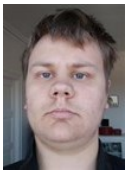
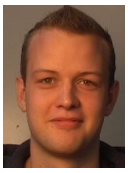
Rapport - CDIO3 Terningespil 3

Peter Revsbech (s183760)



Mathias Bærentzen (s176360)

Sulaiman Kasas (s195462)



Christian Kyed (s184210)

Bashar Bdewi (s183356)

Derar Almosawe (s181553)

Link til GitHub-repo: https://github.com/CKyed/11_del3.git

Rapport i kurset Versionsstyring og Testmetoder 02315
Danmarks Tekniske Universitet
Afleveringsfrist: 01. November 2019

Indhold

1 Abstract

This report describes the work done on the CDIO project part 3 by group 11, which is a software-project written in the Java language. The project is a ”Juniormatador-game played by two to four players. First, the requirements of the product are described, mainly through a use case analysis of the final product. By analyzing use cases like ”Start game”, and ”landed on property”as well as the product description of the real ”Juniormatador-game”, the group has descriped the requirements of the system and divided them in four categories: One that is related to the chancecards, one related to the fields, one that is general requirements of the system, and finally the non-functional requirements. The requirements were reconsidered throughout the whole project.
After this comes the analysis-section, describing the domain-model of the system, mainly through domain-classdiagram. This diagram shows the central classes of a Juniormatador-game such as ”Game”, ”Player”, ”Board”and ”Field”.
Then comes the Design-section describing the process of planning how to make the software classes and methods based on the domain model. This results in a very detalied classdiagram of the software classes.
The last part of the report contains notes on the Project management as well as the tests that were made to check the liability system. There were made several tests, and these were divided into two categories: Automated and manual tests. In the end it is concluded, that the project lives up to all the important requirements as far as the developers know.

2 Indledning

Vi er blevet stillet til opgave at udarbejde et Juniormatador-program i Java. Opgaven kommer som en naturlig forlængelse af vores tidligere CDIO-opgaver, hvor vi har udviklet forskellige stykker software, som kan bruges i forbindelse med udviklingen af dette program, f.eks. de virtuelle terninger og spiller-klassen. Derudover kan opgaven anses for at være en slags generalprøve inden gruppen i januar skal udvikle et lignende projekt, som skal være et egentligt matador-spil i forbindelse med vores eksamen i Indledende Programmering.
Juniormatador er et simpelt brætspil, som børn skal have lettere ved at spille end det rigtige matador. Spillet har nogle overordnet meget tydelige og entydige regler, som på visse punkter gør det ideelt projekt til at lære at programmere. Til gengæld er det også et temmelig omfattende spil med f.eks. det store antal chancekort og felter. Noget af det sværeste ved projektet har derfor vist sig at være, hvordan systemets struktur skulle planlæggs og opbygges på en hensigtsmæssig måde, så den både er forståelig, oveskuelig, velfungerende og kan genbruges til det næste matadorprojekt i januar.

3 Krav

3.1 Use cases

Vores arbejde tager udgangspunkt i nogle use cases. Vi vil gerne forestille os, hvordan en bruger skal kunne benytte vores færdige program i nogle forskellige scenarier. Idet spillet juniormatador har meget lidt valgfrihed til spillerne, er der ikke specielt mange usecases i det hele taget. Vores usecases har vi beskrevet hver især med en tilhørende usecase-tekst. Disse kan ses i figur 1.

Her har vi opstillet 4 usecases hvor flere af dem ”extender”fra en af de ovenstående usecases. UC1-”Start spil”usecasen beskriver hvilke informationer en spiller skal oplyse for at der kan startes en spil. Det er f.eks nødvendigt for at spillet kan spilles korrekt at spilleren oplyser hvem der er yngst og efterfølgende hvilken rækkefølge spillerne sidder i, for at rækkefølgen på de efterfølgende ture kan være korrekt. UC2 er en meget generel usecases hvor man herfra ville kunne forlænge flere usecase. UC3 forlænger fra UC2, hvor vi tager udgagnspunkt i at en spiller, hvis tur er i gang med at blive spillet. Her lander en spiller på en ejendom og der er nu flere muligheder for hvad der fremadrettet vil ske. F.eks hvis en spiller lander på en ejendom vil en spiller enten skulle købe ejendommen eller

ID	Beskrivelse
UC1	Start spil: **1.** Systemet beder spiller om antallet af spillere. **2.** Spillerne angiver antallet. **3.** Systemet beder om spillernes navne. **4.** Spillerne skriver deres navne. **5.** Systemet beder spillerne inputte den nødvendige information for at definere rækkefølgen af turene i spillet. **6.** Spillerne inputter dette.
UC2	Spil tur: 1. Den aktive spiller kaster terningen og rykker det antal felter som øjnene viser. 2. Alt efter hvilken type felt, spilleren er landet på, sker følgende: a. En ejendom b. Et chancefelt: Spilleren tager det øverste chancekort i bunken og læser det. Spilleren gør det der står på kortet. c. Parkering: Der sker ikke noget. d. Besøg i fængsel: Der sker ikke noget. e. Gå i fængsel: Spilleren rykket direkte til fængslet og modtager ikke M2. Næste gang det er spillerens tur, betales enten en bøde på M1 eller med et løsladelseskort. f. START: Spilleren modtager M2 3. Turen går videre til næste spiller
UC3	Spilleren er landet på en ejendom 1. Alt efter om ejendommen allerede er ejet eller ej, sker følgende: a. Ikke ejet: Spilleren betaler ejendommens pris til banken, og ejer nu ejendommen. b. Ejet: Spilleren betaler ejendommens pris til ejeren. 2. Hvis en spiller har 0 eller færre penge tilbage, har spilleren tabt og UC4 starter. 3. Det tjekkes om en spiller er gået fallit
UC4	Tjek for fallit: 1. Det tjekkes, om spilleren skal betale mere end spilleren ejer. 2. Hvis spilleren har nok penge til at betale, gør han det. 3. Hvis ikke spilleren har nok penge til at betale, betaler han det sidste han har, og går fallit. Spillet er nu slut 4. Spillet tjekker hvilken spiller der har flest penge tilbage. Denne udnævnes som vinder. 5. Hvis to eller flere spillere har lige mange penge tilbage udnævnes flere vindere.

