DANMARKS TEKNISKE UNIVERSITET

02312 Indledende programmering
02313 Udviklingsmetoder til IT-systemer
02315 Versionsstyring og testmetoder

CDIO 2

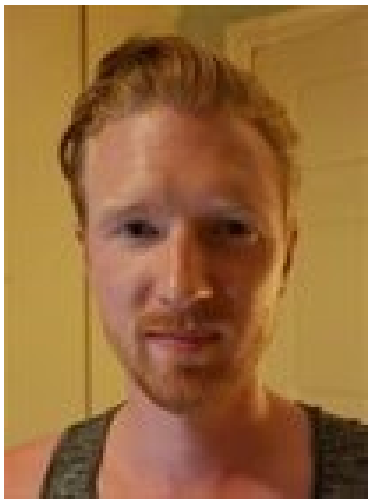
Gruppe 21 består af:
s175565 Nicolai Wolter Hjort Nisbeth
s185020 Nicolai Jeppe Larsen
s185121 Theodor Peter Guttesen
s185024 Marcus Frank Winther-Hinge
s185014 Harald Bruun Brandborg

Undervisere:
Stig Høgh
Ian Bridgewood
Sune Thomas Bernth Nielsen

9. november 2018

Harald B - s185014

Marcus W - s185024



Nicolai L - s185020

Nicolai H - s175565

Theodor - s185121

Indhold

1 Indledning

1.1 Opgavebeskrivelse

Vi har som fiktivt firma, IOOuterActive, fået til opgave af vores kunde, at udvikle et spil, herefter betegnet "Spillet", der skal kunne køres på computerne i DTU's databar. Kundens vision med Spillet er, at to spillere skiftevis kaster med to terninger for derefter at rykke rundt på en spilleplade bestående af 12 felter. Hvert felt på spillepladen har en effekt på spillernes pengebeholdning, som ved spillets begyndelse er 1000. Den spiller der først opnår en pengebeholdning på 3000, vinder spillet.
Sideløbende med udviklingen af Spillet, udarbejdes denne tekniske rapport, som har til formål at beskrive vores analyse, design, implementationen og test af implementationens metoder.

1.2 Abstract

We as a fictitious company named IOOuterActive have been tasked by one of our customers to develop a game, referred as "The Game", which must be playable on the DTU's database. The customer's vision with the game is that two players alternately throws with two dices and then move around on a 12-field playing board. Each field on the board has an effect on the players' bankroll, which at the start of the game is set to 1000. The player who earns 3000 to their bankroll first, wins the game.
Longside the development of the game, a technical report is prepared, which aims to describe our analysis, design, implementation and testing of the implementation methods.

2 Analyse

I følgende analyse tages der udgangspunkt i kundens vision og krav, som i enighed med udviklerne bliver - om nødvendigt - yderligere specificeret og gjort målbare i en kravliste. Som følge heraf vil kravlisten agere som et fælles referencepunkt(baseline) for alle involverede i udviklingen af Spillet og dokumentationen af processen.

2.1 Kravspecifikation

Kravene opdeles her i funktionelle- og ikke-funktionelle krav. Dette gøres for at sikre kvaliteten af programmet, samt at sikre at alle krav bliver dækket. Kigger man på de funktionelle krav, dannes der et overblik, over hvilke krav der stilles til selve koden og programmet som helhed. Man kan sige, at disse krav er programmets funktioner, altså hvad programmet skal kunne gøre. Et eksempel på et funktionelt krav er; “Spillerne starter med en pengebeholdning på 1000”, altså et krav til programmets funktioner.

De ikke-funktionelle krav derimod er indirekte krav, der ikke umiddelbart har noget at gøre med Spillets funktioner. Dette er f.eks; “Alt kode er skrevet på engelsk”

Fordelen ved at inddele kravene i disse to grupper er, at man ved opdelingen opsætter kravene under forskellige parametre, og at man ved at vide om det er tale om funktionelle- eller ikke-funktionelle krav på forhånd ved, at der er forskel på dokumenteringen af løsningen eller dækningen af pågældende krav. Et eksempel på dette er at de funktionelle krav bl.a. kan verificeres og afkrydses ved at teste metoder. De dertilhørende ikke-funktionelle er derimod “blødere” at verificere, f.eks. "Alt kode er skrevet på engelsk."

Funktionelle krav

1. FK01 Spillet skal spilles mellem 2 personer.
2. FK02 Spillet skal spilles ved at spillerne skiftes til at slå med de to terninger og lander på et felt med numrene 2 til 12.
3. Fk03 Spillet skal have felter der påvirker spillernes pengebeholdning på en positiv eller negativ måde.
4. FK04 Spillet skal udskrive en tekst med information om det aktuelle felt.
5. FK05 Spillet skal starte med at spillerne har en pengebeholdning på 1000.
6. FK06 Spillet skal afsluttes når en af spillernes pengebeholding når 3000.
7. FK07 Spillet skal indeholde felter der er beskrevet i feltlisten. Se bilag A.
8. FK08 Spillet skal have en metode til at skifte terningerne.

