTaxAdded3() {
70         int expected = 10000;
71         int actual = this.minieMouse.getAccount().getBalance();
72         assertEquals(expected, actual);
73
74         // Performs the action to be tested
75         this.minieMouse.setPosition(3);
76         this.tax3.landedOn(minieMouse);
77
78         expected = 10000 - 2000;
```

```
79          actual = minieMouse.getAccount().getBalance();
80          assertEquals(expected, actual);
81      }
82
83      @Test
84      public void testTax13_10procent() {
85          int expected = 10000;
86          int actual = this.minieMouse.getAccount().getBalance();
87          assertEquals(expected, actual);
88
89          // Performs the action to be tested
90          this.tax13.pay10procent(minieMouse);
91
92          expected = 10000 - 1000;
93          actual = minieMouse.getAccount().getBalance();
94          assertEquals(expected, actual);
95      }
96
97      @Test
98      public void testTax13_4000() {
99          int expected = 10000;
100         int actual = this.minieMouse.getAccount().getBalance();
101         assertEquals(expected, actual);
102
103         // Performs the action to be tested.
104         this.tax13.pay4000(minieMouse);
105
106
107         expected = 10000 - 4000;
108         actual = minieMouse.getAccount().getBalance();
109         assertEquals(expected, actual);
110     }
111
112 }
113
```

Figur 16: Tax test.

```java
package test;

import org.junit.*;

public class TerritoryTest {

    private Player Donald;
    private Player Mickey;
    private Territory Territory200;
    private Territory Territory0;
    private Territory TerritoryNeg200;
    private Territory TerritoryOwned;
    private Territory TerritoryNotOwned;

    @Before
    public void setUp() throws Exception {
        this.Donald = new Player(1000, "Donald Duck");
        this.Mickey = new Player(2000, "Mickey Mouse");
        this.Territory200 = new Territory(1, 200, 400);
        this.Territory0 = new Territory(2, 0, 0);
        this.TerritoryNeg200 = new Territory(3, -100, -200);
        this.TerritoryOwned = new Territory(3, -100, -200);
        this.TerritoryNotOwned = new Territory(3, -100, -200);
    }

    @After
    public void tearDown() throws Exception {
        this.Donald = new Player(1000, "Donald Duck");
        this.Mickey = new Player(2000, "Mickey Mouse");
        //The fields are unaltered
    }

    @Test
    public void testEntities() {
        Assert.assertNotNull(this.Donald);

        Assert.assertNotNull(this.Territory200);
        Assert.assertNotNull(this.Territory0);
        Assert.assertNotNull(this.TerritoryNeg200);
        Assert.assertNotNull(this.TerritoryOwned);
        Assert.assertNotNull(this.TerritoryNotOwned);

        Assert.assertTrue(this.Territory200 instanceof Territory);
        Assert.assertTrue(this.Territory0 instanceof Territory);
        Assert.assertTrue(this.TerritoryNeg200 instanceof Territory);
        Assert.assertTrue(this.TerritoryOwned instanceof Territory);
        Assert.assertTrue(this.TerritoryNotOwned instanceof Territory);
    }

    @Test //Tests if a territory that is NOT owned actually is NOT owned
    public void testLandOnFieldNotOwned() {

        //Have Donald land on field not owned
        this.TerritoryNotOwned.landedOn(this.Donald);

        Object expected = null;
        Object actual = this.TerritoryNotOwned.getOwner();
        Assert.assertEquals(expected, actual);

    }

    @Test //Tests if a territory that IS owned actually IS owned
    public void testLandOnFieldOwned() {

```

```
67          //Set Mickey as owner
68          this.TerritoryOwned.setOwner(this.Mickey);
69
70          //Have Donald land on field that is owned
71          this.TerritoryOwned.landedOn(this.Donald);
72
73          Object expected = this.Mickey;
74          Object actual = this.TerritoryOwned.getOwner();
75          Assert.assertEquals(expected, actual);
76
77      }
78
79      @Test //Tests whether or not points are deducted from a player landing on a
    field that is already owned
80      public void testLandOnField() {
81          int expected = 1000;
82          int actual = this.Donald.getAccount().getBalance();
83          Assert.assertEquals(expected, actual);
84
85          //Perform the action to be tested
86
87          //First set Mickey as owner
88          this.Territory200.setOwner(this.Mickey);
89
90          //Then have Donald land on this field
91          this.Territory200.landedOn(this.Donald);
92          expected = 1000 - 200;
93          actual = this.Donald.getAccount().getBalance();
94          Assert.assertEquals(expected, actual);
95      }
96
97      @Test //Tests whether or not points are deducted from a player landing on a
    field that is already owned. Twice in a row.
98      public void testLandOnFieldTwice() {
99          int expected = 1000;
100         int actual = this.Donald.getAccount().getBalance();
101         Assert.assertEquals(expected, actual);
102
103         //Perform the action to be tested
104
105         //First set Mickey as owner
106         this.Territory200.setOwner(this.Mickey);
107
108         //Then have Donald land on this field (TWICE IN A ROW! AWWW MAN THATS BAD
    LUCK!)
109         this.Territory200.landedOn(this.Donald);
110         this.Territory200.landedOn(this.Donald);
111         expected = 1000 - 200 - 200;
112         actual = this.Donald.getAccount().getBalance();
113         Assert.assertEquals(expected, actual);
114     }
115
116     @Test //Tests whether or not points are awarded to owner of field after another
    player lands on this field.
117     public void testGetPointsField() {
118         int expected = 2000;
119         int actual = this.Mickey.getAccount().getBalance();
120         Assert.assertEquals(expected, actual);
121
122         //Perform the action to be tested
123
124         //First set Mickey as owner
125         this.Territory200.setOwner(this.Mickey);
126
```