Figur 1: Use case tekst-beskrivelser

Use Case Section	Comment
Use Case navn	Start spil
Primær aktør	Player vil gerne spille spillet
Pre-conditions	Spillerne har åbnet spillet
Succeskriterier/ postconditions	Systemet kender spillernes navne og rækkefølgen, turene skal forløbe i.
Main flow	1. Systemet beder spiller om antallet af spillere. 2. Spillerne angiver antallet. 3. Systemet beder om spillernes navne. 4. Spillerne skriver deres navne. 5. Systemet beder spillerne inputte den nødvendige information for at definere rækkefølgen af turene i spillet. 6. Spillerne inputtet dette.
Alternative flows	1.Systemet beder spillerne om at trykke OK for at starte spillet. 2. Systemet beder spiller om antallet af spillere. 3. Spillerne angiver antallet. 4. Systemet beder om spillernes navne. -Hvis spilleren indtaster et tomt navn eller hvis spilleren indtaste det samme navn som en anden spiller, så får han en fejlmeddelelse ellers: dette gentages indtil alle spillere har indtastet navn. 5. Systemet beder spillerne om at vælge den yngste spiller og hvem der sidder til venstre for, indtil placeringen af alle spillere kendes.

Figur 2: Fully Dressed version af UC1, Start spil

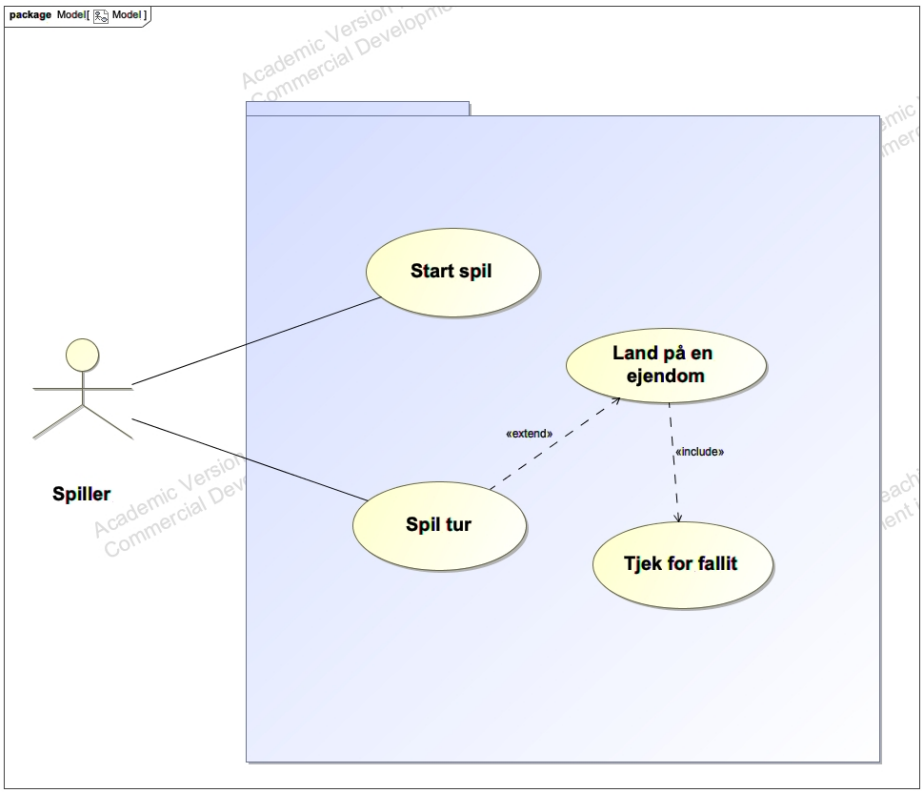
betale husleje til ejeren af ejendommen. Hvis spilleren ikke har nok kapital i banken til enten at betale husleje eller købe ejendommen, vil spileren derfor gå fallit. UC4 er igen en forlængelsen af UC3. UC3 tager nemlig udgangs punkt i den case for hvad der skal ske hvis spiller enten har, eller ikke har, valuta nok.

Vi har valgt at beskrive UC1, Start Spil i en fully dressed version, for at få et bedre overblik over hvordan forløbet af denne skal være. Vores fully dressed beskrivelse kan ses i figur 2. Her har vi taget højde for, hvad der skal ske i de alternative flows, dvs. ikke kun hvis alt går godt.

Vi har valgt at beskrive vores usecases og deres relation til hinanden i et usecasediagram, som ses i figur 3. Her har vi vores Spiller som aktør og har taget et udsnit af de usecases en spiller kan have, som feks ”Start spil”og ”Spil tur”. ”Spil tur”er en meget stor usecase og derfor har vi taget et udsnit af usecasene med som forlænger fra Spil tur, fx ”Land på ejendom”som igen bliver forlænget med ”Tjek for falit”

3.2 Kravbeskrivelser

Analysen af systemet krav har taget udgangspunkt i vores usecases, i den stillede opgavebeskrivelse samt i de officielle Juniormatador-regler. For overskuelighedens skyld, har vi valgt at opdele kravene i de funktionelle og de ikke-funktionelle. De funktionelle krav, har vi yderligere opdelt i flere



Figur 3: Usecase diagram

underkategogrier.

Den første kategori af funktionelle krav, vi har valgt at kigge på er de ”generelle krav”. Det er de krav, som ikke faldt indenfor en af de yderligere kategorier, og som er generelle for hele systemet. Disse kan ses i figur 4. Her kan det bemærkes at kravet KG9 specificerer, at hvis flere spillere står lige om førstepladsen, kåres der flere vindere. Dette har vi valgt at gøre, idet vi pga. tidsmangel ikke kunne implementere at spillernes grundes pris tages med i beregneningen af hvem, der har vundet, når det står lige.