Ikke-Funktionelle krav

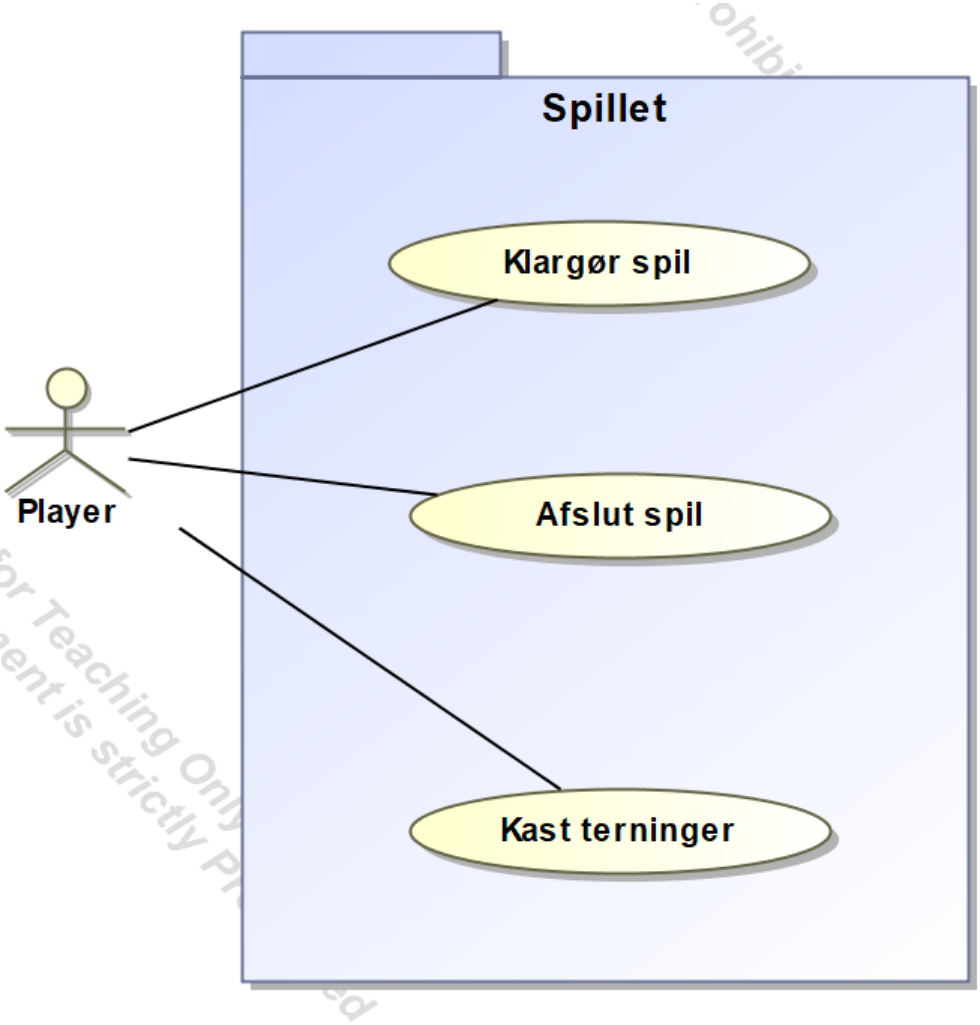
1. IfK01 Spillet skal kunne spilles på maskinerne i DTU’s databarer, uden bemærkelsesværdige forsinkelser.
2. IfK02 Spillet skal være konsekvent ift. om udskrifterne er på dansk eller engelsk.
3. IfK04 Spillet skal overholde GRASP principperne.
4. IfK05 Spillet skal have de centrale metoder testet med implementering JUnit framework.
5. IfK06 Spillet skal designes så spillernes pengebeholdning aldrig kan blive negativ.
6. IfK07 Spillet skal have en brugervejledning ift. krav til styresystem og installerede programmer det kræves for at kunne køre spillet.
7. IfK07 Spillet skal let kunne oversættes til andre sprog.

2.2 Use case diagram

Use case diagrammer består af aktører og deres relation til systemets use cases, i forsøg på at danne et klart overblik over hvad spillet skal kunne.

Ud fra kundens vision og den udarbejdede kravspecifikation, bliver det tydeligt at Spillets omdrejningspunkt er at spillerne på skift kaster med terningerne, og lander på et felt som har konsekvens for deres pengebeholdning. Derfor har vi oprettet en use case "Kast terninger". Dog - inden spillet påbegyndes - skal spilleren først tilmelde sig via navn og blive tildelt en pengebeholdning på 1000, dertil hører use casen "Klargør spil". Afslutningsvis skal vinderen af spillet findes og annonceres, som giver use case navnet "Afslut spil".

Alle use casene tager udgangspunkt i spilleren, derfor er spilleren Spillets primære - og eneste aktør og har relation til samtlige use cases i Spillet, se figur 2.



Figur 2: Use case diagram for brætspillet

2.3 Use case beskrivlser

Alle use cases beskrives på en af følgende måder, “brief”, “casual” eller “fully dressed”. Forskellen på de tre måder ligger i hvor omfattende beskrivelsen er. Vi har valgt at beskrive use casen “Kast terninger” fully dressed, da denne use case er ansvarlig for størstedelen af spillets flow, hvorimod “Klargør spil” og “Afslut spil” begge har mindre ansvar. De tre use case beskrivelser fremgår i samme rækkefølge som i use case diagrammet.

Klargør spil

To personer starter et spil mod hinanden. En person indtaster et navn for spiller 1 hvorefter der oprettes 1000 point til denne spillers pengebeholdning. Samme procedure følger for spiller 2. Personerne vælger herefter hvor mange sidder de ønsker de 2 terninger, der bruges til spillet, skal have og dette oprettes i overensstemmelse med deres ønske. Spillet er nu klargjort.

Afslut spil

På et tidspunkt under spillet vil en af de 2 personer opnå en pengebeholdning på 3000 point, hvilket er målet for at vinde. Spillet giver besked om at en vinder er fundet, og oplyser navnet på Spilleren som har vundet. Herefter afsluttes spillet.