```
127         //Then have Donald land on this field
128         this.Territory200.landedOn(this.Donald);
129         expected = 2000 + 200;
130         actual = this.Mickey.getAccount().getBalance();
131         Assert.assertEquals(expected, actual);
132     }
133
134     @Test //Tests whether or not points are awarded to owner of field after another
    player lands on this field. Twice in a row.
135     public void testGetPointsFieldTwice() {
136         int expected = 2000;
137         int actual = this.Mickey.getAccount().getBalance();
138         Assert.assertEquals(expected, actual);
139
140         //Perform the action to be tested
141
142         //First set Mickey as owner
143         this.Territory200.setOwner(this.Mickey);
144
145         //Then have Donald land on this field (TWICE IN A ROW! AWWW MAN THATS BAD
    LUCK!)
146         this.Territory200.landedOn(this.Donald);
147         this.Territory200.landedOn(this.Donald);
148
149         expected = 2000 + 200 + 200;
150         actual = this.Mickey.getAccount().getBalance();
151         Assert.assertEquals(expected, actual);
152     }
153
154     @Test //Tests what happens when you land on a field Territory that IS owned and
    doesn't have rent
155     public void testLandOnField0() {
156         int expected = 1000;
157         int actual = this.Donald.getAccount().getBalance();
158         Assert.assertEquals(expected, actual);
159
160         //Perform the action to be tested
161
162         //First set Mickey as owner
163         this.Territory0.setOwner(this.Mickey);
164
165         //Then have Donald land on this field
166         this.Territory0.landedOn(this.Donald);
167
168         expected = 1000; //We expect this unchanged
169         actual = this.Donald.getAccount().getBalance();
170         Assert.assertEquals(expected, actual);
171     }
172
173     @Test //Tests what happens when you land on a field Territory that IS owned and
    doesn't have rent. Twice in a row.
174     public void testLandOnField0Twice() {
175         int expected = 1000;
176         int actual = this.Donald.getAccount().getBalance();
177         Assert.assertEquals(expected, actual);
178
179         //Perform the action to be tested
180
181         //First set Mickey as owner
182         this.Territory0.setOwner(this.Mickey);
183
184         //Then have Donald land on this field (TWICE IN A ROW! NOT TOO BAD THO,
    RENT IS 0)
185         this.Territory0.landedOn(this.Donald);
```

```
186         this.Territory0.landedOn(this.Donald);
187
188         expected = 1000;
189         actual = this.Donald.getAccount().getBalance();
190         Assert.assertEquals(expected, actual);
191     }
192
193     @Test //Tests what happens when you land on a field Territory that IS owned and
    has negative rent
194     public void testLandOnFieldNeg() {
195         int expected = 1000;
196         int actual = this.Donald.getAccount().getBalance();
197         Assert.assertEquals(expected, actual);
198
199         //Perform the action to be tested
200
201         //First set Mickey as owner
202         this.TerritoryNeg200.setOwner(this.Mickey);
203
204         //Then have Donald land on this field
205         this.TerritoryNeg200.landedOn(this.Donald);
206
207         //It is not possible to deposit a negative amount? **WHAT IS ACTUALLY GOING
    ON HERE?**
208         expected = 1000;
209         actual = this.Donald.getAccount().getBalance();
210         Assert.assertEquals(expected, actual);
211     }
212
213     @Test //Tests what happens when you land on a field Territory that IS owned and
    has negative rent. Twice in a row.
214     public void testLandOnFieldNegTwice() {
215         int expected = 1000;
216         int actual = this.Donald.getAccount().getBalance();
217         Assert.assertEquals(expected, actual);
218
219         //Perform the action to be tested
220
221         //First set Mickey as owner.
222         this.TerritoryNeg200.setOwner(this.Mickey);
223
224         //Then have Donald land on this field (TWICE IN A ROW! NOT TOO BAD THO,
    NOTHING SHOULD HAPPEN RIGHT?)
225         this.TerritoryNeg200.landedOn(this.Donald);
226         this.TerritoryNeg200.landedOn(this.Donald);
227
228         //It is not possible to deposit a negative amount? **WHAT IS ACTUALLY GOING
    ON HERE?**
229         expected = 1000;
230         actual = this.Donald.getAccount().getBalance();
231         Assert.assertEquals(expected, actual);
232     }
233 }
234
```

Figur 17: Test for territory.