Den næste kategori af funktionelle krav er ”Krav til felter”. Disser omhandler de krav, der er til funktionaliteten af spillets felter, og listen kan ses i figur 5. Disse kravsspecifikationer er stort set udarbejdet direkte fra regelsættet af juniormatador.

Den sidste kategori af funktionelle krav er ”Krav til Chancekort”. Disse omhandler spillets chancekort og kan ses i figur 6. KC1 har vi lavet, idet der er 4 chancekort i juniormatador, hvori en spiller skal give chancekortet til en bestemt anden spiller. Hvis der eksempelvis er 2 spillere i spillet, den blå og den grønne, er det et problem at den blå bliver bedt om at give kortet til den røde, når den røde ikke er med. Det har vi valgt at takle ved at oprette chancekort-dækket med kun de kort, som er relevante for det aktuelle antal spillere.

Desuden har vi i KC4 specificeret, hvad der skal ske, hvis en spiller bliver bedt om at rykke til et ledigt felt, når der ingen ledige felter er. I de officielle juniormatador-regler står der, at spilleren skal købe en grund af en anden spiller i dette tilfælde. Det synes vi er en underlig og meget tvetydig regel. Det fremgår ikke, hvem der skal sælge sin grund, eller hvad prisen skal være, eller hvad der skal ske, hvis ingen spillere ønsker at sælge. Vi har derfor valgt at gøre, så spilleren bare rykker til start, hvis der ingen ledige felter er, og han skal rykke til et ledigt felt.

Den sidste kategori af krav er de ikke-funktionelle krav. Disse krav kan ikke direkte ses ud af vores usecases, men er alligevel nødvendige for at spillet skal kunne fungere som helhed. Listen kan ses i figur 7. Disse krav er blevet opstillet på baggrund af opgavebeskrivelsen. Konfigurationsstying og tests vil vi komme mere ind på senere i rapporten.

ID	Beskrivelse
KG1	Spillet skal kunne spilles af 2-4 spillere.
KG2	Spillerne skal kunne vælge deres egne navne, og disse må ikke være ens.
KG3	Rækkefølgen af ture starter med den yngste spiller og derefter fortsætte mod venstre, dvs. med uret, hvor de sidder.
KG4	Spillerne skiftes til at spille en tur på skift indtil spillets afslutning.
KG5	Spillernes placeringer skal være repræsenteret af hver deres farvede bil på brættet.
KG6	På brættet skal det ses, hvad spillernes balancer er.
KG7	Spillet er slut, når en spiller skal betale et beløb som er større end spillerens balance.
KG8	Ved spillets afslutning betaler den tabende så meget af sin gæld af som muligt med det spilleren har tilbage.
KG9	Ved spillets afslutning kåres vinderen som den spiller, der har den højeste balance. Hvis to eller flere står lige, kåres flere vindere.
KG10	Det skal være muligt at spille igen ved spillets afslutning.

Figur 4: De generelle krav

ID	Beskrivelse
KF1	Felterne skal repræsentere de officielle felter i Juniormatador.
KF2	Felternes navn og evt. pris skal vises på brættet.
KF3	Det skal være muligt, at se hvem ejeren er af et ejendoms-felt.
KF4	Spillerne skal kunne købe ejendoms-felterne hvis de er ledige.
KF5	Spillerne skal betale husleje til ejeren af et ejedomsfelt.
KF6	Huslejen for et bestemt felt skal være den samme hele spillet igennem, og er altså uændret også når to ejendomme i samme farve ejes.
KF7	Hvis start-feltet passeres eller landes på, modtager spilleren M2.
KF8	Hvis en spiller landet på “gå i fængsel” bevæger han sig til fængslet og modtager ikke M2 for at passere start.
Kf9	Hvis en spiller lander på et chance-felt, trækker spilleren et chancekort. Det videre forløb beskrives af kortet.
KF10	Hvis en spiller landet på “gratis parkering” eller “besøg i fængsel” sker ingenting.

Figur 5: Krav til felter

ID	Beskrivelse
KC1	Der skal være 20 chancekort i spillet, hvis der er 4 spillere med. Hvis der kun er 2 eller 3 spillere, skal hhv. et eller to chancekort udelades.
KC2	Chancekortene skal blandes tilfældigt ved spillets start.
KC3	Når et chancekort har været i brug, lægges det nederst i bunken.
KC4	Hvis chancekortet, hvor en spiller rykker til et ledigt felt trækkes, når ingen felter er ledige, rykker spillere bare til start.
KC5	Chancekortene skal vises på skærmen, når de trækkes.
KC6	Chancekortene skal repræsentere de officielle chancekort i Juniormatador.