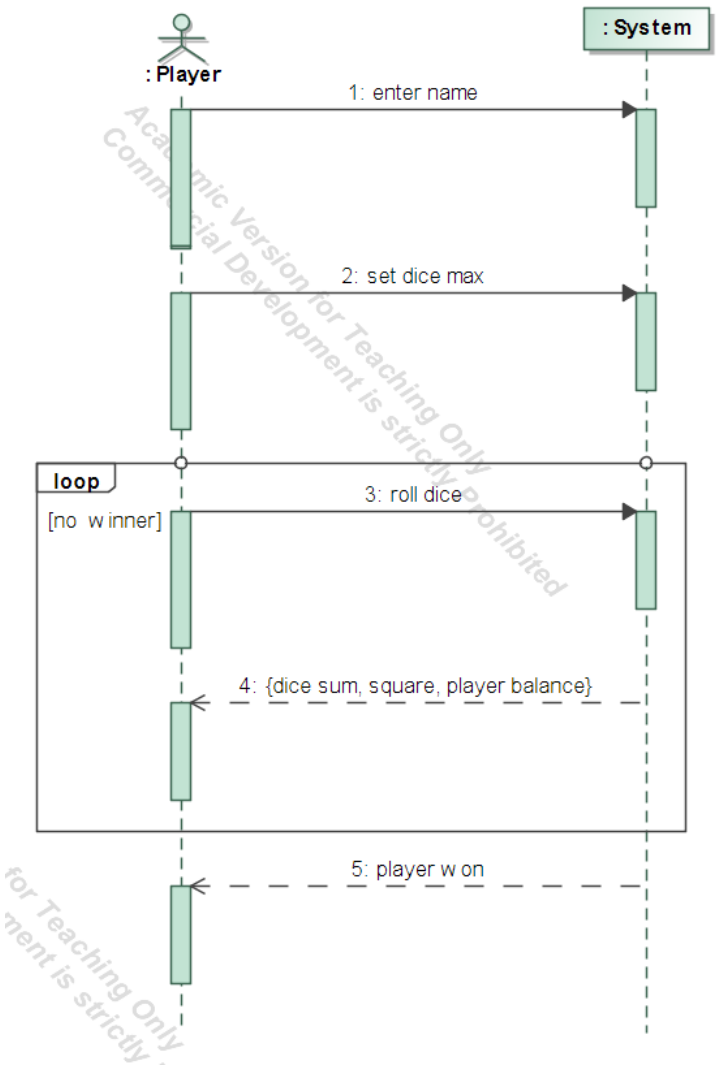
Kast terninger

Use case: Kast terninger
ID: Nr.1
Kort beskrivelse: Spiller 1 og spiller 2 kaster skiftevis med spillets terninger, deres brikker rykkes til et nyt felt og deres pengebeholdning opdateres efter hver kast i overensstemmelse med det felt de har landet på.
Primære aktører: Spillerne
Sekundære aktører: -
Preconditions: To spillere har indtastet navn og angivet terninge type.
Main flow: 1. Spiller 1 kaster med terningerne. 2. Spillet henter terningernes øjne og rykker spiller 1 til det passende felt. 3. Spillet udregner spiller 1’s pengebeholdning og sender det som besked. 4. Spiller 2 kaster med terningerne. 5. Spillet henter terningernes øjne og rykker spiller 2 til det passende felt. 6. Spillet udregner spiller 2’s pengebeholdning og sender det som besked. Spillet gentager trin 1 - 6.
Post Conditionals: Spillet afsluttes når en spillers pengebeholdning når 3000.
Alternative flows: *a. En af spillerne afslutter spillet før, der er fundet en vinder. 3.1 Spiller 1 får en ekstra tur, hvis han lander på felt 10. 6.1 Spiller 2 får en ekstra tur, hvis han lander på felt 10.

Tabel 1: Use case Kast terninger "fully dressed"

2.4 Systemsekvens diagram

Systemsekvensdiagrammet tager udgangspunkt i use case beskrivelserne og visualiserer spillernes interaktioner med Spillet. Begyndelsesvis skal spillerne indtaste deres navne og tildeles en pengebeholdning på 1000 inden spillet påbegyndes. Herefter kaster spillerne skiftevis med terningerne, rykker til det tilsvarende felt og som konsekvens deraf justeres spillernes pengebeholdning. Denne rækkefølge af handlinger gentages indtil en af spillerne har vundet, se figur 3.



