Projektopgave efterår 2013 - jan 2014

02312-14 Indledende programmering og 02313 Udviklingsmetoder til IT-Systemer

Projektnavn: del3 Gruppe nr: 19 Afleveringsfrist: mandag den 02/12 2013 Kl. 5:00

Denne rapport er afleveret via Campusnet (der skrives ikke under)

Denne rapport indeholder 77 sider incl. denne side

Studie nr, Efternavn, Fornavne

s110795, Mortensen, Thomas Martin
Kontakt person (Projektleder)

s113577, Johansen, Chris Dons

s123897, Ahlgreen, Thomas Kamper

Timeregnskab

Dato	Deltager	Design	Implementering	Test	Dok.	Andet	I alt
15/11-13	Thomas Mortensen	2					2
15/11-13	Chris Johansen	2					2
15/11-13	Thomas Ahlgreen	2					2
22/11-13	Thomas Mortensen		2,5				2,5
22/11-13	Chris Johansen		2,5				2,5
22/11-13	Thomas Ahlgren		2,5				2,5
29/11-13	Thomas Mortensen		2,5				2,5
29/11-13	Chris Johansen		2,5				2,5
29/11-13	Thomas Ahlgren		2,5				2,5
2/12-13	Thomas Mortensen			1	6		7
2/12-13	Chris Johansen			1	6		7
2/12-13	Thomas Ahlgren			1	6		7
diverse	Thomas Mortensen				4		
diverse	Chris Johansen					2	
diverse	Thomas Ahlgren						
							48

Indhold

1 Indledning

Spilfirmaet IOOuterActive har givet os endnu en opgave.

Kundens vision er en udvidelse af vores allerede udviklede programmer fra [1913a] og [1913b]

Denne gang ønsker kunden at lave flere forskellige typer felter. Disse felter skal implementeres på en "rigtig"spilleplade, hvor man kan gå i ring. Samtidig ønskes der mulighed for 2-6 spillere. Spillerens konto skal sættes med et fast beløb fra start, og spillet skal slutte, når en spiller er gået bankerot.

Projektet forventes at have elementer fra både **FURPS+** og **GRASP**. Herudover er der lavet specifikke krav til hvilke artefakter, der skal ingå i krav, analyse, kode, test, og designdokumentation.

OBS: alle gruppemedlemmer er lige ansvarlige for alle dele af vores dokumentation

2 Kravspecificering og Use cases

I dette afsnit vil vi beskrive vores kravspecificering og vores Use cases, som vi har udarbejdet til denne rapport.

2.1 Kravspecificering

I dette afsnit har vi taget udgangspunkt i kundens vision. Vi har læst den igennen, og stillet spørgsmålstegn, ved de ting, der kunne være tvivl om. Kunden er dog blevet til kravspecificering i forløbet, og derfor var der få tvivlsspørgsmål.

1. Hvad skal der ske, hvis en spiller ikke kan betale det han skal?
2. Skal man betale, hvis man lander på et felt, der er ejet af en bankerot spiller?
3. Skal man have mulighed for at købe eet felt, der tidliger var ejet af en spiller, der nu er gået bankerot?

Vores kontakt i firmaet, som er vores projektleder, har besvaret disse punkter på følgende måde

1. Spilleren der ikke kan betale, må betale det han har, og går herefter bankerot.
2. Feltet er ikke længere ejet af nogen, og derfor skal der heller ikke betales.
3. Da feltet ikke længere ejes, skal man kunne købe det.

2.2 Fully Dressed Use Case

Vi har i dette afsnit beskrevet vores use case med en fully dressed use case for Fleet.

Use Case

Fleet

Scope

Spil

Level

Bruger Konsekvens

Primary actor

Spiller

Stakeholders and Interests

Spiller: Vil købe felt

Preconditions

Eclipse er installeret på maskinen.
Programmet er kørende.

Main Success Scenarie:

1 Spiller starter spillet

2 Spiller indtaster Navn

3 Spiller Ruller med terningerne

4 Spiller lander på Fleet

5 Spiller køber Fleet

6 System checker pengebeholdning

7 System trækker penge

8 System giver turen til næste spiller

Extensions Alternative scenarier:

*a - til hvert et tidspunkt

1. Spiller lukker spillet ved at taste "q"

5a. Spiller skal betale penge

1. Spiller betaler regning

5b.

1. Spiller kan ikke betale
2. System giver turen videre

Specielle Krav:

- Skal kunne køre på en Windows maskine med Java EE på DTU's computere
- Skal kunne spilles af en almindelig bruger

2.3 Refuge Brief

Når en spiller lander på refuge, skal spilleren (afhængigt af om de lander på Walled City eller Monastery) modtage hhv. 5000 eller 500.

2.4 Fleet Test Brief

En udvikler sætter testen igang. Når testen er igang tester den hvor meget man skal betale ud fra flere scenarier.

2.5 Refuge Test Brief

Denne test, tester om man faktisk får penge når man lander på feltet. Den tester også om hvorvidt der returneres forventede beløb.

3 FURPS+

I **UP** bruger man **FURPS+**, der er udviklet af Hewlett-Packard til at kategoriserer kravene til ens system. "+"i **FURPS+** kom i følge [Wik13] til senere, efter HP ønskede at dække flere kategorier med denne model. Vi har beskrevet kort, hvad der generelt hører under de forskellige punkter i **FURPS+**, og efterfølgende listet de funde krav i den aktuelle opgave op.

3.1 Functional

Hvilke funktioner skal systemt have. Hvad skal det kunne. Sikkerhed.

Vi har taget udgangspunkt i kundens [IOO13] opgave. Ud fra deres vision, bilag og andre punkter i rapporten har vi fundet frem til følgende punkter:

- En udbyggelse af forrige system med forskellige felttyper.
- En spilleplade, hvor spillerne skal kunne lande på et felt og fortsætte på næste slag. Samtidig skal man kunne gå i ring på brættet.
- Man skal kunne vælge mellem 2-6 spillere.
- Et *Territory* skal kunne købes af en spiller, når han lander på feltet. Feltet skalhave en fast leje. Jo højere feltnummer jo højere pris og leje.
- Et *Refuge* skal give en bonus på enten 5000 eller 500 afhængigt af feltnummeret, når en spiller lander på denne type felt.
- Når man lander på et *Labor Camp* felt, skal man betale 100 gange værdien af et nyt terningeslag. Dette beløb skal i øvrigt ganges med antallet af *Labor Camps* med den samme ejer.
- Der er to felter af typen *Tax*. Det ene felt betaler man et fast beløb af Kr. 2000,-, når man lander på feltet. Det andet skal man selv vælge om man vil betale et fast beløb af Kr. 4000,- eller om man vil betale 10% af hele sin formue. (Hvilket betyder, både af hvad der står på kontoen og af felter man ejer).
- Lander man på et felt af typen *Fleet*, bestemmes beløbets størrelse ud fra hvor mange *Fleet* felter, der har samme ejer.

Sikkerhed er et punkt, vi ikke kan finde noget om, så vi må gå ud fra firmaet selv sørger for dette.

3.2 Usability

Menneskelige faktorer, skal man bruge hjælp. Dokumentation af systemet i brug.

Vi går ud fra, at kravene til **Usability** er de samme som i foregående opgaver. Det er kun dokumentation, der er nævnt i denne opgave. Kravene fra sidst, var at det skulle kunne bruges af en normal person med en 9. klasses eksamen. Der må ikke være brug for hjælp. Dog må man godt forklare hvad spillet går ud på.

Til dokumentation forventes der **Designsekvensdiagrammer**, der viser den dynamiske tilgang til programmet.

3.3 Reliability

Hyppigheden af fejl, kan det nemt gendannes og forudsigelighed.

For at formindske hyppigheden af fejl, har vi løbende brugt User Test, for at fange semantiske fejl. Herudover har IOOuterActive stillet en skabelon til**Junit** tests, der gerne skulle fange fejl i vores `Field` klasser. Derudover er der ikke nogle målbare krav til **Reliability**.

3.4 Performance

Responstider, informationer i systemts gennemløb, præcision, tilgængelighed af systemet og hvor mange ressourcer må det bruge.

Vi går ud fra tilgængeligheden er det samme som i de andre opgaver. Her var kravet, at programmet skulle kunne køre på maskinerne i **DTU**'s databarer.

3.5 Supportability

Ændringer, vedligeholdelse, internationalisering og konfigurationsmuligheder.

Her anbefales det at vi så vidt muligt overholder **GRASP** patterns og samtidig benytter os af **FURPS+**. Dette skulle gerne gøre programmet nemmer i forhold til alle punkter i **Supportability**.

Her kommer de underfaktorer, som +'et repræsenterer

3.6 Implementation

Ressource begrænsninger, sprog, værktøjer og hardware

Her er igen ikke nogle direkte krav. Vi går ud fra, at afleveringsproceduren er den samme som i foregående opgaver. Det vil betyde at projektet skal afleveres som et **Eclipse** projekt

3.7 Interface

Begrænsninger fremkommet af at skulle interagere med grænseflader fra eksterne systemer.

Den eneste begrænsning vi har er på den udleverede **GUI**, da vi ikke selv har udviklet denne. Vi kan derfor benytte os af allerede implementerede funktioner i denne.

3.8 Operations

Sporing og kontrollering af ændringer i softwaren.

Det er ikke noget vi skal tage stilling til i dette projekt.

3.9 Packaging

Hvordan skal programmet leveres (Fysisk eller fil)? Hvor mange installationer er der.

Igen går vi ud fra tidligere opgaver, hvor vi skulle aflevere programmet elektronisk, som et pakket **Eclipse** projekt også indeholdende vores rapport. Vi skal ikke installere det for kunden.

3.10 Legal

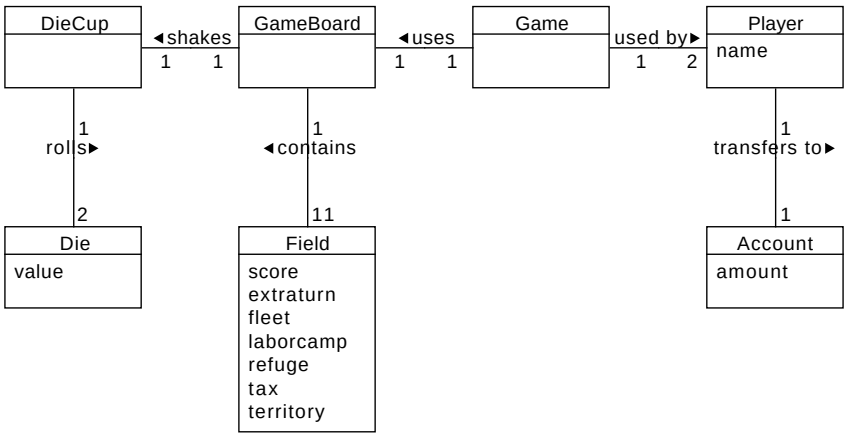
Licensering med videre.

Det er ikke noget vi er gjort opmærksom på af IOOuterActive. Det eneste vi kan nævne i forhold til retsmæssige stridigheder er, at vi skal tillade brugen af vores materiale, hvis dette ønskes benyttet til f.eks. undervisningssituationer.

4 Domænemodel

Vi har i denne omgang valgt at modificere på vores **Domæne Model** fra [1913b], da denne allerede fungerede ret godt. Vi har dog måttet tilføje de nye domæner, som IOOuterActive har leveret i form af deres vision.

Vi kan se at det eneste ekstra domæne vi har tilføjet er *Field*. Vi kunne have valgt at lave domæne klasser til vores forskellige felttyper også, men på den måde ville man hurtigt miste overblikket. Vi vurderede at det vigtigste var at vores kunde også ville kunne forstå modellen. Herudover har vi flyttet lidt rund på relationerne mellem domænerne. Før var det *Game*, der kendte til *DieCup*, men i denne omgang valgte vi at sige, det var mere logisk at man brugte raflebægeret på spillebrættet, hvilket nok er mere tro mod virkeligheden. Dette ses i figur 1 on the following page



Figur 1: Domæne Model

5 BCE model

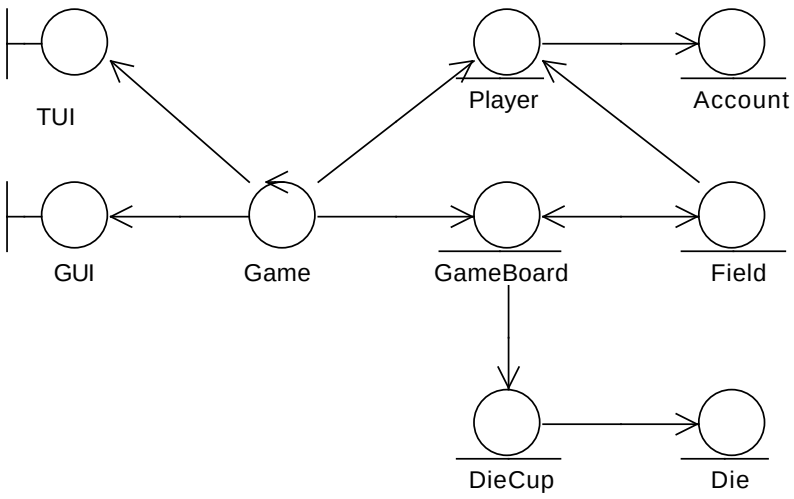
I dette, har vi opfyldt kravet omkring en BCE model til at dokumentere og skabe oveblik over vores kendskab i koden.