Figur 6: Krav til chancekort

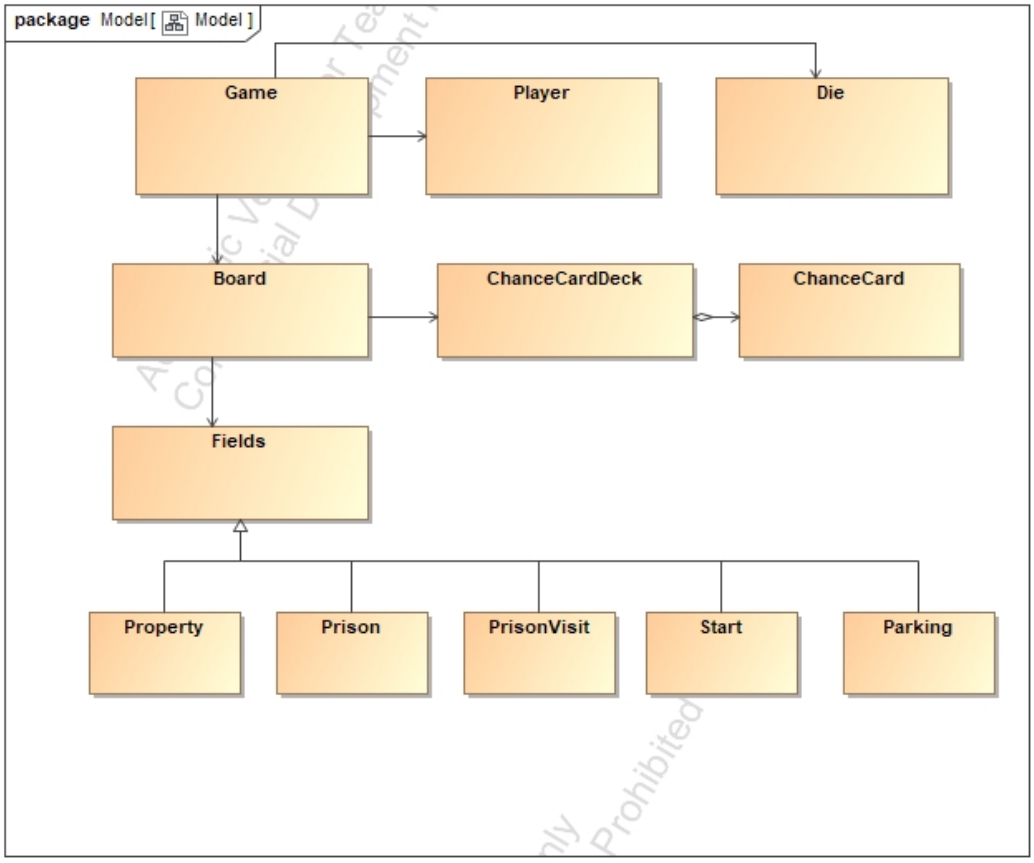
ID	Beskrivelse
KC1	Der skal være 20 chancekort i spillet, hvis der er 4 spillere med. Hvis der kun er 2 eller 3 spillere, skal hhv. et eller to chancekort udelades.
KC2	Chancekortene skal blandes tilfældigt ved spillets start.
KC3	Når et chancekort har været i brug, lægges det nederst i bunken.
KC4	Hvis chancekortet, hvor en spiller rykker til et ledigt felt trækkes, når ingen felter er ledige, rykker spillere bare til start.
KC5	Chancekortene skal vises på skærmen, når de trækkes.
KC6	Chancekortene skal repræsentere de officielle chancekort i Juniormatador.

Figur 7: Ikke-funktionelle krav

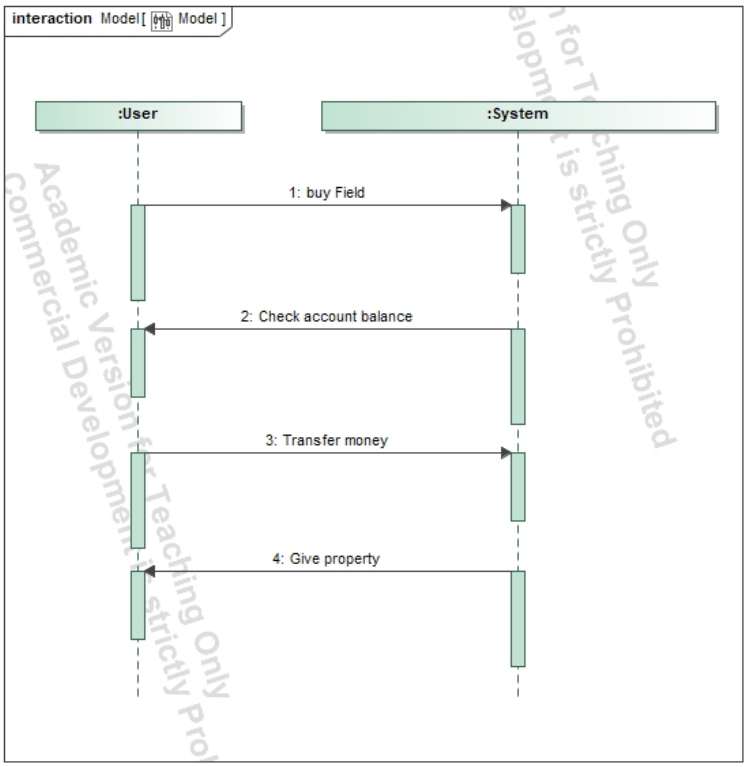
4 Analyse

På baggrund af vores kravsspecifikationer og usecases, har vi udarbejdet en domænemodel for et juniormatadorspil. Vi har i vores usecases bemærket at visse navneord opstår mange gange og vil være oplagte klasser. Eksempelvis, et "spil", en "spiller"og et "bræt". Ud fra disse koncepter fra den virkelige verden, har vi opstillet vores domæne-klassediagram. Den højeststående klasse i diagrammet er Game. Game har kendskab til et Board, nogle Players og en Die. Vi tænker, at Board også har kendskab til et antal Field-objekter. Vi forestiller os at Field kan være superklasse til et antal nedarvede Field-typer, der tilsvarer de forskellige typer felter, man kan lande på i Juniormatador.

Vi har lavet et systemsekvensdiagram hvor tager udgangspunkt i en spiller som køber en ejendom. Systemsekvensdiagrammet kan ses i figur 9. Her kan man se at når spilleren forsøger at købe et felt, tjekker systemet først om spilleren har råd, for derefter at overføre pengene og tildele spilleren ejendommen. Det er derfor tydeligt at systemet skal kunne sikre sig at den kun tildeler ejendomme til spiller som har den nødvendige valuta.