Figur 3: Systemsekvensdiagram for brætspillets central operationer

2.5 Domænemodel

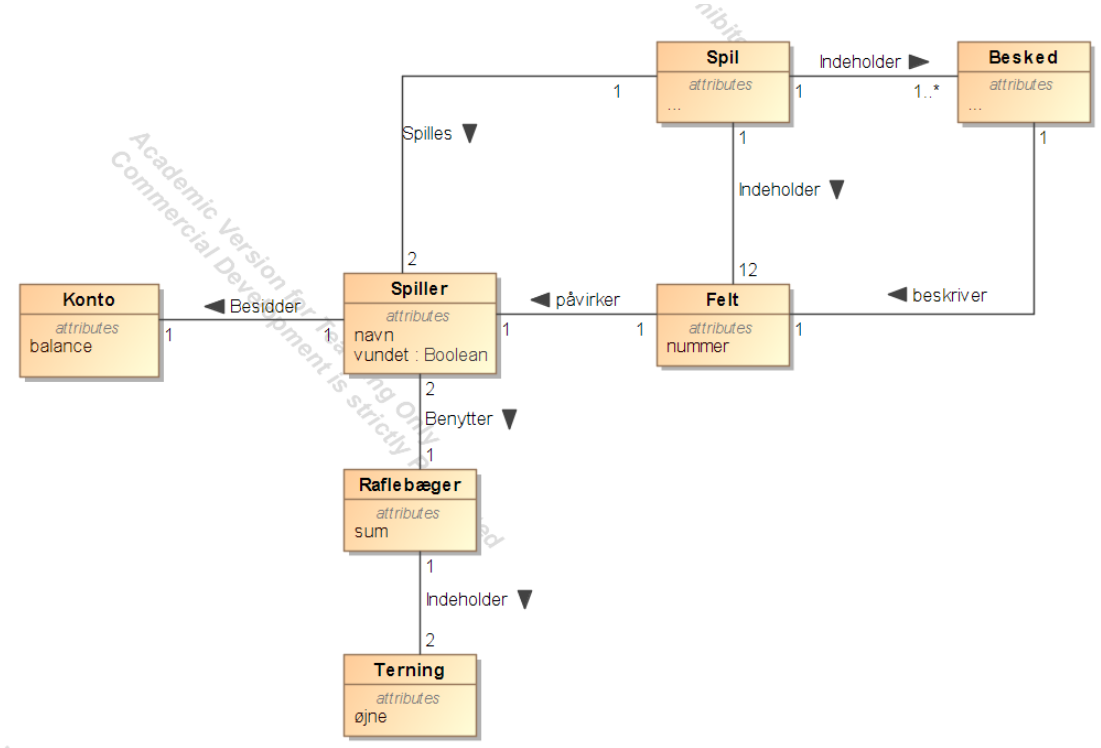
Domænemodellen tager - ligesom systemsekvensdiagrammet - udgangspunkt i use case beskrivelserne. Dog anvendes disse beskrivelser ikke til at finde interaktionen mellem spillerne og Spillet, men derimod i identificering af koncepter/objekter som tilsammen udgør spillets arkitektur. Domænemodellen har til opgave at give et tydeligt overblik over koncepternes indbyrdes relation og forholdene imellem dem. Til identificering af Spillets koncepter har vi derfor udført en navneordsanalyse af use case beskrivelserne, se tabel 2.

Navneord	Klassenavn kandidater	Attribut kandidater
Spiller	x	
Felt	x	
Terning	x	
Øjne		x
Navn		x
Besked	x	
Pengebeholdning	x	
Spil	x	
Vinder		x
Tur		x

Tabel 2: Navnordsanalyse af use cases

På baggrund af navneordsanalysen bliver det klart at Spillet består af spillere, felter, terninger, pengebeholdninger og beskeder til spillerne.

I domænemodellen kan der f.eks. tages udgangspunkt i Spil, et Spil spilles af to Spillere, som hver besidder én Konto og skiftevis benytter et Raflebæger, som indeholder to Terninger. Et spil indeholder også 12 Felter, som hver påvirker én spiller. Derudover styres Spillets kommunikation til spillerne af beskeder, så et Spil indeholder én eller flere beskeder, som hver giver forklaringer om felterne, start- og slutbeskeder.



Figur 4: Domænemodel for spillet

3 Design

I analysen blev samtlige af Spillets, funktionelle og ikke funktionelle krav defineret/opstillet i kravspecifikationen. Alle funktionelle krav blev afdækket i tre use cases og sat i relation til Spillets aktører i use case diagrammet. Use casene blev beskrevet og ud fra dem blev de mest centrale operationer mellem spillerne og Spillet visualiseret i systemsekvensdiagrammet og Spillets koncepter identificeret og opstillet i domænemodellen.

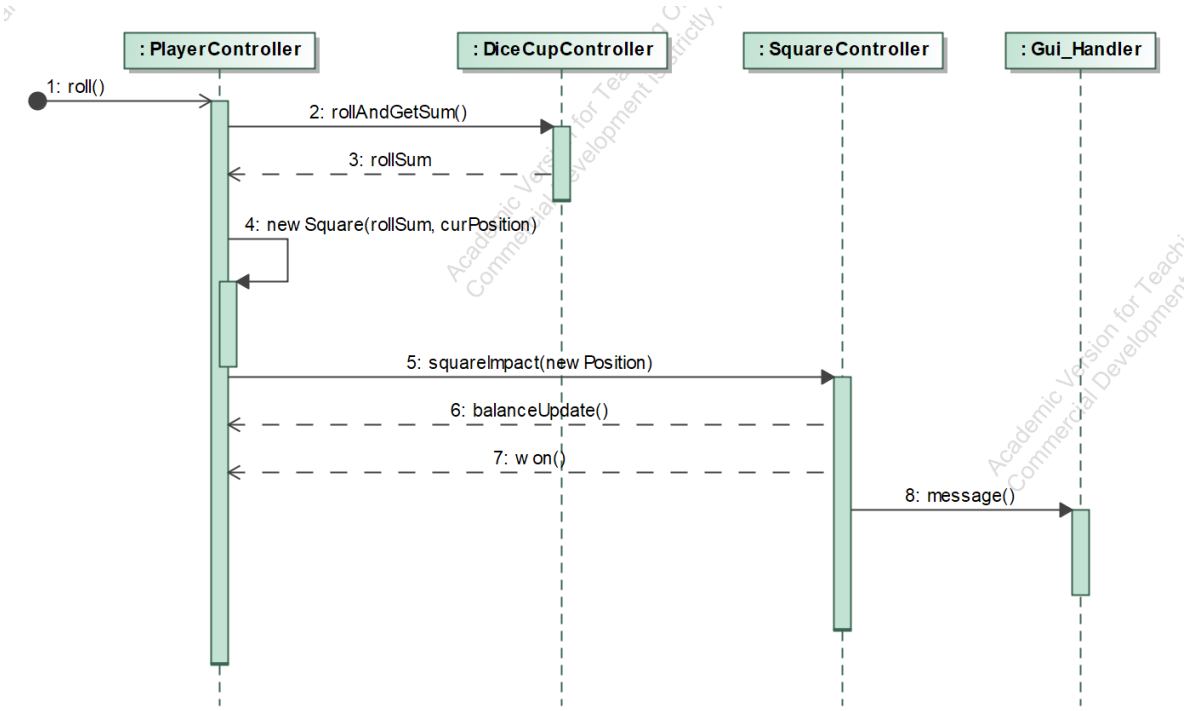
I udarbejdelsen af designet vil systemsekvensdiagrammet og domænemodellen fungere som inspiration i navngivning- og definering af software klasser for at nedsætte det repræsentative kløft - "the representational gap".