Vi har igen brugt `Game` som vores *Controller*, der uddeler ansvaret til de andre klasser. Ud over det er det de samme *Entities* og *Boundaries*, som i [1913b].

I forhold til en optimal **BCE model**, har måttet ændre lidt i dens kendskaber. Efter at have sammenlignet koden med vores udgangspunkt, har vi måttet lave kendskab begge veje mellem `Gameboard` og `Field`. Samtidig fik vi et krav i opgaven, at `Field` skulle kunne kalde objekter af `Player`. Derfor har vi også måttet lave et kendskab der.

Kendskabet til `DieCup` har vi flyttet over i `GameBoard`, for at få det til at passe med vores nye **Domæne Model**, og for at sørge for `Game` ikke blev bloated af ansvarsopgaver.

Overblikket over dette kan ses i figur 2 on the next page



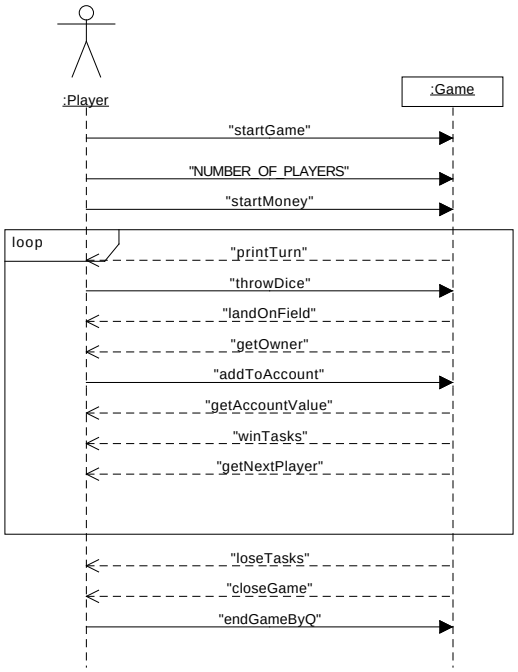
Figur 2: BCE Model

6 Systemsekvens Diagram

Et **SSD** er til for at få en indsigt i, hvordan et program ser ud. I dette diagram kan det ses, hvordan systemet kom til at se ud baseret på krav fra kunden.

I diagrammet ses det, at så snart et spil er startet, bliver nogen forskellige ting oprettet (ligesom i et bræt spil i virkeligheden). Der uddeles nogen penge til hver spiller og mængden af spillere. Spillet kørers så, som det ses i 3 on the following page. landOnField er tilføjet for at man kan se at det har en konsekvens at lande på et felt. Vi har for overblikket skyld ikke lavet returveje fra systemet på samtlige typer af felter, da det hurtigt kunne blive uoverskueligt for kunden.

Alt hvad der står i loop'en sker om og om igen indtil spillet er forbi ved at alle på nær 1 spiller er gået falit. Når en spiller er tilbage erklæres han vinder og spillet kan lukkes.



Figur 3: Systemsekvens Diagram

7 Kode

Her vil vi forklare hvordan vi har grebet selve koden an.

7.1 Struktur og pakker

Ligesom i de to foregående projekter, er programmet skrevet med fokus på at overholde **BCE**-modellen. Kort fortalt betyder det for opdelingen i pakker, at alle klasser er inddelt i pakker efter deres type ift. **BCE**. For en mere uddybende beskrivelse henvises til det tilsvarende afsnit i [1913a] og [1913b] rapporterne.

7.2 Genanvendelse af kode og opbygning

På baggrund af hensigtsmæssigt og godt design i [1913a] og [1913b] projekterne, har det været muligt at genbruge store dele af koden. Således er klasserne `Die` og `DieCup` i praksis identiske med de tilsvarende klasser fra de tidligere projekter, ligesom den generelle opbygning af systemet også ligner meget.

På samme måde er det grundlæggende princip bag `TUI` og `Graphic` klasserne også identisk med 19del2 – klasserne har ikke behov for at bære data, og de forskellige metoder i klasserne interagerer ikke med hinanden gennem felter eller

lign, og kan derfor med fordel være statiske. For en mere dybdegående forklaring henvises til kodeafsnittet i 19del2.

7.3 Bemærkelsesværdige løsninger

Systemet er på nuværende tidspunkt så omfattende, at det ikke giver mening at gennemgå alle overvejelser bag implementeringen af alle klasser. I stedet udrages specielt bemærkelsesværdige dele af koden, og forklares herunder, for at give en bedre forståelse af systemets virkemåde, uden at gennemgå alt. Beskrivelserne er opdelt efter hvilke klasser de er implementeret i.

7.3.1 TUI

Noget af det første der sker, når dette spil startes, er at brugeren anmodes om at indtaste antallet af spillere. Antallet af spillere må ifølge opgavebeskrivelsen ikke være større end 6, og spillet giver ikke meget mening, hvis det er mindre end 2. Med andre ord skal der specifikt indtastes et tal mellem 2 og 6. Dette er en ny udfordring ift. de tidligere opgaver, hvor der blot blev gemt en hel streng. Foruden at teste, at det indtastede tal er i det krævede interval, ønskede vi også at lave systemet robust, og sikre at f.eks. bogstaver eller specialtegn indtastet på dette tidspunkt, ikke kunne få systemet til at gå ned. Dette ses i figur 4. Tjekket og sikringen mod karakterer som ikke er tal, sker i `TUI`, inden

```
public static int getNumberOfPlayers(Scanner scanner) {
    int numberOfPlayers = 0;

    // Get the number of players from console. Keep trying until input is
    // valid
    while (numberOfPlayers == 0) {
        printNumberRequest();
        try {
            numberOfPlayers = new Integer(getUserInput(scanner));
            if (numberOfPlayers < 2 || numberOfPlayers > 6) {
                numberOfPlayers = 0;
            }
        } catch (Exception e) {
            numberOfPlayers = 0;
        }
    }

    return numberOfPlayers;
}
```

Figur 4: Antal spillere

det sendes videre til controlleren. Antallet af spillere sættes indledningsvist til 0, og forespørgslen kører så i en løkke, som kun brydes når antallet af spillere bliver sat til noget der er forskelligt fra 0. Det input der hentes fra konsollen er som udgangspunkt en streng, og skal derfor konverteres før det kan bruges som en talværdi. Denne konvertering vil kaste en Exception, hvis der gives et input, der ikke umiddelbart giver mening som tal-værdi – f.eks. et bogstav. Det smarte er så, at denne Exception kan fanges, og kode kan udføres til at ”reparere” den

fejl, som har forårsaget den. I vores tilfælde sætter vi bare værdien for antal af spillere tilbage til 0, fordi det betyder at løkken kører igen, så brugeren bliver spurgt efter et nyt input. Ligeledes hvis der gives et input, som succesfuldt kan konverteres til en talværdi, tjekkes der om tallet er mellem 2 og 6 – hvis ikke det er det, sættes værdien tilbage til 0, og løkken kører igen. Så snart løkken brydes, returnerer metoden.

7.3.2 GameBoard

Ligesom ved et fysisk spil, indeholder dette system en spilleplade `GameBoard`, som er bygget op af felter af forskellig type. Disse felter er bygget i et system med arv, som beskrevet senere i dette afsnit, og dette giver en stor fordel for datastrukturen i `GameBoard`. Fordi alle de forskellige felttyper `Territory, LaborCamp, Fleet, Tax og Refuge` nedarver fra Field, kan der laves en enkelt liste (array) af felter, af typen `Field`, som kan indeholde alle de forskellige felter, selvom der er tale om forskellige typer objekter, med forskellige implementeringer af metoder mm. Koden ses i figur 5.

```
public static int getNumberOfPlayers(Scanner scanner) {
    int numberOfPlayers = 0;

    // Get the number of players from console. Keep trying until input is
    // valid
    while (numberOfPlayers == 0) {
        printNumberRequest();
        try {
            numberOfPlayers = new Integer(getUserInput(scanner));
            if (numberOfPlayers < 2 || numberOfPlayers > 6) {
                numberOfPlayers = 0;
            }
        } catch (Exception e) {
            numberOfPlayers = 0;
        }
    }

    return numberOfPlayers;
}
```

Figur 5: Antal spillere

7.3.3 Field

Som nævnt tidligere, er felterne bygget op med at system af arv. Alle felterne nedarver fra `Field`, som indeholder de ting der er fælles for alle felter. I realiteten er det ikke ret meget der er det samme for alle felter – der er forskellige muligheder ift. køb, der er forskellige konsekvenser (få penge, miste penge), og selv de felter der har den samme konsekvens – at man skal betale til andre – har forskellige måde at beregne beløbet på.

Alle felter har dog et navn, og alle felter har mulighed for at man kan lande på dem. Derfor indeholder `Field` et felt til navn, og en abstrakt metode – `landOnField` – der betyder at alle klasser som arver fra `Field`, skal "love" at de implementerer `landOnField`. De forskellige underklasser kan så have forskellige

implementeringer af `landOnField`, så længe typen af parametre og returværdi er de samme.

To typer felter – `Tax` og `Refuge` – nedarver direkte fra `Field`, men de resterende 3 felttyper nedarver i stedet fra `Ownable`, som så igen nedarver fra `Field`. Dette skyldes, at disse 3 felter – `Territory, LaborCamp` og `Fleet` – kan købes. Dette er en funktionalitet, som grundlæggende vil virke ens for alle tre typer felter – der skal være et pegepind til en ejer, der skal være en måde at købe feltet osv. Desuden vil den grundlæggende procedure, når der landes på feltet, også være ens – der skal tjekkes om der er nogen der ejer feltet, og hvis der er, skal der betales leje til ejeren. Hvordan lejen udregnes, er forskellige for de forskellige typer at felter, men den overordnede procedure er ens. Derfor er `landOnField`-metoden, der som bekendt skal implementeres, når klasserne arver fra `Field`, implementeret i `Ownable`. Metoden i `Ownable` kalder så `getRent-metoden`, der har forskellig implementering i hver af underklasserne.

7.3.4 Fleet

Af de underklasser, hvor der skal udregnes leje, er `Fleet` en af de mere interessante. Lejen for `Fleet` afhænger nemlig af hvor mange andre `Fleet`-felter ejeren af det felt der landes på, har. Det betyder, at det `Fleet`-felt der er landet på, er nødt til at kende ejeren af de øvrige `Fleet`-felter, for at kunne udregne lejen. I praksis implementeres det ved, at et `Fleet` felt-objekt tager den spillepladе `GameBoard` det oprettes i, med som parameter, når det oprettes. Således har `Fleet`-feltet mulighed for at gå ud og se på andre felter på den samme spillepladе.

Dernæst er problematikken blot, at de objekter på spillepladen, som dette objekt nu har adgang til, er af typen `Field`. Overklassen `Field` indeholder ikke en ejer, så før ejeren af feltet kan findes, må det konverteres til `Ownable`. Herefter er det blot en simpel løkke, som tjekker ejeren på hvert `Fleet`-felt, og summerer op. Når antallet af `Fleet`s er fundet, kan lejen simpelt findes med en switch-sætning.

7.3.5 LaborCamp

På samme måde som `Fleet`, bruges antallet af ejede felter af samme type også til beregningen af lejen ved `LaborCamp`. Fremgangsmåden til at finde antallet af ejede felter, er helt den samme som ved `Fleet`. Foruden antallet af ejede `LaborCamps`, indgår også et terningslag i beregningen af lejen – det betyder at `LaborCamp` er nødt til at have kendskab til `DieCup`, for at kunne få værdien af et terningslag. Imidlertid befinder`DieCup` sig på spillepladen `GameBoard`, så i kraft af at `LaborCamp` allerede har `GameBoard` med som argument, for at kunne kigge på andre felter, har den også let adgang til `DieCup`.

7.3.6 Ownable

Som nævnt tidligere i afsnittet, er der 3 felttyper som kan købes. Nå en spiller køber et felt, skal der trækkes nogle penge fra spillerens konto, og `owner-`

pegepinden i feltet skal sættes til at pege på spillerens objekt. Før dette sker, skal der imidlertid have været præsenteret et valg for spilleren, om hvorvidt denne ønsker at købe feltet eller ej, og lige netop dette viser sig at være den mest krævende del at implementere af denne funktionalitet.

Ideen med hele opbygningen af systemet efter **BCE**-modellen er nemlig, at elementer fra entitets-laget aldrig har direkte adgang til elementer fra brugergrænsefladelaget. Men for at kunne præsentere valget om køb for spilleren, er feltet nødt til at kalde `TUI`'en, for at printe på skærmen og tage input fra konsollen – og vil netop give det føromtalte uønskede kendskab fra brugergrænsefladen til en entitet. Alternativt kunne man måske forstille sig, at feltet så havde kendskab til controlleren, og så kunne kalde TUI'en den vej igennem, men det vil også bryde mønsteret – ideen er jo at det er controlleren der skal kontrollere programmet, og kalde metoder i entiteter og brugergrænsefladen.