Figur 8: Domæne-klassediagram

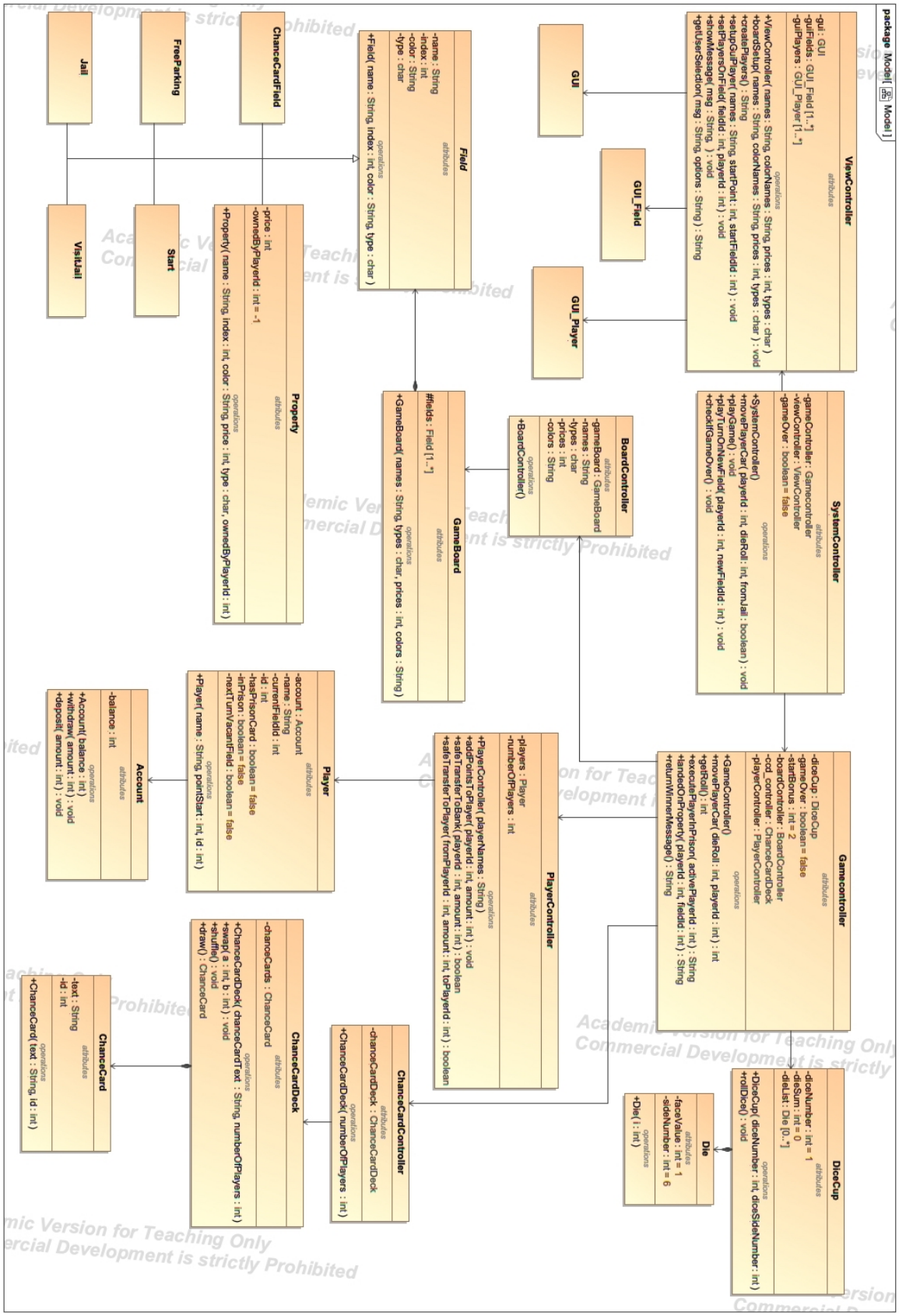


Figur 9: Systemsekvensdiagram over forløbet, når en spiller skal købe en ejendom.

5 Design

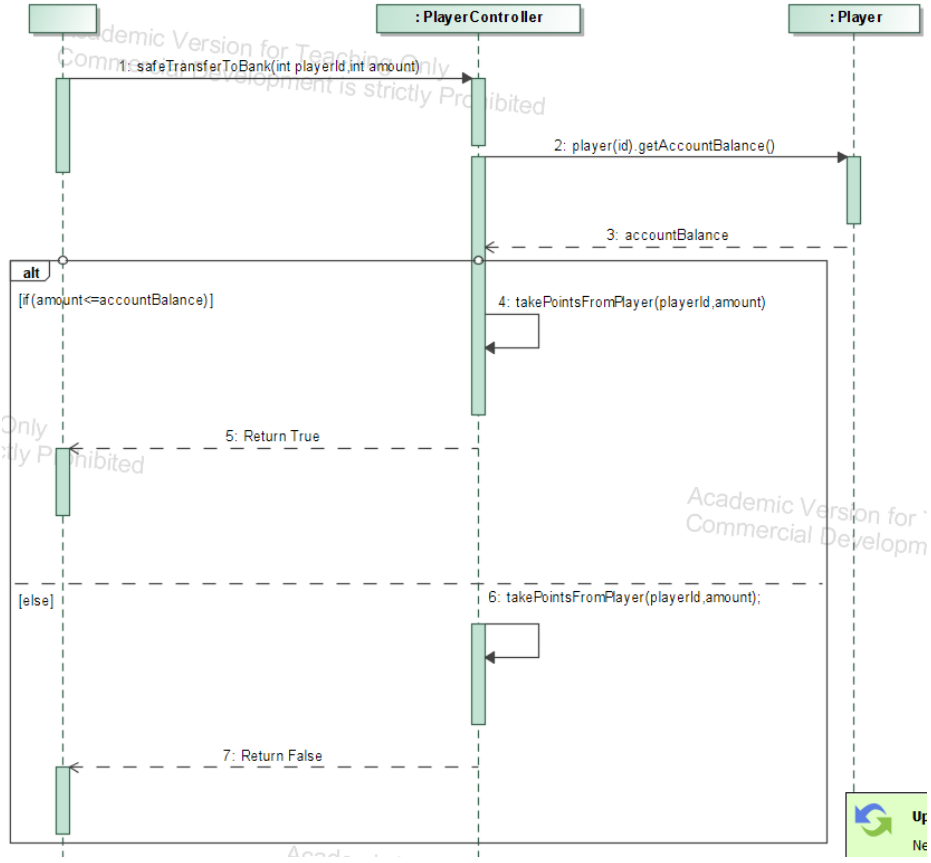
Med udgangspunkt i vores domænemodel og vores viden om matador, samt erfaring fra tidligere projekter, har opstillet et design-klassediagram, som beskriver de klasser vi skal have i vores system. Klassediagrammet ses i figur 10. Det ses, at det er temmelig stort og kan virke uoverskueligt ved første øjekast. Vi har imidlertid benyttet os af flere GRASP-mønstre, der gør at der er en vis struktur i vores klasser. Et meget gennemgående mønster vi har brugt, er controller-mønstret, der helt generelt har gjort os i stand til at styre adfæren af mange af vores objekter af bestemte klasser. Vi bruger altså forskellige controller-klasser til at styre de forskellige dele af programmet. Den mest overordnede controller-klasse er SystemController, der egentlig er det eneste bindeled mellem domænelaget og view-laget. Dvs. SystemControlleren er information expert for alt kommunikation mellem de to lag, og det giver vores system en Model-View-Controller -struktur. SystemControlleren har kendskab til en ViewController, som er information expert for view-laget og desuden til GameControlleren som er information expert i model-laget.