3.1 Sekvens diagram

Sekvensdiagrammet belyser den mest centrale metode i Spillet, metoden roll() under klassen PlayerController. Metoden er central for Spillet, fordi den både skal stå for at kaste med terninger, ændre spillerens position på spillepladen og opdatere spillerens pengebeholdning i overensstemmelse med det felt han lander på. Da det er et objekt af DiceCupController der kalder metoden til at kaste med sine egne terninger, er det det første der sker i roll().

Terningekast metoden returnerer summen af terningernes øjne, som sammen med spillerens nuværende position bruges som argument til newSqare(), der udregner spillerens nye position.

Derefter bruges spillerens nye position som argument i metoden squareImpact(), der sender en besked til spilleren om det felt han har landet på, opdaterer spillerens pengebeholdning og vurderer om han har vundet. Til sidst bliver der sendt en besked til spilleren, som oplyser om ændringen i hans pengebeholdning.



Figur 5: Sekvens diagram af Spillet

3.2 Designklasse diagram

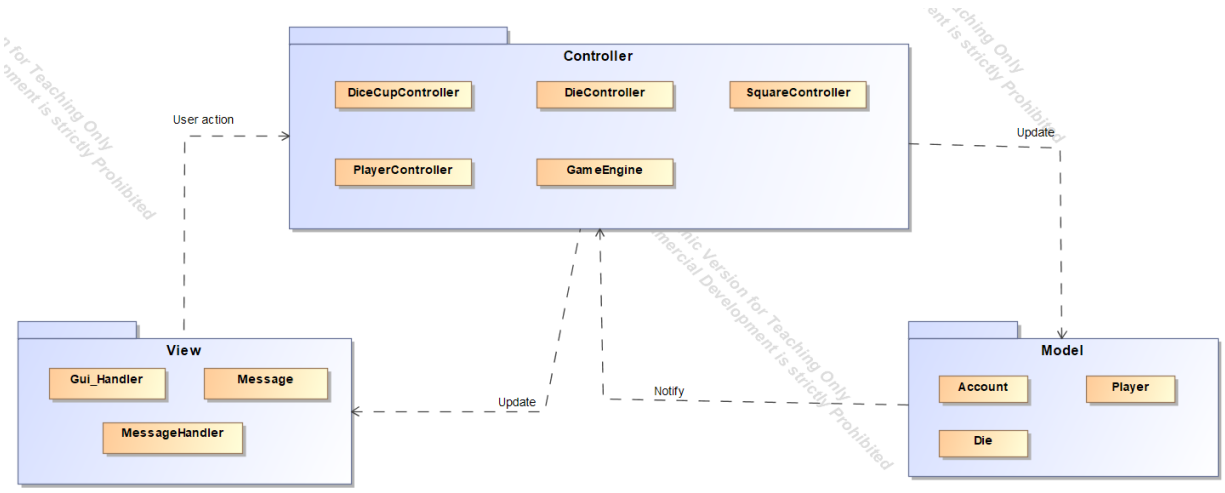
Design klassediagrammet er udgangspunktet for software koden. Det er blevet inspireret af domænemodellen og er blevet udviklet parallelt med udviklingen af Spillet. Diagrammet er en statisk visualisering af Spillets klasser med klassernes attributter og metoder. Diagrammet viser relationerne klasserne imellem. De enkelte klasser bliver beskrevet dybere under afsnittet om implementering.

3.3 GRASP-mønstre

For at beskrive arkitekturen mere overordnet og anvende principperne i GRASP, har vi delt Spillets klasser ind i tre pakker; Model, View og Controller. Dette har gjort det lettere at adskille ansvarsområder og opnå lavere kobling og afhængighed på tværs af klasserne. F.eks bliver model-klasserne skabt af deres respektive controller-klasser.

Derudover har hver klasse et veldefineret ansvarsområde, hvilket medfører en højere grad af samhørighed, cohesion. De forskellige klassers metoder påvirker én bestemt model-klasses tilstand og ikke flere forskellige. F.eks. ligger metoden newSquare() under PlayerController og ikke under SquareController, da metoden ændrer på en spillers data, se figur 6. Vi har givet de forskellige klasser og deres metoder navne, som let kan forstås uden behov for mange kommentarer og har derfor opnået.

Klassen GameEngine er en klassisk Creator, den skaber flere af de controller-klasser spillet bruger. Det samme er DiceController, DiceCupController og PlayerController. Også Model-klassen Player skaber et objekt. Dette af klassen Account. GameEngine er også Information Expert, da den indeholder information om flere klasser.



Figur 6: Model View Controller for brætspillet

4 Implementering

4.1 Overvejelser ift. implementeringen af kode

Vi har haft følgende overvejelser ift. implementeringen af koden og er kommet frem til følgende:

Skal udfaldet af spillerens kast være nummeret på det felt han rykker til?
Et stort spørgsmål angående implementering af koden var, hvordan det skulle håndteres at spilleren efter at have kastet terningerne bliver rykket til det rigtige felt. Dette valgte vi at håndtere ved at bruge modulus til at udregne det nye felt. Se mere under afsnittet PlayerController om metoden roll.

Skal det være muligt for spillerne - inden spillets begyndelse - at vælge terningernes type?
Kravet om at spillerne skal kunne vælge en anden type terningen end den traditionelle 6-sidet, FK08, har medført at vi måtte begrænse valget til at terningerne skal have minimum en side og maksimalt seks, da GUI'en ikke er sat op til at kunne animere terninger på mere end seks sider.