Uanset hvordan det drejes, er der ikke rigtigt en perfekt løsning på denne problemstilling – i hvert fald ikke uden at der skal ændres på de metoder, som er givet i opgavebeskrivelsen, der som udgangspunkt ikke må ændres. Med andre ord handler det nok om at finde den mindst dårlige løsning.

I vores projekt vælger vi at udnytte, at både felterne og controlleren har kendskab til spillerens objekt. I stedet for at kalde en metode, når en bruger lande på et felt der kan købes, sættes således et "flag" i spillerens objekt `isOnBuyableField = true`. Når der så returneres til controlleren, kan der tjekkes på om spilleren står på et felt der kan købes, og selve købet foretages i controlleren, som jo har kendskab til både `TUI` og felterne.

7.3.7 Tax

Der er to typer af `Tax` – den simple, hvor der blot betales et fast beløb, og den mere avancerede, hvor der kan vælges mellem et fast beløb og en procentdel af spillerens formue. Sidstnævnte er interessant, dels fordi den er nødt til at kende ejeren af alle andre felter, for at kunne beregne spillerens formue, dels fordi den skal spørge spilleren hvilken mulighed der foretrækkes.

At beregne formuen er i store træk implementeret på samme måde som `Fleet` og `LaborCamp` – feltet tager `GameBoard` med som parameter, og kan så få fat i ejeren af de andre felter. Her summers så ikke op på antal, men på pris.

At spørge brugeren, giver til gengæld samme problematik som omtalt ved køb af felter i `Ownable` – en entitet er nødt til at "snakke" med brugergrænsefladen. Det kunne løses på samme måde som ved køb af felter, ved at bruge spillerens objekt til at gemme et "flag", men forskellen er, at både det faste `Tax`-beløb og det udregnede procentvise `Tax`-beløb er nødt til at med ud til controlleren, for at der kan opkræves korrekt. Det betyder dels, at der, foruden "flaget", også skal gemmes to talværdier i spillerens objekt, og dels at man effektivt vil have flyttet hele logikken fra Tax ud i controlleren. I praksis vil den løsning ganske vist løse opgaven og give pæne diagrammer, hvor der ikke er nogen synlig kobling mellem `Tax` og `TUI` – men reelt giver det en skjult kobling, som måske i virkeligheden er højere end den ville være, hvis `Tax` bare havde kendskab direkte til `TUI`, og som under alle omstændigheder er langt sværere at

gennemskue.

Derfor vælger vi at lade `Tax` have kendskab til `TUI`, men knytter hertil samtidigt en klar bemærkning om, at dette er et af de steder hvor systemet kan forbedres.

7.3.8 Game

Strukturen og ideen i `Game`-controlleren er stadig helt den samme som i de tidligere projekter. En af de største forskelle i controlleren, er muligheden for mere end 2 spillere, og måske endnu vigtigere, det faktum at der ikke bare er en spiller som vinder – der er spillere som taber løbende, og som så skal udgå af spillet. Det betyder nemlig, at der ikke blot kan laves en simpel valgsætning, som afgør om en spiller har vundet – i stedet må der tælles hvor mange spillere der er tilbage, hver gang en spiller er udgået.

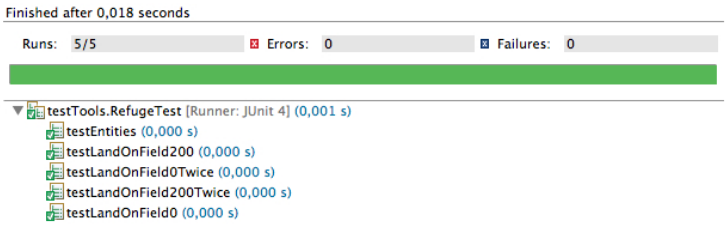
8 Test

Vi blev bedt om at lave en **J-unit** test af vores felttypers `landOnField` metoder. Disse vil være beskrevet i dette afsnit.

8.1 Test af Refuge

Sammen med opgaven fik vi et bilag, der beskrev hvordan testen af `Refuge` skulle laves.

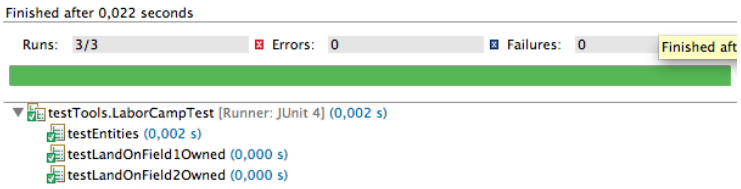
Til formålet oprettes en testklasse `RefugeTest`. I denne klasse tog vi indholdet fra bilagene, og modificerede det lidt. For at få koden til at passe til vores program ændrede vi nogle småting. Da vi har benyttet os af **BCE** notationen til vores pakker, måtte vi importere vores `Player` til at oprette objekter igennem `intity` pakken. Samtidig har vi kun en metode til både at trække penge og give penge på en spillers konto. Derfor har vi slettet `Neg200` funktionerne i alle tests. Derfor bestemmer konsekvensen af feltypen i stedet, om der skal trækkes eller indsættes penge. Kørslen af testen ses i figur 6.



Figur 6: Resultatet af testkørsel Refuge

8.2 Test af Laborcamp

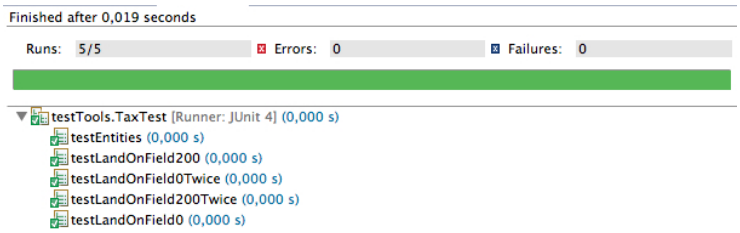
`LaborCampTest` er stort set magen til. Her har vi bare trukket beløbet fra på `expected`, da vores felttype nu giver en negativ konsekvens for spilleren. Samtidig har vi omdøbt pointerne til *Field*, så de hedder noget med `labor`. Kørslen af testen ses i figur 7.



Figur 7: Resultatet af testkørsel LaborCamp

8.3 Test af Tax

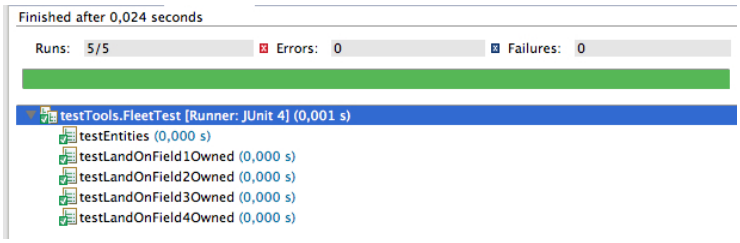
Den første testklasse til `TaxTest`, er præcis magen til `LaborCampTest`. Bortset fra vi har skiftet navnene, så de passer til denne testklasse. Kørslen af testen ses i figur 8.



Figur 8: Resultatet af testkørsel Tax

8.4 Test af Fleet

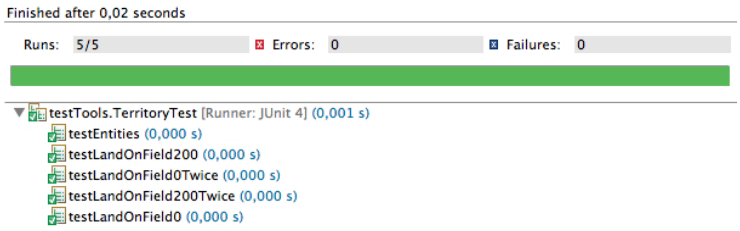
Testen af `Fleet` krævede en lille smule mere arbejde. Her var vi nødt til at oprette vores `Gameboard`, for at kune sætte en ejer på vores `Fleet`. Dette gjorde vi på samme måde, som vi satte vores `player` nemlig med `this.owner = new Player(1000, "Andersine");` Vi opretter også vores `Fleet` felter `this.gameBoard.setField(new Fleet("Fleet1", 0, gameBoard), 18);`. Vi fortæller nu, hvilke felt vores `owner` har, og hvilket felt vores `player` står på, og så er det ellers stort set samme procedure som de andre tests. Kørslen af testen ses i figur 9.



Figur 9: Resultatet af testkørsel Fleet

8.5 Test af Territory

Denne testklasse er bygget op på stort set samme måde som `Fleet` med en `owner` og en `player`. Det eneste specielle er at vi bliver nødt til at kalde `((Ownable)ter200).setOwner(owner);` for at kunne sætte ejeren. Kørslen af testen ses i figur 10 on the next page.



Figur 10: Resultatet af testkørsel Territory

8.6 Test og fejlfinding generelt

Vi har genbrugt vores "snydeterning"fra sidste projekt, som vi har brugt til lettere at lande på bestemte felter, når vi skulle teste en bestemt felttype. Ellers har vi benyttet os de **White Box** tests, der var lavet en **Junit** skabelon til.

Ud over det har vi lavet en enkelt **User Test**, da al vores kode var færdig. Lige som sidst skabte det forvirring, at spillet skulle startes i Eclipse, og beskederne kom både i konsol og via **GUI**.

9 GRASP (General Responsibilty Assignment Software Patterns)

Vi vil i dette afsnit beskrive de forskellige GRASP patterns, som vi har benyttet i vores program.

9.1 Controller

Vi har for overskueligeheden, og funktionaliteten i vores program, lavet en controller, *Game*, der styrer alt hvad der sker. Alle kald og informationer der går på tværs af programmet går igennem vores kontroller uden at den egentlig har noget med nogen af signalerne at gøre udover at bestemme hvad der skal ske i de forskellige tilfælde. Man kan vel næsten sige at controlleren er vores lyskryds hvor en masse veje mødes og bliver omdirigeret.

9.2 Creater

I vores program har vi gjort brug af Creatorer. Eksempelvis kan vi se på vores klasse diagram at der laves et object af en klassen *Field* i *Ownable*. På den måde kan vi have adgang til data fra Field uden at skulle tilgå klassen. Det er også rigtig godt for vores ønske om høj binding og lav kobling.

9.3 Expert

I vores kode har der blandt andet været brug for en expert til at holde styr på vores felter på spillepladen. Derfor har vi lagret alle informationer, såsom hvad en grund koster at købe eller hvad den koster at lande på når en anden ejer den, i *gameBoard* klassen. Det gør samtidigt at vi hurtigt kan komme til informationerne fra andre klasser da vi kun skal gå et sted hen for at hente informationerne. Man kan bruge et eksempel fra den virkelige verden. Forestil dig at du skal slå 20 dyr op. Hvis du skal slå op i en bog for hvert dyr kan det tage sin tid, i forhold til hvis du kun skal have fat i en bog.

9.4 High Cohesion (Høj binding)

Høj binding er noget man altid stræber efter i et system. Det har vi også gjort som man kan se på vores *Account* og *Player* klasser. De kender til hindanden uden at de kan gøre andet end at give kommandoer. Det vil sige at Player for eksempel kan bede om data om en spiller fra *Account*, og modsat kan *Account* modtage data fra *Player* om en spiller.

9.5 Indirection

Indirection er noget der bliver brugt i vores kode. Faktisk er vores controller *Game* en indirection da den lever op til det krav at den formidler information imellem to parter der ikke kender hinanden.

9.6 Low Coupling (Lav kobling)

Et godt eksempel på lav kobling i vores program er imellem vores *Field* og *GameBoard* klasser. Her er det kun *GameBoard* der kender til *Field*. Generelt i vores klassediagram kan man se at der ikke er nogen returnerende pile, hvilket antyder lav kobling.

9.7 Polymorphism

Polymorfi bruger vi til vores *Field*, *Ownable*, *Game*, *Fleet*, *LabourCamp*, og *Territory klasser*. Polymorfi er et andet ord for nedarving til flere klasser. Vi nedarver fra Field, for at gøre vores program det mindre, hurtigere og mere overskueligt. Havde vi ikke brugt det, var vi kommet ud i en situation hvor vi ville skulle have lavet en masse ekstra kald for at hente alle data fra andre klasser.

9.8 Protected Variations

Vi bruger meget *Protected Variations* i vores kode da det sørger for at vi ikke ved en fejl kommer til at ændre en variabel et sted hvor det ikke er intentionen. Det kan bl. a. ses i vores kode i *Ownable* klassen hvor "price"er protected netop for at undgå dette.

9.9 Pure Fabrication

Vores Field klasse kunne anses for at være *Pure Fabrication*. Egentlig *kan man godt* gøre det uden en Field klasse, det ville bare ødelægge alle former for lav kobling man ellers arbejder hårdt for at opnå i et system.