Generelt har vi også prøvet at opnå både low coupling og high cohesion. Low coupling betyder, at klasserne ikke har kendskab til flere andre klasser end højest nødvendigt. I klassediagrammet kan den relativt lave kopling ses ved, at der ikke er mange klasser, der har kendskab ret mange klasser. High cohesion betyder at klasserne har et veldefineret ansvarsområde. Der er altså ikke nogen klasser, der har ansvar for noget, som ikke vedkommer den klasse.



Figur 10: Klassediagram

En anden del af vores design-model er vores sekvensdiagram, som kan ses i figur 11. Sekvensdiagrammet vi har valgt at lave, illustrerer flowet i metoden safeTransferToBank() i PlayerControlleren, som overfører penge fra en spiller til "banken". Metoden sørger for, at hvis spilleren går fallit under overførslen, betaler han så meget han kan, men ikke mere. Metoden returnerer "true"hvis spilleren kunne betale, og "false", hvis han gik fallit.



Figur 11: Sekvensdiagram over metoden safeTransferToBank()

6 Test

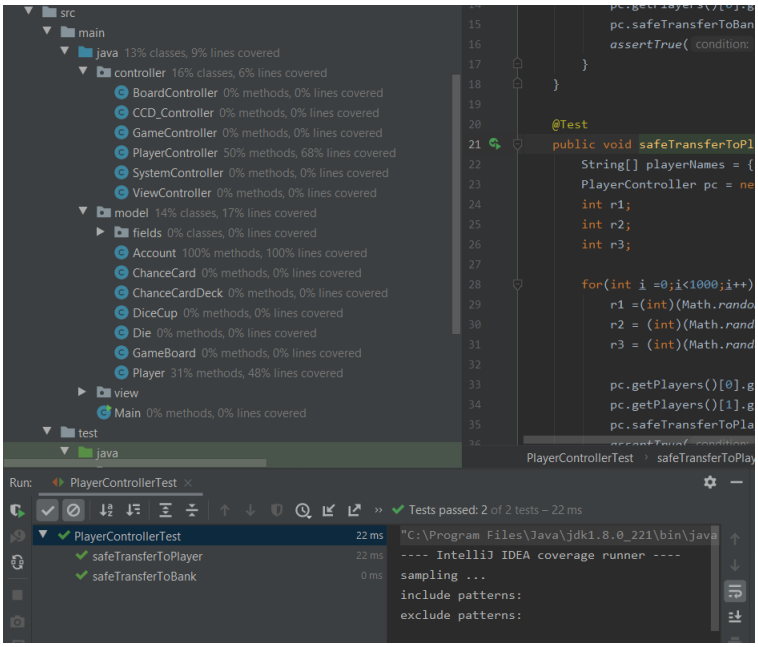
Vi har udført flere forskellige tests for at sikre os at vores system virker på den måde, vi forventer.

6.1 Automatiserede tests

Vi har lavet 2 automatiserede JUnit-tests, som tester 2 centrale metoder i PlayerController-klassen, hhv. safeTransferToBank() og safeTransferToPlayer. Disse er opsat på en måde, så de kan køres fra klasen PlayerControllerTest i test-pakken og programmet, og vil udprinte i konsollen, om de er bestået.

Metoden safeTransferToBank() tager 2 int-argumenter. Den ene er et id på den spiller, der skal lave overførslen, og den anden er beløbet, der skal overføres. Metoden skal gerne sikre, at spillerens balancee aldrig bliver negativ, men at spilleren blot betaler sine sidste penge, hvis spilleren går fallit.

Testen af safeTransferToBank() tester, at ligemeget hvilken (ikke-negativ) balance en spiller har, vil balancen stadig være ikke-negativ efter at safeTransferToBank() er blevet kaldt på den spiller med et vilkårligt positivt eller negativt beløb.



Figur 12: Dokumentation af code coverage i vores JUnit test.

Metoden safeTransferToBank() tager 3 int-argumenter. 2 af dem er id på spillerne, overførslen skal ske imellem, og den sidste er beløbet, der skal overføres. Metoden skal gerne sikre, at spillerens balance ikke bliver negativ, men at spilleren blot betaler sine sidste penge, hvis spilleren går fallit.

Testen af safeTransferToPlayer() tester, at ligemeget hvilke (ikke-negative) balancer de to spillere har, vil deres balancer stadig være ikke-negative efter at safeTransferToBank() er blevet kaldt mellem dem med et vilkårligt positivt beløb. Det vil aldrig være relevant at kalde den metode med et negativt beløb, da man i så fald kunne kalde den ”den modsatte vej”altså.

Vi har af flere omgange gennemført disse tests og altid bestået dem, som de er i deres nuværende form. I figur 12 ses et screenshot fra Intellij efter at de to tests er gennemført med code-coverage. På billedet ses code coverage for de to tests, dvs. det vises hvor mange procent af systemets klasser og linjer, der har været aktiveret under eksekveringen i systemets forskellige pakkker. Vi testede overførselsfunktioner i PlayerController-klassen. Derfor giver det god mening, at kun har været i PlayerController, Account og Player at der er blevet eksekveret kode, som det også fremgår af billedet. Ideelt skulle vi gerne have unit-tests til at teste størstedelen af programmet igennem. Kravet til dette projekt var dog blot at lave en enkelt unit-test. Havde der været mere tid, havde vi lavet nogle flere og mere gennemgående test.