4.2 Oversigt over klasser og metoder

De forskellige klasser beskrevet i vores design, har alle forskellige ansvarsområder med dertilhørende funktioner. Disse bliver herunder uddybet.

Main
Klassen Main ligger ikke i nogen pakke. Den indeholder kun en metode, main-metoden, som starter programmet. I main-metoden, kaldes metoden start() fra klassen GameEngine. Således startes spillet.

Controller
Klasserne i pakken Controller indeholder al Spillets logik. De forskellige controller-klasser er ansvarlige for at instantiere, de model-klasser, hvis data de har ansvar for at bruge og påvirke.
GameEngine er er ansvarlig for at intentiere klassserne PlayerController, DiceCupController, SquareController og GuiHandler. Den instantierer ingen model-klasser, og dens ansvar består i at køre spillet. Når metoden start() bliver sat igang, starter to metoder; setupGame() efterfulgt af playGame().
Den første metode starter med at køre metoden setGameUpGui(), der skaber spillepladen, spillere, og giver dem hver deres spillebrik under klassen GuiHandler. Så får spillerne instruktioner om spillet, og bliver bedt om at indtaste deres navne, og hvilke slags terninger de ønsker at bruge. Til dette bruges to metoder fra klassen GuiHandler, der håndterer alle GUI'ens metoder. DiceCupController bliver instantieret i overensstemmelse med det input spillerne giver den.
Derefter kører den næste metode playGame(). Metoden indeholder et do-while loop, der styrer spillets gang. Først er det spiller1's tur, så spiller2. Loopet kører indtil en af spillerne vinder spillet, og giver så til sidst en besked om, hvem der har vundet spillet.

PlayerController instantierer et objekt af model-klassen Player. Hvert objekt af klassen har ansvaret for at styre objektet af Player's data. For at kaste med terninger og spille spillet bruges metoden roll().
Metoden kalder metoden rollAndGetSum() fra DiceCupController, som kaster med begge terninger og returnerer deres samlede værdi. Værdien bruges, sammen med metoden getPosition(), der returnerer spillererens nuværende position på spillepladen, som argument i metoden newSquare().
Denne metode har som eneste ansvar at udregne spillerens næste position på spillepladen. Da der kun er 12 felter, og hvis spilleren står på felt nummer 1 og slår 12, ville han lande på felt nummer 13 der ikke findes. Derfor undersøges metoden om spillerens nuværende position summeret med værdien af hans

terningekast er lig 24 eller over 12. Hvis den er lig 24 skal spillerens position sættes til 12. Hvis summen er over 12, som f.eks 13, skal hans position sættes til 13%12, der giver resultatet 1. Og endelig, hvis hans slag og position summeret ikke er over 12, skal hans nuværende position sættes til den udregnede sum, hvilket bliver gjort med et metodekald til setPosition().
Efter spillerens position er udregnet bliver metoden squareImpact() kaldt. Den tager en specifik spiller, en reference til et objekt af GuiHandler og af diceCup som argument. Herefter bliver det via en switch-statement bestemt, hvordan spillerens nye placering på spillepladen skal påvirke hans pengebeholdning, om han skal kaste med terninger igen, og om han har vundet.

SquareController er en controller klasse, som indeholder en metode: squareImpact. Metoden indeholder et switch statement. Den værdi der switches på er newSquare, som er lig spillerens position.
Der 12 cases, som alle sammen indeholder en metode til opdatering(specifik for feltet) af spillerens balance og hvis den opdatering er positiv, er der også en metode der finder ud af om spilleren har vundet, som er en mulighed, fordi spillerens pengebeholdning kan være 3000 eller over.
De indeholder også allesammen en metode der får GUI til at vise den nye position. Og en metode der får GUI til at vise feltets besked. Derudover er der break; for ellers ville de efterfølgende cases blive kørt.

Den case der adskiller sig fra de andre er case 10, hvis man lander på det felt får man et ekstra slag. Derfor er der en metoder som giver spilleren i GUI giver en ekstra tur dvs. besked om at slå igen. Der er også en metode der fjerner bilen i gui fra felt 10 ellers ville der i GUI være flere af den samme bil på pladen. Og til sidst er der en metode der kaster terningerne.

DiceCupController har en constructor med parameterne antal terninger og en reference til et objekt af GuiHandler klassen. Den bruger dette objekt til at få antallet af sider på Spillets terninger, som kommer fra brugeren. Terningerne opbevares i et array i DiceCupController klassen.

Model

De tre klasser i pakken Model er Die, Player og Account. De indeholder de forskellige klassers attributter samt getter og setter metoder. Die holder på data for de terninge objekter, Spillet bruger. Die har en privat int final MAX, som står for antallet af sider der er på terningen.
Player indeholder spillerens navn, position på spillepladen, samt en boolean der angiver, om spilleren har vundet eller ej. Når et objekt af Player bliver instantieret af PlayerController skabes et objekt af klassen Account.
Hver spiller har deres egen account, der skal justeres for positive og negative ændringer, som opstår, når spilleren lander på de forskellige felter på spillepladen. Spillerens Account objekt har en attribut der hedder balance. Den må få en værdi under 0. Dette kontrolleres af metoden updateBalance(), der bliver brugt hver gang der attributten bliver ændret. Hvis ændringen medfører at balance går i minus, sættes den til 0.