10 Design Sekvens Diagram

I modsætning til de tidligere projekter, hvor der blev udarbejdet sekvensdiagram over hele det samlede systems virkemåde, er der til dette projekt alene lavet sekvensdiagram til en enkelt metode – landOnField i Fleet – som krævet i opgavebeskrivelsen. Dette skyldes, at systemet nu er så omfattende, at det dels vil være enormt tidskrævende at udvikle et designesekvensdiagram for hele systemet, og dels vil være så stort, at det næppe vil skabe andet end forvirring, og dermed give meget lidt værdi ift. omkostningerne ved udarbejdelsen.

Diagrammet over landOnField metoden i Fleet giver til gengæld et rigtig fint indblik i systemets generelle virkemåde, og er samtidigt ikke større end at det kan overskue af de fleste.

Det er værd at bemærke, at vi vælger en notationsform, hvor der ikke indtegnes retur-pile, med mindre der returneres en værdi af betydning. Dette fordi der ikke benyttes asynkrone kald i systemet, og det derfor bør være klart hvordan sekvensen foregår, selv uden returpile – således vurderer vi, at det giver bedre mening at udelade de "tomme" returpile, idet det giver et simplere diagram, der er nemmere at overskue, uden at det forstyrrer meningen i figuren.

Desuden har vi tilladt os at slå de 4 identiske (i hvert fald i forhold til sekvensen) Fleet felter på fields-pladserne 18-21 sammen til en enkelt "instans" på figuren – det giver mening fordi de alene bliver tilgået i en løkke, som behandler alle fire objekter ens.

I korte træk foregår sekvensen således:

1. Der undersøges om feltet er ejet af en spiller. Er det ikke det, sættes isOnBuyableField-"flaget" i spillerens objekt, som bevirker at spilleren senere får mulighed for at købet feltet. Metoden returnerer herefter. Er feltet ejet af en spiller laves yderligere et tjek – nemlig om feltet er ejet af den spiller som er landet på feltet – er det det, returnerer metoden. Er det ikke det, iværksættes koden til at beregne leje og overføre lejen til feltets ejer.

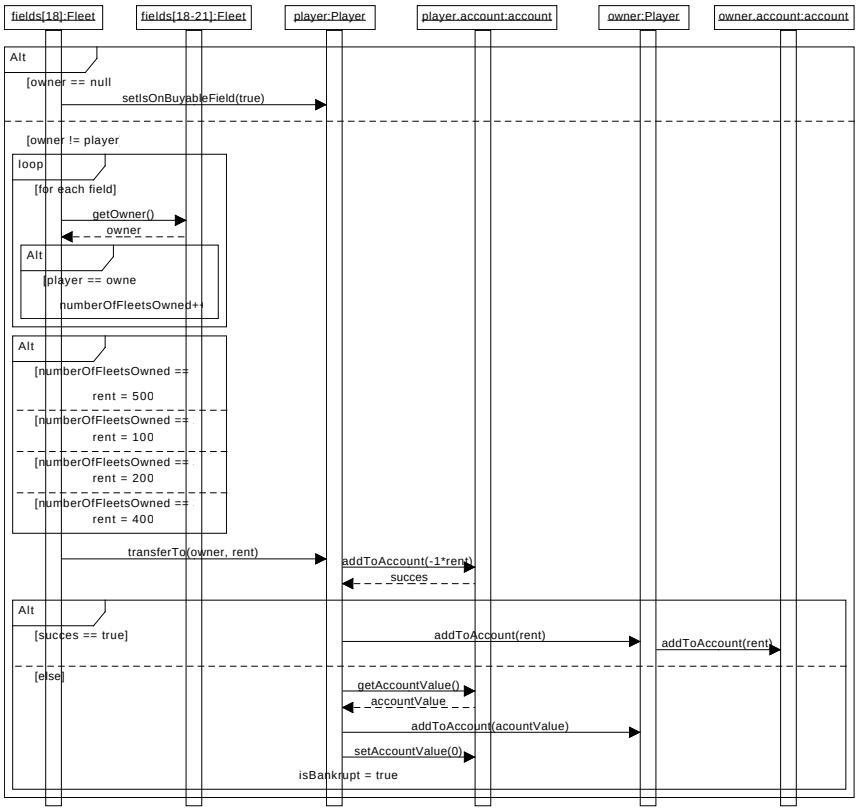
2. For at beregne lejen, skal der tælles hvor mange andre Fleet felter ejeren af dette felt har. Dette sker ved at hente de andre felter fra GameBoard, og sammenligne ejeren af disse, med ejeren af dette felt. For hvert felt der matcher, tælles én op.

3. Når antallet af felter kendes, kan lejen findes. I koden sker det med en switch-sætning, men ift. sekvensen er den egentlige implementering ligegyldig – det vigtige er blot, at der findes et beløb for leje ud fra antallet af ejede Fleet felter.

4. Når lejen er udregnet, overføres beløbet fra spilleren som landede på feltet, til spilleren som ejer feltet. Overførslen sker med transferTo-metoden i Player, som samtidigt tjekker om spilleren har nok penge til at betale. Har han det, overføres beløbet, og metoden returnerer. Har han ikke det, overføres alt hvad spilleren har, og spilleren erklæres konkurs. Her er det

værd at bemærke, at det faktisk er den samme metode der bruges til at lægge penge til og trække penge fra på en konto, så i praksis udføres tjekket for, om spillerens kontobalance kommer under 0 også ved modtageren af beløbet – men det er udeladt på diagrammet, idet det under normale omstændigheder altid vil gå godt, og dermed resultere i den sekvens som er illustreret. Et alternativt udfald her vil kræve uhensigtsmæssige ændringer i koden, eller deciderede platformsfejl, og vil i praksis kun være en teoretisk mulighed.

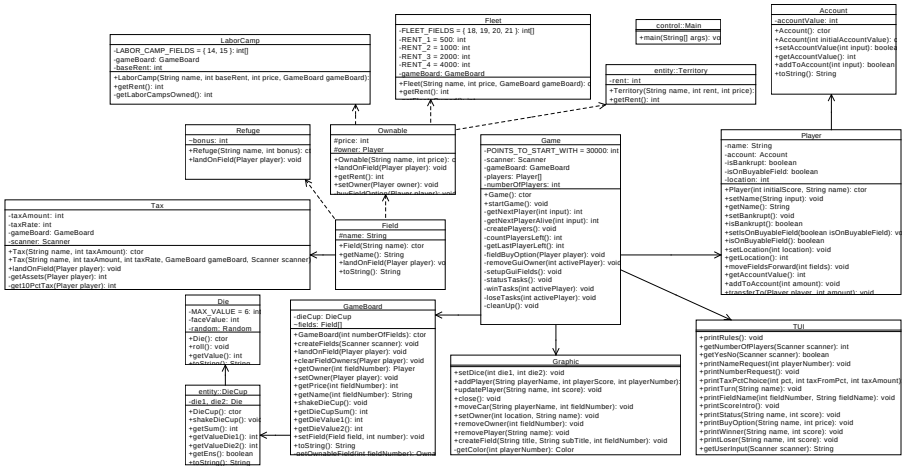
Dette ses i figur 11



Figur 11: Design Sekvens Diagram

11 Design-klassediagram

Her kan man se vores Design Klasse Diagram. Hvis man kigger nok på det, vil man se at der er masser af pile, men ingen af dem har nogen der går retur. Det vil sige det er lykkedes os på en nogenlundef ornuftig måde at overholde de fleste patterns i **GRASP**. Dette ses i figur 12.



Figur 12: Design-Klassediagram

12 Kildeliste

Her vil vi oplyse om hvilke kilder, der er brugt til rapporten. Vi vil både oplyse om hvilke bøger, hjemmesider og software vi har brugt.

Referencer

[1913a] Gruppe 19. *19_del1*. DTU, okt. 2013.

[1913b] Gruppe 19. *19_del2*. DTU, nov. 2013.

[IOO13] IOOuterActive. *CDIO opgave 3 (Version 12)*. IOOuterActive, nov. 2013.

[Wik13] Wikipedia. *FURPS — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-November-2013]. 2013. URL: `\url{http://en.wikipedia.org/wiki/FURPS}`.

13 Anvendte værktøjer

- **Eclipse - kepler**. Brugt som vores værktøj til at kode java i.
- **UMLet**. Foretrukket værktøj til **UML** diagrammer.
- **Dropbox**. Brugt til at dele filer imellem os.
- **TexMaker**. Brugt for at skabe en flot rapport skrevet i LaTeX

14 Bilag

14.1 Kode

Her vil hele vores kode til programmet være repræsenteret som bilag.

14.1.1 TUI - Boundary

```
1  package boundary;
2
3  import java.util.Scanner;
4
5  /**
6   * Class to handle input/output to/from the console.
7   *
8   * @author DTU 02312 Gruppe 19
9   *
10  */
11 public class TUI {
12   /**
13    * Prints game rules.
14    */
15   public static void printRules() {
16     System.out.println("");
17     System.out
18         .println("
                ----------------------------------------------------------------------
                ");
19     System.out
20         .println("| Rules:

                |");
21     System.out
22         .println("| You roll two dice. The sum determines
                    which field you hit. |");
23     System.out
24         .println("| Each field have it's own value.
                                          |");
25     System.out
26         .println("| You get points on some, and loses
                   points on others.        |");
27     System.out
28         .println("| A player wins when the others go
                   bankrupt.                |");
29     System.out
```

```
30          .println("| Press \"Enter\" to roll, press \"q\"
                to exit.                    |");
31      System.out
32          .println("
                ------------------------------------------------------------------
                ");
33      System.out.println("");
34    }
35
36    /**
37     * Gets an integer from 1-6. Keeps asking until valid
           input is given.
38     *
39     * @param scanner An open scanner to read from.
40     * @return An integer from 1-6.
41     */
42    public static int getNumberOfPlayers(Scanner scanner) {
43      int numberOfPlayers = 0;
44
45      // Get the number of players from console. Keep
           trying until input is
46      // valid
47      while (numberOfPlayers == 0) {
48        printNumberRequest();
49        try {
50          numberOfPlayers = new Integer(getUserInput(
                scanner));
51          if (numberOfPlayers < 0 || numberOfPlayers > 6) {
52            numberOfPlayers = 0;
53          }
54        } catch (Exception e) {
55          numberOfPlayers = 0;
56        }
57      }
58
59      return numberOfPlayers;
60    }
61
62    /**
63     * Method that takes input from console and translates
           it to boolean yes/no.
64     * Supports both upper and lower case.
65     * Keeps asking until valid input is given (Y/y or N/n)
           .
66     *
```

```
67      * @param scanner A scanner to use for reading from
            console.
68      * @return True if Y/y is written, False if N/n.
69      */
70     public static boolean getYesNo(Scanner scanner) {
71       String input;
72
73       while(true) {
74         input = getUserInput(scanner);
75         if ("Y".equals(input) || "y".equals(input)) {
76           return true;
77         }
78
79         if("N".equals(input) || "n".equals(input)) {
80           return false;
81         }
82
83         System.out.print("Not valid. Must be \"Y\" og \"N\"
               ");
84       }
85     }
86
87     /**
88      * Prints a short text, asking the specified player to
            type his name.
89      *
90      * @param playerNo The player number to print as part
            of the message.
91      */
92     public static void printNameRequest(int playerNumber) {
93       System.out
94           .println("Insert name for player " + (
                playerNumber + 1) + ".");
95     }
96
97     /**
98      * Prints a short text, asking the user to type the
            number of players.
99      */
100    public static void printNumberRequest() {
101      System.out.println("Select the number of players 1-6:
             ");
102    }
103
104    /**
```

```
105     * Prints a question asking if the user would like to
            pay x% in Tax instead of a fixed amount.
106     *
107     * @param pct The Tax percent to pay.
108     * @param taxFromPct The calculated amount of tax from
            percent.
109     * @param taxAmount The fixed amount of tax.
110     */
111    public static void printTaxPctChoice(int pct, int
           taxFromPct, int taxAmount) {
112      System.out.println("Would you like to pay " + pct + "
            % of your assets (" + taxFromPct + ") in Tax,
            instead of " + taxAmount + "?");
113    }
114
115    /**
116     * Prints a short text, telling the player who's turn
            it is, and asking him
117     * to roll.
118     *
119     * @param name The name to print as part of the message
            .
120     */
121    public static void printTurn(String name) {
122      System.out.print("\n\nIt's " + name + "'s turn. Press
             enter to roll.");
123    }
124
125    /**
126     * Print info about a field.
127     *
128     * @param fieldNumber The field number used as part of
            the printed text
129     * @param fieldName The field name used as part of the
            printed text
130     */
131    public static void printFieldName(int fieldNumber,
           String fieldName) {
132      System.out.println("You hit field number " +
            fieldNumber + ", " + fieldName);
133    }
134
135    /**
136     * Prints a title for the status
137     */
138    public static void printScoreIntro() {
```

```
139       System.out.println("The score is now:");
140     }
141
142     /**
143      * Prints the current status of the game. Thats all
             players name and score.
144      *
145      * @param players An array of players to get the
             information from.
146      */
147     public static void printStatus(String name, int score)
           {
148       System.out.print(name + " = " + score + "\t");
149     }
150
151     /**
152      * Prints a message in the console, giving the player
             an option to buy af field.
153      *
154      * @param name The name of the field.
155      * @param price The price of the field.
156      */
157     public static void printBuyOption(String name, int
           price) {
158       System.out.println("Would you like to buy " + name +
             " for " + price + "? (Y/N)");
159     }
160
161     /**
162      * Prints the name and score of the winning player.
163      *
164      * @param name The name of the player who should be
             declared the winner.
165      * @param score The score for the winning player.
166      */
167     public static void printWinner(String name, int score)
           {
168       System.out.println("Congratulations! " + name + " has
              won with "
169           + score + " points!\nPress Enter to exit.");
170     }
171
172     /**
173      * Prints the name and score of a player that is
             bankrupt.
174      *
```

```
175     * @param name The name of the player who should be
            declared bankrupt.
176     * @param score The score for the losing player.
177     */
178    public static void printLoser(String name, int score) {
179      System.out.println("\nSorry! " + name + " you are
            bankrupt.");
180    }
181
182    /**
183     * Reads a line from the console.
184     *
185     * @param scanner The scanner to read from
186     * @return Whatever the user inputs.
187     */
188    public static String getUserInput(Scanner scanner) {
189      return scanner.nextLine();
190    }
191  }
```