6.2 Manuelle tests

Vi har også udarbejdet 3 testcases med tilhørende testrapporter. Disse kan ses i figur 13, 14 og 15. Disse er tests af funktionaliteten af større dele af programmet, end det er muligt at teste med unit-tests. Unittests virker nemlig rigtig dårligt til at teste en brugergrænseflade, da opførslen af en brugergrænseflade er svær at teste automatiseret. Vores testcases indeholder derfor ”instruktioner”som en bruger kan læse, så brugeren ved hvordan testen skal udføres. Brugeren kan så verificere eller falsificeres testen, og de kan reproduceres af forskellige brugere.

6.2.1 Brugertest

For at teste brugervenligheden og funktionaliteten af vores program, har vi også lavet en brugertest. I brugertesten har vi fået en person, som ikke har kendskab til programmet til at forsøge at spille spillet. Brugeren blev bedt om at ”tænke højt”mens hun spillede, og en udvikler (Peter) tog noter

Test case ID	TC1
Beskrivelse	Test navneinputtet i spillets opsætning
Preconditions	Spillet er startet og har budt velkommen. Ingen spillere har indtastet deres navne.
Postconditions	Systemet kender spillernes navne og den rækkefølge, deres ture skal afvikles.
Procedure	1. Vælg et antal spillere. 2. Indtast det samme antal spillernavne. 3. Vælg en spiller til at være den yngste. 4. Vælg en rækkefølge, disse spillere sidder i omkring det bord, de spiller på. 5. Når spillet starter, tjek at turene afvikles i korrekt rækkefølge: Først den yngste spiller. Dernæst spilleren til venstre for den yngste og dernæst spilleren til venstre for denne osv. 6. Hvis de afvikles korrekt, er testen bestået, eller er testen ikke bestået.
Forventet resultat	Turene blev afviklet korrekt.
Egentligt resultat	Turene blev afviklet korrekt.
Status	Er bestået
Testet af	Sulaiman
Dato	26.11.2019
Test enviroment	IntelliJ IDEA 2019.2.4 (Ultimate Edition) Build #UI-192.7142.36 built on 29 October 2019 On macOS Catalina Version 10.15.1.

Figur 13: Test case 1: Test navneinputtet i spillets opsætning

Test case ID	TC2
Beskrivelse	Test at ejeren af en ejendom vises på brættet.
Preconditions	Spillet er initialiseret med 2-4 spillere.
Postconditions	Spillet er i gang.
Procedure	1. Start spillet op og angiv spillernes navne. 2. Tryk på spillets ejendomme, dvs. de farvede felter og læs at de alle "ingen ejer" har. 3. Når nogen har købt en ejendom, tryk igen på feltet på brættet. 4. Tjek at den korrekte spiller er angivet som ejer.
Forventet resultat	Feltets ejer er nu opdateret med det rigtige navn.
Egentligt resultat	Feltets ejer er nu opdateret med det rigtige navn.
Status	Er bestået
Testet af	Sulaiman
Dato	26.11.2019
Test enviroment	IntelliJ IDEA 2019.2.4 (Ultimate Edition) Build #UI-192.7142.36 built on 29 October 2019 On macOS Catalina Version 10.15.1.

Figur 14: Test case 2: Test at ejeren af en ejendom vises på brættet

Test case ID	TC3
Beskrivelse	Test at spillerne rykker det korrekte antal felter ved terningeslag.
Preconditions	Spillet er initialiseret med 2-4 spillere. Ingen spillere har slået med terningen endnu.
Postconditions	Første spiller har slået med terningen og er rykket.
Procedure	1. Tryk OK for at slå med terningen 2. Tæl antallet af felter den første spiller har rykket. 3. Tjek antallet af øjne på terningen svarer til antallet af felter der er rykket.
Forventet resultat	Det antal øjne, terningen viser er det samme som det antal felter den første spiller har rykket
Egentligt resultat	Det antal øjne, terningen viser er det samme som det antal felter den første spiller har rykket
Status	Er bestået
Testet af	Sulaiman
Dato	26.11.2019
Test enviroment	IntelliJ IDEA 2019.2.4 (Ultimate Edition) Build #UI-192.7142.36 built on 29 October 2019 On macOS Catalina Version 10.15.1.

Figur 15: Test case 3: Test at spillerne rykker det korrekte antal felter ved terningeslag

til brugerens bemærkninger. Brugeren spillede på egen hånd, og fik ingen hjælp eller introduktion til programmet udover at hun på forhånd havde læst reglerne til Juniormatador.
Testen blev foretaget den 28/11/2019 med den på tidspunktet nyeste version af spillet i master-branchen. Dette er den samme version, som den færdige version.

6.2.2 Opsummering af brugerens bemærkninger

Efter testen nedskrev udvikleren følgende opsummering af noterne om brugerens oplevelse med spillet.
Brugeren lykkedes overordnet med at spille spillet. Hun valgte at spille med to "spillere"og færdiggjorde spillet på omtrent 5 minutter. Hun var indforstået med spillets overordnede formål og regler, og forstod hvad der skete, og hvilken spiller der vandt. Der var imidlertid nogle ting ift. brugervenlighed, som hun syntes kunne have været bedre.

1) Ved første tur der blev spillet, var det forvirrende, at systemet ikke fortalte brugeren, at terningen blev rullet. Brugeren opdagede først terningerne på brættet i anden tur.
2) Brugeren fandt ikke ud af, at man kan se ejeren af et felt ved at trykke på feltet. Hun var derfor frustreret over, ikke at vide hvem der ejede de forskellige felter, når hun måtte vælge hvilket felt, en spiller ville rykke til.
3) OK-knappen rykkede sig ofte. Brugeren fandt ikke ud af, at man kunne bruge enter-knappen til at trykke OK.
4) Chancekortene blev stadig vist midt på brættet efter de var blevet brugt, hvis ikke man førte musen hen over. Det kunne have været forvirrende, mente brugeren, hvis man ikke kunne huske hvem der havde trukket et chancekort.