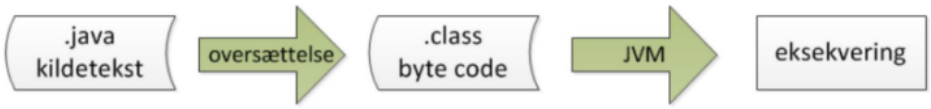
View

Pakken View indeholder al kommunikation mellem spilleren og Spillet. Spillerens input bliver håndteret ved hjælp af en GUI, og beskeder til spillerne fra Spillet er beskrevet i Message, MessageHandler og GuiHandler
Klassen GuiHandler er implementeret til at håndtere alle anvendelser af GUI'ens metoder. Konstruktøren i GuiHandler instantierer et array med 12 referencer til Spillets 12 felter, som Spillets spilleplade indeholder. Felterne får i metoden fieldsAttributes() tildelt den titel og beskrivelse, der beskriver det respektive felt, se bilag A.

Metoden bliver kaldt som det første i metoden setGameUpGui(). Metoden tager to referencer af objekter af PlayerController som parametre. Referencerne bruges til at instantiere to GUIPlayer objekter, der hver får tildelt en spillebrik i form af et objekt af Car. Disse spillebrikker bliver så sat på spillepladen på felt nummer 1.
For let at kunne flytte den enkelte spillers brik på spillepladen med metoden setPlayerCar(), finderer getGuiPlayer() ud af hvilken guiPlayer, der skal bruges som argument til at sætte spillebrikken. Metoden setDieFaces() returnerer et objekt af klassen DiceCupController med antallet af øjne på terningerne, som input fra spillerne.

4.3 Konfiguration

For at køre Spillet er der nogle minimumskrav, som skal opfyldes. Man skal have intelliJ samt en Java platform, JDK, installeret. Derudover skal alle systemkrav til Java 7 overholdes, se bilag B.
For at hente koden fra et git repository, starter man med at oprette et nyt Maven projekt. Derefter vælges "Checkout from version control" i menuen VCS. Her skal url-adressen til projektets git repository skrives, "https://github.com/Nicolai2100/CDIO2.0.git". Så at trykker man på "clone" og projektet bliver hentet.
For at kunne køre koden skal man højreklikke på pom-filen og trykke "Reimport" under Maven. Så bliver det eksterne GUI-bibliotek downloadet, og man kan køre main-metoden.
Kildekoden køres gennem metoden main() i klassen Main. Derfra køres alle statements i metoden et efter et. Alle Java programmer starter fra denne metode. Programmer i Java skrives i et high-level sprog, som kan læses af mennesker. Da computere kun kan læse code i binær form, skal kildekoden først oversættes af compileren til .class filer som indeholder byte code. Byte code skal derefter køres af Java's virtuelle maskine (JVM), se figur 7.



Figur 7: Kompilering i Java

5 Test

Pålidelig software er gennemtestet software. Derfor har vi som udviklere, parallelt med implementeringen af kildekode, opstillet og udført testcases i takt med at de enkelte dele af programmet blev færdigt (unit tests). På den måde sikrer vi os at fejlfyldt kode ikke bliver tilføjet systemet og forringer kvaliteten.
Samtlige tests er automatiske frem for manuelle og er udarbejdet vha. frameworket JUnit. Det skyldes ikke mindst på grund af brætspillets udformning, da muligheden for at udføre manuelle tests er begrænset. Derudover vil brugen af automatiske tests sikre konsekvente udfald til opstillede scenarier og åbne muligheden for at udfører fx. 10.000 terningekast markant hurtigere end nogen anden manuel test ville kunne.
Nedenfor har vi dokumenteret vores test proces - for de mest centrale metoder - for at sikre at udfaldet kan reproduceres, da kunden skal have muligheden for at gentage afprøvningen. Vores tests tager højde for de tre fejltyper syntaksfejl, run-time fejl og logisk fejl.

5.1 Test cases

PlayerControllerTest
TC01 Metoden roll() er testet ved først at sikre at spillerens startposition er 1 vha. assertEquals. Derefter kaldes metoden roll() og der tjekkes om spillerens nye position er korrekt.

TC02 Metoden newSquare() er testet ved først at bekræfte at spillerens startposition er 1 vha. assertEquals. Derefter kaldes metoden newSquare tre gange med forskellige argumenter i forsøg på at teste alle metodens forgreninger. For hver kald bliver der afslutningsvis undersøgt om spillerens nye position er hvad der forventes vha. assertEquals.
Første kald har argumenterne rollSum = 12 og getSqaure = 12, spillerens nye position i dette tilfælde skal være 12.
Næste kald har argumenterne rollSum = 11 og getSquare = 3, spillerens nye position i dette tilfælde skal være 2.
Sidste kald har argumenterne rollSum = 2 og getSqaure = 3, spillerens nye position is dette tilfælde skal være 5.
Derudover bliver metoden også testet for hvorledes alle nye positioner er mellem 1 og 12 ved brug af en for-løkke som kaster og udregner nye positioner 1000 gange.

TC03 Metoden won() bliver testet ved først at undersøge om spillerens pengebeholdning er 1000 ved spillets begyndelse, anvende metoden won og tjekke om sandhedsværdien er falsk vha. assertFalse. Derefter opdateres spillerens pengebeholdningen til 3000 og metoden won() kaldes igen og der tjekkes påny om spillerens sandhedsværdi er ændret fra falsk til sand vha. assertTrue.

TC04 Metoden updateAccountBalance() testes ved først at bekræfte at spillerens indledende pengebeholdning er 1000. Derefter kaldes updateAccountBalance() med argumentet accountUpdate = 500, som skal opdatere spillerens pengebeholdning til 1500 og tjekkes med assertEquals. Afslutningsvis forsøges at gøre spillerens pengebeholdning negativ ved at give updateAccountBalance() argumentet accountUpdate = -100000, og det aflæses at pengebeholdningen er sat til 0.