14.1.2 Graphic - Boundary

```
1  package boundary;
2
3  import java.awt.Color;
4
5  import boundaryToMatador.GUI;
6
7  /**
8   * Class to send commands to the GUI.
9   *
10  * @author DTU 02312 Gruppe 19
11  *
12  */
13 public class Graphic {
14   /**
15    * Method to set the value of the dice on the GUI.
16    *
17    * @param die1 Value of die1.
18    * @param die2 Value of die2.
19    */
20   public static void setDice(int die1, int die2) {
21     GUI.setDice(die1, die2);
22   }
23
24   /**
```

```
25      * Method to add a player.
26      *
27      * @param playerName The name of the player to add.
28      * @param playerScore The score of the player to add.
29      */
30     public static void addPlayer(String playerName, int
            playerScore, int playerNumber) {
31       GUI.addPlayer(playerName, playerScore, getColor(
             playerNumber));
32     }
33
34     /**
35      * Method to update all players information on the GUI
            according to a given
36      * array of player objects.
37      *
38      * @param players The array of player objects to get
            the information from.
39      */
40     public static void updatePlayer(String name, int score)
            {
41       GUI.setBalance(name, score);
42     }
43
44     /**
45      * Close the GUI window.
46      */
47     public static void close() {
48       GUI.close();
49     }
50
51     /**
52      * Method to move a car from any field to the field
            number given.
53      *
54      * @param playerName The name of the player who's car
            should be moved.
55      * @param fieldNumber The number of the field the car
            should be moved to.
56      */
57     public static void moveCar(String playerName, int
            fieldNumber) {
58       GUI.removeAllCars(playerName);
59       GUI.setCar(fieldNumber, playerName);
60     }
61
```

```
62    /**
63     * Method to set owner of a field. Marks the given
          field number with the given players color.
64     *
65     * @param location The number of the field.
66     * @param name The name of the player to set as owner.
67     */
68    public static void setOwner(int location, String name)
          {
69      GUI.setOwner(location, name);
70    }
71
72    /**
73     * Method to remove the owner-marking from a field.
74     *
75     * @param fieldNumber The number of the field to remove
          owner from.
76     */
77    public static void removeOwner(int fieldNumber) {
78      GUI.removeOwner(fieldNumber);
79    }
80
81    /**
82     * Method to remove a players car from the board.
83     *
84     * @param name Name of the player to remove car for.
85     */
86    public static void removePlayer(String name) {
87      GUI.removeAllCars(name);
88    }
89
90    /**
91     * Method to setup all the parameters of a field on the
          GUI at the
92     * same time.
93     *
94     * @param title The title to set.
95     * @param subTitle The subtitle to set.
96     * @param fieldNumber The number of the field to change
          .
97     */
98    public static void createField(String title, String
          subTitle, int fieldNumber) {
99      GUI.setTitleText(fieldNumber, title);
100     GUI.setSubText(fieldNumber, subTitle);
101     GUI.setDescriptionText(fieldNumber, title);
```

```
102     }
103
104     private static Color getColor(int playerNumber) {
105       switch(playerNumber) {
106       case 0:
107         return Color.RED;
108       case 1:
109         return Color.BLUE;
110       case 2:
111         return Color.YELLOW;
112       case 3:
113         return Color.WHITE;
114       case 4:
115         return Color.GREEN;
116       default:
117         return Color.BLACK;
118       }
119     }
120 }
```

14.1.3 Main - Controller

```
1  package control;
2
3  public class Main {
4      public static void main(String[] args) {
5        Game game = new Game();
6        game.startGame();
7      }
8  }
```

14.1.4 Game - Controller

```
1  package control;
2
3  import java.util.Scanner;
4
5  import boundary.Graphic;
6  import boundary.TUI;
7  import entity.GameBoard;
8  import entity.Player;
9
10 /**
11  * This is controller class in the game.
12  *
13  * @author DTU 02312 Gruppe 19
14  *
```

```
15   */
16  public class Game {
17    private final int POINTS_TO_START_WITH = 30000;
18
19    private Scanner scanner;
20    private GameBoard gameBoard;
21    private Player[] players;
22
23    private int numberOfPlayers;
24
25    /**
26     * Game constructor. Creates new instances of the
          required classes.
27     */
28    public Game() {
29      scanner = new Scanner(System.in);
30      gameBoard = new GameBoard(22);
31      gameBoard.createFields(scanner);
32      setupGuiFields();
33    }
34
35    /**
36     * Start the game.
37     */
38    public void startGame() {
39      int activePlayer = 0;
40      String userInput;
41
42      TUI.printRules();
43      numberOfPlayers = TUI.getNumberOfPlayers(scanner);
44      players = new Player[numberOfPlayers];
45
46      createPlayers();
47
48      // Start of the actual game-turns
49      while (true) {
50        // Write whos turn it is and wait for input
51        TUI.printTurn(players[activePlayer].getName());
52        userInput = TUI.getUserInput(scanner);
53
54        // Exit game if user inputs "q"
55        if ("q".equals(userInput)) {
56          cleanUp();
57        }
58
59        gameBoard.shakeDieCup();
```

```
60          players[activePlayer].moveFieldsForward(gameBoard.
                getDieCupSum());
61          Graphic.moveCar(players[activePlayer].getName(),
                players[activePlayer].getLocation());
62          Graphic.setDice(gameBoard.getDieValue1(), gameBoard
                .getDieValue2());
63          TUI.printFieldName(players[activePlayer].
                getLocation(), gameBoard.getName(players[
                activePlayer].getLocation()));
64          gameBoard.landOnField(players[activePlayer]);
65
66          if(players[activePlayer].isOnBuyableField()) {
67            fieldBuyOption(players[activePlayer]);
68          }
69
70          statusTasks();
71
72          if (players[activePlayer].isBankrupt()) {
73            loseTasks(activePlayer);
74          }
75
76          // Switch turn to the next player
77          activePlayer = getNextPlayerAlive(activePlayer);
78        }
79      }
80
81      private int getNextPlayer(int input) {
82        if (input + 1 >= numberOfPlayers) {
83          return 0;
84        }
85
86        return input + 1;
87      }
88
89      private int getNextPlayerAlive(int input) {
90        int playerToTest = getNextPlayer(input);
91
92        while (players[playerToTest].isBankrupt()) {
93          playerToTest = getNextPlayer(playerToTest);
94        }
95
96        return playerToTest;
97      }
98
99      private void createPlayers() {
100       int i;
```

```
101       String userInput;
102
103       // Ask for all player names and save them in the
              player objects.
104       for (i = 0; i < numberOfPlayers; i++) {
105         TUI.printNameRequest(i);
106
107         userInput = TUI.getUserInput(scanner);
108         if("".equals(userInput)) {
109           userInput = "Player" + (i+1);
110         }
111
112         players[i] = new Player(POINTS_TO_START_WITH,
              userInput);
113         Graphic.addPlayer(players[i].getName(), players[i].
              getAccountValue(), i);
114       }
115     }
116
117     private int countPlayersLeft() {
118       int i, playersLeft = 0;
119
120       for (i = 0; i < numberOfPlayers; i++) {
121         if (!players[i].isBankrupt()) {
122           playersLeft++;
123         }
124       }
125
126       return playersLeft;
127     }
128
129     private int getLastPlayerLeft() {
130       int i;
131
132       for (i = 0; i < numberOfPlayers; i++) {
133         if (!players[i].isBankrupt()) {
134           return i;
135         }
136       }
137
138       return 0;
139     }
140
141     private void fieldBuyOption(Player player) {
142       TUI.printBuyOption(gameBoard.getName(player.
              getLocation()), gameBoard.getPrice(player.
```

```
            getLocation()));
143       boolean wantToBuy = TUI.getYesNo(scanner);
144
145       if (wantToBuy) {
146         player.addToAccount(-1 * gameBoard.getPrice(player.
              getLocation()));
147         gameBoard.setOwner(player);
148         Graphic.setOwner(player.getLocation(), player.
              getName());
149       }
150
151       player.setIsOnBuyableField(false);
152     }
153
154     private void removeGuiOwner(int activePlayer) {
155       int i;
156       for(i = 0; i<=21; i++) {
157         if(gameBoard.getOwner(i) == players[activePlayer])
              {
158           Graphic.removeOwner(i);
159         }
160       }
161     }
162
163     private void setupGuiFields() {
164       int i;
165
166       for(i = 1; i<=21; i++) {
167         Graphic.createField(gameBoard.getName(i), "", i);
168       }
169
170       // Remove unused fields from GUI
171       for (i = 22; i < 41; i++) {
172         Graphic.createField("", "", i);
173       }
174     }
175
176     private void statusTasks() {
177       int i;
178       TUI.printScoreIntro();
179       for(i=0; i<players.length; i++) {
180         TUI.printStatus(players[i].getName(), players[i].
              getAccountValue());
181         Graphic.updatePlayer(players[i].getName(), players[
              i].getAccountValue());
182       }
```

```
183     }
184
185     private void winTasks(int activePlayer) {
186       TUI.printWinner(players[activePlayer].getName(),
187           players[activePlayer].getAccountValue());
188       TUI.getUserInput(scanner);
189       cleanUp();
190     }
191
192     private void loseTasks(int activePlayer) {
193       TUI.printLoser(players[activePlayer].getName(),
              players[activePlayer].getAccountValue());
194       removeGuiOwner(activePlayer);
195       Graphic.removePlayer(players[activePlayer].getName())
              ;
196       gameBoard.clearFieldOwners(players[activePlayer]);
197
198       if (countPlayersLeft() == 1) {
199         winTasks(getLastPlayerLeft());
200       }
201     }
202
203     private void cleanUp() {
204       Graphic.close();
205       scanner.close();
206       System.exit(0);
207     }
208 }
```

14.1.5 Player - Entity

```
1  package entity;
2
3  /**
4   * Class to create a player. This class can be used for
        storing a name and account for a player.
5   *
6   * @author DTU 02312 Gruppe 19
7   *
8   */
9  public class Player {
10     private String name;
11     private Account account;
12     private boolean isBankrupt;
13     private boolean isOnBuyableField;
14     private int location;
```

```
15
16      /**
17       * Constructor that initiates name to empty and
             account to an initial score.
18       */
19      public Player(int initialScore, String name) {
20          this.name = name;
21          account = new Account(initialScore);
22          isBankrupt = false;
23          isOnBuyableField = false;
24          location = 1;
25      }
26
27      /**
28       * Saves the given name.
29       *
30       * @param input The name to save.
31       */
32      public void setName(String input) {
33          name = input;
34      }
35
36      /**
37       * Gets the name of the player
38       *
39       * @return The name of this player.
40       */
41      public String getName() {
42          return name;
43      }
44
45      /**
46       * Method to set the bankrupt status, which is used
             to determine if a player is still in the game.
47       */
48      public void setBankrupt() {
49        isBankrupt = true;
50      }
51
52      /**
53       * Method to get the bankrupt status, which is used
             to determine if a player is still in the game.
54       *
55       * @return True if player is bankrupt, otherwise
             false.
56       */
```

```
57      public boolean isBankrupt() {
58        return isBankrupt;
59      }
60
61      /**
62       * Method to set the boolean for whether the player
             is on a field that can be bought.
63       *
64       * @param isOnBuyableField What the value should be
             set to.
65       */
66      public void setIsOnBuyableField(boolean
           isOnBuyableField) {
67        this.isOnBuyableField = isOnBuyableField;
68      }
69
70      /**
71       * Method to get the boolean for whether the player
             is on a field that can be bought.
72       *
73       * @return True if player is on a field that can be
             bought, otherwise false.
74       */
75      public boolean isOnBuyableField() {
76        return isOnBuyableField;
77      }
78
79      /**
80       * Method to set the players current location (field
             number).
81       *
82       * @param location The location to set the player to.
83       */
84      public void setLocation(int location) {
85        this.location = location;
86      }
87
88      /**
89       * Method to get the players current location (field
             number).
90       *
91       * @return The players current location.
92       */
93      public int getLocation() {
94        return location;
95      }
```

```
96
97      /**
98       * Method to move a player forward on the board.
99       * Takes the players current location and adds a
             given number of fields.
100      *
101      * @param fields The number of fields to move forward
             .
102      */
103     public void moveFieldsForward(int fields) {
104       location = location + fields;
105       if(location > 21) {
106         location = location - 21;
107       }
108     }
109
110     /**
111      * Method to get a players account value.
112      *
113      * @return The players current account value.
114      */
115     public int getAccountValue() {
116       return account.getAccountValue();
117     }
118
119     /**
120      * Method to add to a players account value.
121      * Takes what the player has in the account and adds
             a given number.
122      *
123      * @param amount Amount to add to the account.
124      */
125     public void addToAccount(int amount) {
126       boolean succes = account.addToAccount(amount);
127
128       if(!succes) {
129         isBankrupt = true;
130         account.setAccountValue(0);
131       }
132     }
133
134     /**
135      * Method to transfer "money" from one player to
             another.
136      * Also checks if the player has enough money for the
              transfer, and sets player to bankrupt if not.
```

```
137         *
138         * @param player Player to transfer to.
139         * @param amount Amunt to transfer.
140         */
141        public void transferTo(Player player, int amount) {
142          boolean succes = account.addToAccount(-1*amount);
143
144          if(succes) {
145            player.addToAccount(amount);
146          }
147          else {
148            player.addToAccount(account.getAccountValue());
149            account.setAccountValue(0);
150            isBankrupt = true;
151          }
152        }
153
154        /**
155         * Method that makes a text with the most important
               values in the class, and some description.
156         *
157         * @return A coherent string with values of name and
               account.
158         */
159        public String toString() {
160            return "Name = " + name + ", Account = " +
                   account;
161        }
162    }
```