6.2.3 Mulige løsninger

Overordnet må vi sige, at det var nogle rigtig nyttige kommentarer, vi har fået ud af brugertesten. Vi har i denne omgang ikke fået tid til at foretage nogen ændringer på baggrund af testen i denne

version af programmet, men skulle man lave en ny version, ville det være oplagt at reagere på dem. Ift. pointe 1) kunne man ved de første 2-3 ture få systemet til at fortælle brugeren, at terningerne kastes, og hvad der blev rullet.
Ift. pointe 2) og 3) kunne vi have skrevet en slags introduktion eller manual til hvordan spillet benyttes, så brugeren vidste hvilke funktionaliteter, hun havde til rådighed. Ift. pointe 4) kunne man have vist beskeden "Prøv Lykken"på chancekort-feltet i turene efter at et chancekort er blevet trukket og på den måde skjule chancekortet efter det er blevet brugt. Det ville give brugeren illusionen af at lægge kortet "nederst i bunken"igen.

7 Projektplanlægning

Efter at vi havde sat os ind i hvordan Juniormatador spilles, gik vi gang med at lave vores klassediagram og arbejdede med at finde en god struktur i vores system. Vi fik arbejdet os frem til en god skabelon til et klassediagram, hvor der var lav kobling og høj samhørighed mellem klasserne. Dette har vi brugt som generel pejling i løbet af hele projektet for, hvordan vi gerne ville strukturere systemet. Vi begyndte derefter at skrive koden ud fra "bottom up-metoden hvor vi lavede de mest uafhængige klasser først, fordi vi derfor havde større frihed til at ændre i vores struktur hvis vi skulle blive klogere undervejs. Med uafhængige klasser mener vi f.eks. Die, Field, GameBoard osv., altså klasser, der ikke styrer ret mange andre klasser. Efter det begyndte vi langsomt at skrive funktionaliteten i vores controller-klasser. Vi fokuserede på at implementere en funktion af gangen, fx opsætningen af spillebrættet, og samtidig undervejs verificerede og testede at det implementerede virkede som vi ønskede.

7.1 Konfigurationsstyring

Til konfigurationsstyring af vores projekt har vi benyttet os af Maven sammen med IntelliJ for at sikre en god og ensartet konfiguration på tværs af alle udviklere. Ved hjælp af Maven har vi importeret JUnit version 4.12 samt Matador GUI version 3.1.6. De to biblioteker bliver automatisk importeret i IntelliJ og dermed har vi sikret os at alle udviklere benytter sig af samme version af de eksterne biblioteker. Med Maven har vi samtidig også mulighed for nemt at opdaterer hvilken version vi benytter os af, skulle dette blive nødvendigt.

7.2 Importering af Git-Reppository i IntelliJ

I IntelliJ kan man importere projeket direkte fra GitHub på følgende måde: File → New → Project from Version Control → Git
Derefter indsætter man følgende link og klikker på "Clone":

https://github.com/CKyed/11_del3.git
Efter at IntelliJ har hentet projektet fra GitHub skal man klikke på "Auto import"i den boksen der popper op nede i højre hjørne. Derefter vil Maven automatisk importere de forskellige biblioteker.

8 Konklusion

Vores projekt lever så vidt vi ved op stille kravlisten, som vi selv definerede i starten af projektet. Vi har lavet flere forskellige tests, som alle viser positive resultater. Hvis der havde været mere tid til rådighed, havde det helt sikkert været relevant at lave nogle flere og mere uddybende tests til simpelthen at tjekke mere af programmet. Desuden har vi udeladt at implementere en enkelt funktionalitet fra det officielle Juniormatador på grund af tidsmangel.

9 Bilag

9.1 Ordbog

Følgende er en oversigt over de vigtigste begreber der bruges i projektet.

private Hvis en metode eller et felt er private, kan det kun tilgås fra den klasse, det er defineret i.

public Hvis en metode eller et felt er public, kan det tilgås fra alle steder.

get-metode En public metode der returnerer værdien af en klasses felt, som ellers var private.

set-metode En public metode der ændrer værdien af en klasses felt, som ellers var private.

GIT Versioneringsprogrammet, der bruges til dette projekt.

GitHub Online service hvor GIT-projekter kan tilgås.

Nedarving En klasse kan være nedarvet fra en anden klasse. Det medfører at subklassen "arver"superklassens metoder. Eksempelvis nedarver alle vores felt-klasser fra den abstrakte felt-klasse.

Abstrakt En abstrakt klasse er en klasse, der ikke kan oprettes objekter af. Dens formål er derfor egentlig at man skal kunne nedarve andre klasser fra den. Eksempelvis er der i vores spil på intet tidspunkt instantieret et objekt af Felt-klassen, mens der er instantieret mange Property-objekter. Property nedarver, som beskrevet i klassediagrammet, nemlig fra Field.

Polymorfi Polymorfi betyder at flere subklasser, der nedarver fra den samme super-klasse kan have en metode, der hedder det samme, men gør noget forskelligt i hver subklasse. Vi har i vores projekt valgt at bruge polymorfi et enkelt sted, nemlig til getPrice()-metoden. Den returnerer for et property-felt ejendommens pris, mens den for andre typer felter returnerer et negativt tal, som indikerer at feltet ikke kan købes.

9.2 Timeregnskab

	Christian	Peter	Derar	Bashar	Mathias	Sulaiman
08-11	2	2	2	2	2	2
11-11	4	4	4	4	4	4
12-11	3	3	2	2	2	2
13-11	3	3	3	3		3
18-11	5	5	5	5	5	5
19-11	2	2	1	1	1	2
20-11	2,5	2,5				
21-11	2	2				
22-11	5	5		3	2	3
25-11	5	5	5	5	5	5
26-11	5	5	5		5	5
27-11						
28-11	1	2				
29-11						

Figur 16: Timeregnskab