SquareControllerTest
TC05 Metoden squareImpact() er testet ved at undersøge hvorvidt spillerens pengebeholdning opdateres korrekt ud fra hans position på et af de tolv felter. Her anvendes en for-løkke som looper spilleren

over samtlige felter og deres virkninger, og tjekker om hans pengebeholdning opdateres korrekt vha. assertEqual.

GameEngineTest
TC6 Metoden setUpGame() bliver testet ved at kalde metoden setUpGame(), hente en statisk variable numOfPlayers af data typen int, som bliver inkrementeret hver gang en spiller bliver instantieret vha. konstruktøren i playerTurnController og tjekke om variablen er lig 2 vha. assertEquals.

DiceCupController
TC7 Metoden setDieFaces() testes for at undersøge om metoden til at sætte terningernes max værdi fungerer, som tiltænkt. Spillerne skal ikke kunne indtaste en værdi der er under 1 og større end 6.

5.2 Accept Test

For at sikre at alle spillets funktionelle krav er implementeret opstilles en traceability matrix (sporbarheds-matrix). Matricen indeholder alle test cases vertikalt og krav horisontalt. Ud fra tabel 3, aflæses det at alle funktionelle krav mindst har bestået en test case og dermed også implementeret i systemet.
Det bemærkes at FK04, som den eneste, ikke er opfyldt af en automatisk test case men derimod en manuel. Det skyldes kravets udformning, da spilleren skal modtager en feltbeskrivelse efter ankomst til feltet.

Test cases	**Funktionelle Krav**							
	FK01	FK02	FK03	FK04	FK05	FK06	FK07	FK08
TC01		x						
TC02							x	
TC03						x		
TC04					x			
TC05			x				x	
TC06	x							
TC07								x

Tabel 3: Traceability matrix af testcases og funktionelle krav.

6 Konklusion

Vi har som spilfirmaet IOOuterActive udviklet et spil, der lever op til de krav, vi fik til opgave. Vi har valgt at inkludere en GUI og fået et færdigt produkt med en grafisk brugergrænseflade, der forestiller et simpelt matador-spil.

I afsnittet herunder vil vi redegøre for og vurdere processen og det udarbejdede produkt.

6.1 Overordnet konklusion

CDIO projekt er gennemarbejdet med alle krav implementeret og testet. Koden er letlæselig og veldokumenteret, hvilket vil gøre en evt. videreudvikling let. Produktet er blevet færdigt til tiden, og derfor formodes det at kunden må være tilfreds med det udviklede produkt.

6.2 Procesorienteret konklusion

Vi har denne gang ikke været lige så struktureret mht. tidsplan og timeregistrering, da vi følte at udbyttet ikke var den tidsmæssige investering værd. Da vi derudover har været ramt af fravær i form af sygdom osv., mener vi alligevel, at vi har fået løst opgaven tilstrækkeligt.I udvikling af projektet har arbejdet ikke været lige så jævnt fordelt ud over gruppens medlemmer som førhen. Dette har ført til at den individuelle investering ift. tid svajer en del mellem gruppens medlemmer, da nogle har båret det tunge læs og lagt flere timer i projektet.

6.3 Videre arbejde

Produktet er opsat med de basale funktioner, der kan danne rammen for udvikling af et spil med yderligere kompleks funktionalitet. Da vi allerede på forhånd har fået at vide at målet er at udvikle et funktionelt matador-spil, har vi her efter afslutningen af CDIO 2 fået en god forståelse for en stor del af de klasser, der også kommer til at være relevante i det endelige matador-spil.

Bilag

A Første bilag

Feltliste

1.	(Man kan ikke slå 1 med to terninger)		
2.	Tower	+250	
3.	Crater	-100	
4.	Palace gates	+100	
5.	Cold Desert	-20	
6.	Walled city	+180	
7.	Monastery	0	
8.	Black cave	-70	
9.	Huts in the mountain	+60	
10.	The Werewall (werewolf-wall)	-80,	men spilleren får en ekstra tur.
11.	The pit	-50	
12.	Goldmine	+650	

B Anden bilag

Java 7 System Requirements

Detailed information on Java 7 system requirements are available at Java 7 Supported System Configurations.

Windows

- Windows 10 (7u85 and above)
- Windows 8.x (Desktop)
- Windows 7 SP1
- Windows Vista SP2
- Windows Server 2008 SP2 and 2008 R2 SP1 (64-bit)
-
- Windows Server 2012 (64-bit) and 2012 R2 (64-bit)
-
- RAM: 128 MB; 64 MB for Windows XP (32-bit)
- Disk space: 124 MB
- Browsere: Internet Explorer 7.0 and above, Firefox 3.6 and above

Note: As of April 8, 2014 Microsoft stopped supporting Windows XP and therefore it is no longer an officially supported platform. Users may still continue to use Java 7 updates on Windows XP at their own risk, but support will only be provided against Microsoft Windows releases Windows Vista or later. See Third Party Vendor-Specific Support Terms on Oracle Software Technical Support Policies (pdf) for details.

Mac OS X

- Intel-based Mac running Mac OS X 10.7.3 (Lion) or later.
- Administrator privileges for installation
- 64-bit browser

A 64-bit browser (Safari, for example) is required to run Oracle Java on Mac.

Linux

- Oracle Linux 5.5+
- Oracle Linux 6.x (32-bit), 6.x (64-bit)[3]
- Oracle Linux 7.x (64-bit)[3] (7u67 and above)
- Red Hat Enterprise Linux 5.5+, 6.x (32-bit), 6.x (64-bit)[3]
- Red Hat Enterprise Linux 7.x (64-bit)[3] (7u67 and above)
- Suse Linux Enterprise Server 10 SP2, 11.x
- Suse Linux Enterprise Server 12.x (7u75 and above)
- Ubuntu Linux 10.04 and above
-
- Browsers: Firefox 3.6 and above