14.1.6 Account - Entity

```
1   package entity;
2
3   /**
4    * Class to create an Account. This class can be used to
         store a number representing an account value.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class Account {
10    private int accountValue;
11
12    /**
```

```
13      * Constructor to create a new account. Takes no
            arguments, and sets the initial account value to 0.
14      */
15     public Account() {
16       accountValue = 0;
17     }
18
19     /**
20      * Constructor to create a new account. Takes an
            argument for initial account value.
21      *
22      * @param initialAccountValue The value to set the new
            account to.
23      */
24     public Account(int initialAccountValue) {
25       accountValue = initialAccountValue;
26     }
27
28     /**
29      * Method to set the account.
30      * Checks if the account will go below 0 before setting
            it.
31      *
32      * @param input The value to set the account to.
33      * @return True if the account was set correctly. False
            if the account was not set, because the input
            value was below 0.
34      */
35     public boolean setAccountValue(int input) {
36       if(input >= 0) {
37         accountValue = input;
38         return true;
39       }
40
41       return false;
42     }
43
44     /**
45      * Method to get the value of the account.
46      *
47      * @return The value of the account.
48      */
49     public int getAccountValue() {
50       return accountValue;
51     }
52
```

```
53    /**
54     * A method to add to the account, so the resulting
          value will be the existing value plus the input
          value.
55     * Checks if the account will go below 0 before adding
          to it.
56     *
57     * @param input The value to add.
58     * @return True if the account was set correctly. False
           if the account was not set, because the value
          would have gone below 0.
59     */
60    public boolean addToAccount(int input) {
61      if(accountValue + input >= 0) {
62        accountValue = accountValue + input;
63        return true;
64      }
65
66      return false;
67    }
68
69    /**
70     * A method to get the contents of the account as a
          string.
71     *
72     * @return The account value as a string.
73     */
74    public String toString() {
75      return Integer.toString(accountValue);
76    }
77  }
```

14.1.7 Gameboard - Entity

```
1  package entity;
2
3  import java.util.Scanner;
4
5  /**
6   * Class to create a game board. This class takes in a
       lot of fields and makes
7   * it a board.
8   *
9   * @author DTU 02312 Gruppe 19
10  *
11  */
```

```
12 public class GameBoard {
13   private DieCup dieCup;
14   Field[] fields;
15
16   /**
17    * Constructor that makes an array for fields and a
          DieCup
18    */
19   public GameBoard(int numberOfFields) {
20     dieCup = new DieCup();
21     fields = new Field[numberOfFields];
22   }
23
24   /**
25    * Creates all the fields according to the game rules.
26    */
27   public void createFields(Scanner scanner) {
28     fields[1] = new Territory("Tribe Encampment", 100,
          1000);
29     fields[2] = new Territory("Crater", 300, 1500);
30     fields[3] = new Territory("Mountain", 500, 2000);
31     fields[4] = new Territory("Cold Dessert", 700, 3000);
32     fields[5] = new Territory("Black Cave", 1000, 4000);
33     fields[6] = new Territory("The Werewall", 1300, 4300)
          ;
34     fields[7] = new Territory("Mountain Village", 1600,
          4750);
35     fields[8] = new Territory("South Citadel", 2000,
          5000);
36     fields[9] = new Territory("Palace ates", 2600, 5500);
37     fields[10] = new Territory("Tower", 3200, 6000);
38     fields[11] = new Territory("Castle", 4000, 8000);
39
40     fields[12] = new Refuge("Walled city", 5000);
41     fields[13] = new Refuge("Monastry", 500);
42
43     fields[14] = new LaborCamp("Huts in the mountain",
          100, 2500, this);
44     fields[15] = new LaborCamp("The pit", 100, 2500, this
          );
45
46     fields[16] = new Tax("Goldmine", 2000);
47     fields[17] = new Tax("Caravan", 4000, 10, this,
          scanner);
48
49     fields[18] = new Fleet("Second Sail", 4000, this);
```

```
50      fields[19] = new Fleet("Sea Grover", 4000, this);
51      fields[20] = new Fleet("The Buccaneers", 4000, this);
52      fields[21] = new Fleet("Privateer armade", 4000, this
            );
53    }
54
55    /**
56     * Method that calls the landOnField method on the
            fieldNumber that the player is on.
57     *
58     * @param player The player that landed on a field.
59     */
60    public void landOnField(Player player) {
61      fields[player.getLocation()].landOnField(player);
62    }
63
64    /**
65     * Method that sets the owner to null in all the fields
            owned by a given player.
66     *
67     * @param player The player to remove.
68     */
69    public void clearFieldOwners(Player player) {
70      int i;
71      for(i = 0; i<=21; i++) {
72        if(getOwner(i) == player) {
73          ((Ownable)fields[i]).owner = null;
74        }
75      }
76    }
77
78    /**
79     * Gets the owner of a field.
80     *
81     * @param fieldNumber The number of the field to get
            owner for.
82     * @return The owner of the field.
83     */
84    public Player getOwner(int fieldNumber) {
85      if(getOwnableField(fieldNumber) != null) {
86        return getOwnableField(fieldNumber).owner;
87      }
88
89      return null;
90    }
91
```

```
92    /**
93     * Method to set a given player as owner of the field
          he is on.
94     *
95     * @param player
96     */
97    public void setOwner(Player player) {
98      getOwnableField(player.getLocation()).setOwner(player
            );
99    }
100
101   /**
102    * Gets the price of a field.
103    *
104    * @param fieldNumber The number of the field to get
          price for.
105    * @return The price of the field.
106    */
107   public int getPrice(int fieldNumber) {
108     return getOwnableField(fieldNumber).price;
109   }
110
111   /**
112    * Gets the name of a field.
113    *
114    * @param fieldNumber The number of the field to get
          name for.
115    * @return The name of the field.
116    */
117   public String getName(int fieldNumber) {
118     return fields[fieldNumber].getName();
119   }
120
121   /**
122    * Method to shake the DieCup.
123    */
124   public void shakeDieCup() {
125     dieCup.shakeDieCup();
126   }
127
128   /**
129    * Gets the sum of the values of the Dice in the DieCup
          .
130    *
131    * @return The sum of the Dice.
132    */
```

```
133     public int getDieCupSum() {
134       return dieCup.getSum();
135     }
136
137     /**
138      * Gets the value of Die1.
139      *
140      * @return The value of Die1.
141      */
142     public int getDieValue1() {
143       return dieCup.getValueDie1();
144     }
145
146     /**
147      * Gets the value of Die2.
148      *
149      * @return The value of Die2.
150      */
151     public int getDieValue2() {
152       return dieCup.getValueDie2();
153     }
154
155     /**
156      * Method to set a field.
157      *
158      * @param field Field object to insert.
159      * @param number Place number to instert field on.
160      */
161     public void setField(Field field, int number) {
162       fields[number] = field;
163     }
164
165     /**
166      * A method to generate a nice string containing the
           value of all the
167      * fields. Also contains value of the DieCup.
168      *
169      * @return All the field values as a string.
170      */
171     public String toString() {
172       String output = "";
173       int i;
174
175       for (i = 0; i < fields.length; i++) {
176         if (fields[i] != null) {
177           output = output + fields[i] + "\n";
```

```
178         }
179       }
180
181       return output + dieCup;
182     }
183
184     private Ownable getOwnableField(int fieldNumber) {
185       if (fields[fieldNumber] instanceof Ownable) {
186         return (Ownable)fields[fieldNumber];
187       }
188
189       return null;
190     }
191   }
```

14.1.8 Field - Entity

```
1   package entity;
2
3   /**
4    * Class to create a field. This class can be used to
       contain the score of a field and a value for extra
       turn.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public abstract class Field {
10    protected String name;
11
12    /**
13     * Constructor to set field name.
14     *
15     * @param name Name of field.
16     */
17    public Field(String name) {
18      this.name = name;
19    }
20
21    /**
22     * Method to get the name of the field.
23     *
24     * @return The name of the field.
25     */
26    public String getName() {
27      return name;
```

```
28    }
29
30    /**
31     * Method to take care of everything that should happen
          , when a player lands on this field.
32     * Has different implementations for different types of
          fields.
33     *
34     * @param player The player that landed on the field.
35     */
36    public abstract void landOnField(Player player);
37
38    /**
39     * Method to get content of class as a string.
40     */
41    public String toString() {
42      return name;
43    }
44  }
```

14.1.9 Ownable - Entity

```
1   package entity;
2
3   /**
4    * Class that contains all the methods and values
       relevant for ownable fields.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public abstract class Ownable extends Field {
10    protected int price;
11    protected Player owner;
12
13    /**
14     * Constructor that set name and price.
15     *
16     * @param name Name of the field.
17     * @param price Price of the field.
18     */
19    public Ownable(String name, int price) {
20      super(name);
21      this.price = price;
22      owner = null;
23    }
```

```
24
25    /**
26     * Method to take care of everything that should happen
          , when a player lands on this field.
27     * This implementation is used by all the own able
          fields that inherits from this class.
28     */
29    public void landOnField(Player player) {
30      if (owner == null) {
31        buyFieldOption(player);
32      }
33      else if(owner != player) {
34        int rent = getRent();
35        player.transferTo(owner, rent);
36      }
37    }
38
39    /**
40     * Method to calculate rent. Has different
          implementation for different types of fields.
41     *
42     * @return
43     */
44    public abstract int getRent();
45
46    /**
47     * Method to set owner of the field.
48     *
49     * @param owner The player to set as owner.
50     */
51    public void setOwner(Player owner) {
52      this.owner = owner;
53    }
54
55    private void buyFieldOption(Player player) {
56      player.setIsOnBuyableField(true);
57    }
58  }
```

14.1.10 Fleet - Entity

```
1  package entity;
2
3  /**
4   * Class to make a Fleet-field.
5   *
```

```
6   * @author DTU 02312 Gruppe 19
7   *
8   */
9  public class Fleet extends Ownable {
10   private final int[] FLEET_FIELDS = { 18, 19, 20, 21 };
11
12   private final int RENT_1 = 500;
13   private final int RENT_2 = 1000;
14   private final int RENT_3 = 2000;
15   private final int RENT_4 = 4000;
16
17   private GameBoard gameBoard;
18
19   /**
20    * Constructor that takes all inputs needed for the
          class.
21    *
22    * @param name The name of the field.
23    * @param price The price of the field.
24    * @param gameBoard The gameboard that this field is
          created in.
25    */
26   public Fleet(String name, int price, GameBoard
         gameBoard) {
27     super(name, price);
28     this.gameBoard = gameBoard;
29   }
30
31   /**
32    * Method to calculate rent for this field.
33    *
34    * @return The rent for this field.
35    */
36   public int getRent() {
37     int numberOfFleetsOwned = getFleetsOwned();
38
39     switch (numberOfFleetsOwned) {
40     case 1:
41       return RENT_1;
42     case 2:
43       return RENT_2;
44     case 3:
45       return RENT_3;
46     case 4:
47       return RENT_4;
48     default:
```

```
49        return 0;
50      }
51    }
52
53    private int getFleetsOwned() {
54      int i, numberOfFleetsOwned = 0;
55
56      for (i = 0; i < FLEET_FIELDS.length; i++) {
57        if (owner == gameBoard.getOwner(FLEET_FIELDS[i])) {
58          numberOfFleetsOwned++;
59        }
60      }
61
62      return numberOfFleetsOwned;
63    }
64  }
```

14.1.11 LaborCamp - Entity

```
1  package entity;
2
3  /**
4   * Class to make a LaborCamp field.
5   *
6   * @author DTU 02312 Gruppe 19
7   *
8   */
9  public class LaborCamp extends Ownable {
10   private final int[] LABOR_CAMP_FIELDS = { 14, 15 };
11
12   private GameBoard gameBoard;
13   private int baseRent;
14
15   /**
16    * Constructor that takes all inputs needed for the
          class.
17    *
18    * @param name The name of the field.
19    * @param baseRent The baseRent to multiply with dice
          and number of LaborCamps owned
20    * @param gameBoard The gameboard that this field is
          created in.
21    */
22   public LaborCamp(String name, int baseRent, int price,
        GameBoard gameBoard) {
23     super(name, price);
```

```
24      this.gameBoard = gameBoard;
25      this.baseRent = baseRent;
26    }
27
28    /**
29     * Method to calculate rent for this field.
30     *
31     * @return The rent for this field.
32     */
33    public int getRent() {
34      gameBoard.shakeDieCup();
35      return baseRent * gameBoard.getDieCupSum() *
            getLaborCampsOwned();
36    }
37
38    private int getLaborCampsOwned() {
39      int i, numberOfLaborCampsOwned = 0;
40
41      for (i = 0; i < LABOR_CAMP_FIELDS.length; i++) {
42        if (owner == gameBoard.getOwner(LABOR_CAMP_FIELDS[i
              ])) {
43          numberOfLaborCampsOwned++;
44        }
45      }
46
47      return numberOfLaborCampsOwned;
48    }
49  }
```

14.1.12 Refuge - Entity

```
1   package entity;
2
3   /**
4    * Class to make a Refuge-field.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class Refuge extends Field {
10    int bonus;
11
12    /**
13     * Constructor that takes all inputs needed for the
          class.
14     *
```

```
15      * @param name The name of this field.
16      * @param bonus How much the field should give as bonus
            .
17      */
18     public Refuge(String name, int bonus) {
19       super(name);
20       this.bonus = bonus;
21     }
22
23     /**
24      * Method to take care of everything that should happen
            , when a player lands on this field.
25      * Adds the bonus to the player given.
26      */
27     public void landOnField(Player player) {
28       player.addToAccount(bonus);
29     }
30 }
```

14.1.13 Tax - Entity

```
1  package entity;
2
3  import java.util.Scanner;
4
5  import boundary.TUI;
6
7  /**
8   * Class to make a Tax-field.
9   *
10  * @author DTU 02312 Gruppe 19
11  *
12  */
13 public class Tax extends Field{
14   private int taxAmount;
15   private int taxRate;
16   private GameBoard gameBoard;
17   private Scanner scanner;
18
19   /**
20    * 1 of 2 constructors. Used for fields that has only
          fixed amount of tax.
21    *
22    * @param name The name of this field.
23    * @param taxAmount The amount of tax to pay.
24    */
```

```
25    public Tax(String name, int taxAmount) {
26      super(name);
27      this.taxAmount = taxAmount;
28      taxRate = -1;
29    }
30
31    /**
32     * 2 of 2 constructors. Used for fields that has both
          fixed amount and percentage of assets as tax.
33     * Takes more arguments, to be able to get other fields
          , and to be able to ask user for fixed or
          percentage.
34     *
35     * @param name The name of this field.
36     * @param taxAmount The amount of tax to pay, if fixed
          amount i chosen.
37     * @param taxRate The percentage of assets to pay, if
          percentage is chosen.
38     * @param gameBoard The gameboard that this field is on
          .
39     * @param scanner A scanner to use for console input.
40     */
41    public Tax(String name, int taxAmount, int taxRate,
         GameBoard gameBoard, Scanner scanner) {
42      super(name);
43      this.taxAmount = taxAmount;
44      this.taxRate = taxRate;
45      this.gameBoard = gameBoard;
46      this.scanner = scanner;
47    }
48
49    /**
50     * Method to take care of everything that should happen
          , when a player lands on this field.
51     * Ask player for fixed/percentage if available, and
          subtracts score accordingly.
52     */
53    public void landOnField(Player player) {
54      int taxToPay;
55
56      if(taxRate != -1) {
57        int taxFromPct = get10PctTax(player);
58        TUI.printTaxPctChoice(taxRate, taxFromPct,
             taxAmount);
59        boolean payPct = TUI.getYesNo(scanner);
60
```

```
61          if(payPct) {
62            taxToPay = get10PctTax(player);
63          }
64          else {
65            taxToPay = taxAmount;
66          }
67        }
68        else {
69          taxToPay = taxAmount;
70        }
71
72        player.addToAccount(-1*taxToPay);
73      }
74
75      private int getAssets(Player player) {
76        int i, assets = 0;
77
78        for(i=1; i<=21; i++) {
79          if(gameBoard.getOwner(i) == player) {
80            assets = assets + gameBoard.getPrice(i);
81          }
82        }
83
84        return assets + player.getAccountValue();
85      }
86
87      private int get10PctTax(Player player) {
88          return getAssets(player) * taxRate / 100;
89      }
90  }
```

14.1.14 Territory - Entity

```
1   package entity;
2
3   /**
4    * Class to make a Territory-field.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class Territory extends Ownable {
10    private int rent;
11
12    /**
```

```
13      * Constructor that takes all inputs needed for the
            class.
14      *
15      * @param name The name of this field.
16      * @param rent The rent a player should pay if landing
            on this field.
17      * @param price The price of this field.
18      */
19     public Territory(String name, int rent, int price) {
20       super(name, price);
21       this.rent = rent;
22     }
23
24     /**
25      * Method to calculate rent for this field.
26      *
27      * @return The rent for this field.
28      */
29     public int getRent() {
30       return rent;
31     }
32 }
```

14.1.15 DieCup - Entity

```
1  package entity;
2
3  /**
4   * Class to create a diecup. This class will take in 2
        dice.
5   *
6   * @author DTU 02312 Gruppe 19
7   *
8   */
9  public class DieCup {
10     private Die die1, die2;
11
12     /**
13      * Constructor that set up two new dice.
14      */
15     public DieCup() {
16         die1 = new Die();
17         die2 = new Die();
18     }
19
20     /**
```

```
21          * Method to shake the diecup.
22          */
23         public void shakeDieCup() {
24             die1.roll();
25             die2.roll();
26         }
27
28         /**
29          * Adds value from die1 and die2.
30          *
31          * @return The sum of die1 and die2.
32          */
33         public int getSum() {
34             return die1.getValue() + die2.getValue();
35         }
36
37         /**
38          * Get faceValue from die1
39          *
40          * @return The current value of die1.
41          */
42         public int getValueDie1() {
43             return die1.getValue();
44         }
45
46         /**
47          * Get faceValue from die2
48          *
49          * @return The current value of die2.
50          */
51         public int getValueDie2() {
52             return die2.getValue();
53         }
54
55         /**
56          * Checks if the values of the dice are equal.
57          *
58          * @return True if facevalues of die1 and die2 are
                the same, otherwise False.
59          */
60         public boolean getEns() {
61             if (die1.getValue() == die2.getValue()) {
62                 return true;
63             }
64             return false;
65         }
```

```
66
67      /**
68       * Method that makes a text with the most important
             values in the class, and some description.
69       *
70       * @return A coherent string with values of Die1 and
             Die2
71       */
72      public String toString() {
73          return "Die1 = " + die1 + ", Die2 = " + die2;
74      }
75  }
```

14.1.16 Die - Entity

```
1   package entity;
2
3   import java.util.Random;
4
5   /**
6    * Class to create a die. This class can be used to
         generate random numbers from 1 to 6.
7    *
8    * @author DTU 02312 Gruppe 19
9    *
10   */
11  public class Die {
12
13      private final int MAX_VALUE = 6;
14      private int faceValue;
15      private Random random;
16
17      /**
18       * Constructor to set the facevalue of the die and
             instanciate the random generator.
19       */
20      public Die() {
21          random = new Random();
22          faceValue = 0;
23      }
24
25      /**
26       * Method to roll the die, and give it a new value.
27       */
28      public void roll() {
```

```
29          //Generator makes numbers from 0 to 5, so we add
                1 to get 1 to 6
30          faceValue = random.nextInt(MAX_VALUE) + 1;
31      }
32
33      /**
34       * Method so you can get the facevalue.
35       *
36       * @return The current facevalue of the die.
37       */
38      public int getValue() {
39          return faceValue;
40      }
41
42      /**
43       * Method that makes a string of facevalue to print.
44       *
45       * @return The current facevalue of the die as a
                string.
46       */
47      public String toString() {
48          return Integer.toString(faceValue);
49      }
50  }
```

14.1.17 DieCupTestEntity - TestTools

```
1   package testTools;
2
3   /**
4    * Class to create a "cheating" diecup. This class will
         not actually use dice, but just give a hardcoded
         number every time.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class DieCupTestEntity {
10    private final int VALUE_TO_GIVE1 = 1;
11    private final int VALUE_TO_GIVE2 = 2;
12
13    /**
14     * Constructor
15     */
16      public DieCupTestEntity() {
17
```

```
18      }
19
20      /**
21       * Method to shake the diecup.
22       * This won't do anything in this class, since the
             values are final. It just have to be there, to
             make
23       * it possible to substitute the "real" DieCup with
             this "false" one.
24       */
25      public void shakeDieCup() {
26        //Does nothing, since the values are fixed
27      }
28
29      /**
30       * Adds value from die1 and die2.
31       *
32       * @return The sum of die1 and die2.
33       */
34      public int getSum() {
35          return VALUE_TO_GIVE1 + VALUE_TO_GIVE2;
36      }
37
38      /**
39       * Get faceValue from die1
40       *
41       * @return The current value of die1.
42       */
43      public int getValueDie1() {
44          return VALUE_TO_GIVE1;
45      }
46
47      /**
48       * Get faceValue from die2
49       *
50       * @return The current value of die2.
51       */
52      public int getValueDie2() {
53          return VALUE_TO_GIVE2;
54      }
55
56      /**
57       * Checks if the values of the dice are equal.
58       *
59       * @return True if facevalues of die1 and die2 are
             the same, otherwise False.
```

```
60          */
61         public boolean getEns() {
62             if (VALUE_TO_GIVE1 == VALUE_TO_GIVE2) {
63                 return true;
64             }
65             return false;
66         }
67 
68         /**
69          * Method that makes a text with the most important
                values in the class, and some description.
70          *
71          * @return A coherent string with values of Die1 and
                Die2
72          */
73         public String toString() {
74             return "Die1 = " + VALUE_TO_GIVE1 + ", Die2 = " +
                    VALUE_TO_GIVE2;
75         }
76 }
```

14.1.18 FleetTest - TestTools

```
1  package testTools;
2  
3  import org.junit.After;
4  import org.junit.Assert;
5  import org.junit.Before;
6  import org.junit.Test;
7  
8  import entity.Fleet;
9  import entity.GameBoard;
10 import entity.Player;
11 
12 public class FleetTest {
13   private Player player;
14   private Player owner;
15   private GameBoard gameBoard;
16 
17   @Before
18   public void setUp() throws Exception {
19     this.gameBoard = new GameBoard(22);
20     this.player = new Player(5000, "Anders And");
21     this.owner = new Player(1000, "Andersine");
22 
```

```
23      this.gameBoard.setField(new Fleet("Fleet1", 0,
            gameBoard), 18);
24      this.gameBoard.setField(new Fleet("Fleet2", 0,
            gameBoard), 19);
25      this.gameBoard.setField(new Fleet("Fleet3", 0,
            gameBoard), 20);
26      this.gameBoard.setField(new Fleet("Fleet4", 0,
            gameBoard), 21);
27
28      this.owner.setLocation(18);
29      this.gameBoard.setOwner(owner);
30
31      this.player.setLocation(18);
32    }
33
34    @After
35    public void tearDown() throws Exception {
36      this.player = new Player(5000, "Anders And");
37      this.owner = new Player(1000, "Andersine");
38      this.gameBoard.setField(new Fleet("Fleet2", 0,
            gameBoard), 19);
39      this.gameBoard.setField(new Fleet("Fleet3", 0,
            gameBoard), 20);
40      this.gameBoard.setField(new Fleet("Fleet4", 0,
            gameBoard), 21);
41    }
42
43    @Test
44    public void testEntities() {
45      Assert.assertNotNull(this.player);
46      Assert.assertNotNull(this.owner);
47      Assert.assertNotNull(this.gameBoard);
48    }
49
50    @Test
51    public void testLandOnField1Owned() {
52      int expected = 5000;
53      int actual = this.player.getAccountValue();
54      Assert.assertEquals(expected, actual);
55
56      // Perform the action to be tested
57      this.gameBoard.landOnField(this.player);
58      expected = 5000 - 500;
59      actual = this.player.getAccountValue();
60      Assert.assertEquals(expected, actual);
61    }
```

```
62
63      @Test
64      public void testLandOnField2Owned() {
65        int expected = 5000;
66        int actual = this.player.getAccountValue();
67        Assert.assertEquals(expected, actual);
68
69        // Perform the action to be tested
70        this.owner.setLocation(19);
71        this.gameBoard.setOwner(owner);
72
73        this.gameBoard.landOnField(this.player);
74        expected = 5000 - 1000;
75        actual = this.player.getAccountValue();
76        Assert.assertEquals(expected, actual);
77      }
78
79      @Test
80      public void testLandOnField3Owned() {
81        int expected = 5000;
82        int actual = this.player.getAccountValue();
83        Assert.assertEquals(expected, actual);
84
85        // Perform the action to be tested
86        this.owner.setLocation(19);
87        this.gameBoard.setOwner(owner);
88        this.owner.setLocation(20);
89        this.gameBoard.setOwner(owner);
90
91        this.gameBoard.landOnField(this.player);
92        expected = 5000 - 2000;
93        actual = this.player.getAccountValue();
94        Assert.assertEquals(expected, actual);
95      }
96
97      @Test
98      public void testLandOnField4Owned() {
99        int expected = 5000;
100       int actual = this.player.getAccountValue();
101       Assert.assertEquals(expected, actual);
102
103       // Perform the action to be tested
104       this.owner.setLocation(19);
105       this.gameBoard.setOwner(owner);
106       this.owner.setLocation(20);
107       this.gameBoard.setOwner(owner);
```

```
108       this.owner.setLocation(21);
109       this.gameBoard.setOwner(owner);
110
111       this.gameBoard.landOnField(this.player);
112       expected = 5000 - 4000;
113       actual = this.player.getAccountValue();
114       Assert.assertEquals(expected, actual);
115     }
116   }
```

14.1.19 LaborCampTest - TestTools

```
1   package testTools;
2
3   import org.junit.After;
4   import org.junit.Assert;
5   import org.junit.Before;
6   import org.junit.Test;
7
8   import entity.LaborCamp;
9   import entity.GameBoard;
10  import entity.Player;
11
12  public class LaborCampTest {
13    private Player player;
14    private Player owner;
15    private GameBoard gameBoard;
16
17    @Before
18    public void setUp() throws Exception {
19      this.gameBoard = new GameBoard(16);
20      this.player = new Player(5000, "Anders And");
21      this.owner = new Player(1000, "Andersine");
22
23      this.gameBoard.setField(new LaborCamp("LaborCamp1",
            100, 0, gameBoard), 14);
24      this.gameBoard.setField(new LaborCamp("LaborCamp2",
            100, 0, gameBoard), 15);
25      this.gameBoard.shakeDieCup();
26
27      this.owner.setLocation(14);
28      this.gameBoard.setOwner(owner);
29
30      this.player.setLocation(14);
31    }
32
```

```
33    @After
34    public void tearDown() throws Exception {
35      this.player = new Player(5000, "Anders And");
36      this.owner = new Player(1000, "Andersine");
37      this.gameBoard.setField(new LaborCamp("LaborCamp2",
            100, 0, gameBoard), 15);
38    }
39
40    @Test
41    public void testEntities() {
42      Assert.assertNotNull(this.player);
43      Assert.assertNotNull(this.owner);
44      Assert.assertNotNull(this.gameBoard);
45    }
46
47    @Test
48    public void testLandOnField1Owned() {
49      int expected = 5000;
50      int actual = this.player.getAccountValue();
51      Assert.assertEquals(expected, actual);
52
53      // Perform the action to be tested
54      this.gameBoard.landOnField(this.player);
55      expected = 5000 - (1 * 100 * this.gameBoard.
            getDieCupSum());
56      actual = this.player.getAccountValue();
57      Assert.assertEquals(expected, actual);
58    }
59
60    @Test
61    public void testLandOnField2Owned() {
62      int expected = 5000;
63      int actual = this.player.getAccountValue();
64      Assert.assertEquals(expected, actual);
65
66      // Perform the action to be tested
67      this.owner.setLocation(15);
68      this.gameBoard.setOwner(owner);
69
70      this.gameBoard.landOnField(this.player);
71      expected = 5000 - (2 * 100 * this.gameBoard.
            getDieCupSum());
72      actual = this.player.getAccountValue();
73      Assert.assertEquals(expected, actual);
74    }
75  }
```

14.1.20 RefugeTest - TestTools

```
1  package testTools;
2
3  import org.junit.After;
4  import org.junit.Assert;
5  import org.junit.Before;
6  import org.junit.Test;
7
8  import entity.Field;
9  import entity.Player;
10 import entity.Refuge;
11
12 public class RefugeTest {
13   private Player player;
14   private Field refuge200;
15   private Field refuge0;
16
17   @Before
18   public void setUp() throws Exception {
19     this.player = new Player(1000, "Anders And");
20     this.refuge200 = new Refuge("Helle +200", 200);
21     this.refuge0 = new Refuge("Helle 0", 0);
22   }
23
24   @After
25   public void tearDown() throws Exception {
26     this.player = new Player(1000, "Anders And");
27     // The fields are unaltered
28   }
29
30   @Test
31   public void testEntities() {
32     Assert.assertNotNull(this.player);
33     Assert.assertNotNull(this.refuge200);
34     Assert.assertNotNull(this.refuge0);
35     Assert.assertTrue(this.refuge200 instanceof Refuge);
36     Assert.assertTrue(this.refuge0 instanceof Refuge);
37   }
38
39   @Test
40   public void testLandOnField200() {
41     int expected = 1000;
42     int actual = this.player.getAccountValue();
43     Assert.assertEquals(expected, actual);
44
```

```
45      // Perform the action to be tested
46      this.refuge200.landOnField(this.player);
47      expected = 1000 + 200;
48      actual = this.player.getAccountValue();
49      Assert.assertEquals(expected, actual);
50    }
51
52    @Test
53    public void testLandOnField200Twice() {
54
55      int expected = 1000;
56      int actual = this.player.getAccountValue();
57      Assert.assertEquals(expected, actual);
58
59      // Perform the action to be tested
60      this.refuge200.landOnField(this.player);
61      this.refuge200.landOnField(this.player);
62      expected = 1000 + 200 + 200;
63      actual = this.player.getAccountValue();
64      Assert.assertEquals(expected, actual);
65    }
66
67    @Test
68    public void testLandOnField0() {
69      int expected = 1000;
70      int actual = this.player.getAccountValue();
71      Assert.assertEquals(expected, actual);
72
73      // Perform the action to be tested
74      this.refuge0.landOnField(this.player);
75      expected = 1000;
76      actual = this.player.getAccountValue();
77      Assert.assertEquals(expected, actual);
78    }
79
80    @Test
81    public void testLandOnField0Twice() {
82      int expected = 1000;
83      int actual = this.player.getAccountValue();
84      Assert.assertEquals(expected, actual);
85
86      // Perform the action to be tested
87      this.refuge0.landOnField(this.player);
88      this.refuge0.landOnField(this.player);
89      expected = 1000;
90      actual = this.player.getAccountValue();
```

```
91      Assert.assertEquals(expected, actual);
92    }
93  }
```

14.1.21 TaxTest - TestTools

```
1   package testTools;
2
3   import org.junit.After;
4   import org.junit.Assert;
5   import org.junit.Before;
6   import org.junit.Test;
7
8   import entity.Field;
9   import entity.Player;
10  import entity.Tax;
11
12  public class TaxTest {
13    private Player player;
14    private Field tax200;
15    private Field tax0;
16
17    @Before
18    public void setUp() throws Exception {
19      this.player = new Player(1000, "Anders And");
20      this.tax200 = new Tax("Helle +200", 200);
21      this.tax0 = new Tax("Helle 0", 0);
22    }
23
24    @After
25    public void tearDown() throws Exception {
26      this.player = new Player(1000, "Anders And");
27      // The fields are unaltered
28    }
29
30    @Test
31    public void testEntities() {
32      Assert.assertNotNull(this.player);
33      Assert.assertNotNull(this.tax200);
34      Assert.assertNotNull(this.tax0);
35      Assert.assertTrue(this.tax200 instanceof Tax);
36      Assert.assertTrue(this.tax0 instanceof Tax);
37    }
38
39    @Test
40    public void testLandOnField200() {
```

```
41      int expected = 1000;
42      int actual = this.player.getAccountValue();
43      Assert.assertEquals(expected, actual);
44
45      // Perform the action to be tested
46      this.tax200.landOnField(this.player);
47      expected = 1000 - 200;
48      actual = this.player.getAccountValue();
49      Assert.assertEquals(expected, actual);
50    }
51
52    @Test
53    public void testLandOnField200Twice() {
54
55      int expected = 1000;
56      int actual = this.player.getAccountValue();
57      Assert.assertEquals(expected, actual);
58
59      // Perform the action to be tested
60      this.tax200.landOnField(this.player);
61      this.tax200.landOnField(this.player);
62      expected = 1000 - 200 - 200;
63      actual = this.player.getAccountValue();
64      Assert.assertEquals(expected, actual);
65    }
66
67    @Test
68    public void testLandOnField0() {
69      int expected = 1000;
70      int actual = this.player.getAccountValue();
71      Assert.assertEquals(expected, actual);
72
73      // Perform the action to be tested
74      this.tax0.landOnField(this.player);
75      expected = 1000;
76      actual = this.player.getAccountValue();
77      Assert.assertEquals(expected, actual);
78    }
79
80    @Test
81    public void testLandOnField0Twice() {
82      int expected = 1000;
83      int actual = this.player.getAccountValue();
84      Assert.assertEquals(expected, actual);
85
86      // Perform the action to be tested
```

```
87      this.tax0.landOnField(this.player);
88      this.tax0.landOnField(this.player);
89      expected = 1000;
90      actual = this.player.getAccountValue();
91      Assert.assertEquals(expected, actual);
92    }
93  }
```

14.1.22 TerritoryTest - TestTools

```
1   package testTools;
2
3   import org.junit.After;
4   import org.junit.Assert;
5   import org.junit.Before;
6   import org.junit.Test;
7
8   import entity.Field;
9   import entity.Ownable;
10  import entity.Player;
11  import entity.Territory;
12
13  public class TerritoryTest {
14    private Player player;
15    private Player owner;
16    private Field ter200;
17    private Field ter0;
18
19    @Before
20    public void setUp() throws Exception {
21      this.player = new Player(1000, "Anders And");
22      this.owner = new Player(1000, "Andersine");
23      this.ter200 = new Territory("Territory +200", 200,
          200);
24      this.ter0 = new Territory("Territory 0", 0, 0);
25      ((Ownable)ter200).setOwner(owner); //Cast to ownable
          to be able to set owner
26      ((Ownable)ter0).setOwner(owner); //Cast to ownable to
            be able to set owner
27    }
28
29    @After
30    public void tearDown() throws Exception {
31      this.player = new Player(1000, "Anders And");
32      this.owner = new Player(1000, "Andersine");
33      // The fields are unaltered
```

```
34    }
35
36    @Test
37    public void testEntities() {
38      Assert.assertNotNull(this.player);
39      Assert.assertNotNull(this.ter200);
40      Assert.assertNotNull(this.ter0);
41      Assert.assertTrue(this.ter200 instanceof Territory);
42      Assert.assertTrue(this.ter0 instanceof Territory);
43    }
44
45    @Test
46    public void testLandOnField200() {
47      int expected = 1000;
48      int actual = this.player.getAccountValue();
49      Assert.assertEquals(expected, actual);
50
51      // Perform the action to be tested
52      this.ter200.landOnField(this.player);
53      expected = 1000 - 200;
54      actual = this.player.getAccountValue();
55      Assert.assertEquals(expected, actual);
56    }
57
58    @Test
59    public void testLandOnField200Twice() {
60
61      int expected = 1000;
62      int actual = this.player.getAccountValue();
63      Assert.assertEquals(expected, actual);
64
65      // Perform the action to be tested
66      this.ter200.landOnField(this.player);
67      this.ter200.landOnField(this.player);
68      expected = 1000 - 200 - 200;
69      actual = this.player.getAccountValue();
70      Assert.assertEquals(expected, actual);
71    }
72
73    @Test
74    public void testLandOnField0() {
75      int expected = 1000;
76      int actual = this.player.getAccountValue();
77      Assert.assertEquals(expected, actual);
78
79      // Perform the action to be tested
```

```
80      this.ter0.landOnField(this.player);
81      expected = 1000;
82      actual = this.player.getAccountValue();
83      Assert.assertEquals(expected, actual);
84    }
85
86    @Test
87    public void testLandOnField0Twice() {
88      int expected = 1000;
89      int actual = this.player.getAccountValue();
90      Assert.assertEquals(expected, actual);
91
92      // Perform the action to be tested
93      this.ter0.landOnField(this.player);
94      this.ter0.landOnField(this.player);
95      expected = 1000;
96      actual = this.player.getAccountValue();
97      Assert.assertEquals(expected, actual);
98    }
99  }
```