CDIO 1

Anders B. P.,

Magnus R. B.,

Michael J.,

Nicklas C F.,

Philip B. Ø.,

William D.

12. oktober 2018

Indholdsfortegnelse

1 | Indledning

Denne opgave er udarbejdet i samarbejde med kurserne: udviklingsmetoder til IT-systemer(02313), indledende programmering(02312) og versionsstyring og testmetoder(02315).
Vi skal i dette projekt udarbejde et spil til windows for firmaet IOOuterActive, hvor 2 spillere kan slå med 2 terninger. Spillerne får point ud fra summen af øjne de slår, og den første spiller der når 40 point vinder. Der er andre udvidede krav til spillet som vi kommer ind i under vores kravspecifikation.

2 | Formål

Vores formål med opgaven er først og fremmest at se på kundens krav til produktet, og udarbejde en detaljeret kravspecifikation til programmet. Dette gør vi for at sikre at vi har forstået kundens krav og for at tilføre andre funktionelle og supplerende krav til spillet, der vil være til gavn for kunden. Inden vi begynder på at kode vil vi også arbejde med use case modellering, domænemodel og aktivitetsdiagram. Ud fra dette vil vi så programmere spillet i IntelliJ. Vi vil dokumentere vores tanker i selve implementeringen i form af java kommentare i source code. Efter færdigøring af vores spil vil vi også teste om terningskastet fungere korrekt i forhold til den teoretiske sandsynlighed.

3 | Kravspecifikation

Ud fra kundens vision, som vi har fået udleveret i vores opgavebeskrivelse, kan vi her opstille nogen krav og specifikationer, som vores program skal indeholde. Kunden vil gerne have et program, som kan spilles på maskinerne i DTU's databar, som kører på Windows operativsystemet. Herudover skal spillet kunne spilles af 2 personer, hvor de 2 personer spiller imod hinanden. De 2 personer skal hver især kunne slå med 2 terninger, hvorefter summen af terningernes værdi lægges til den individuelle spillers point. Spilleren skal med det samme kunne se resultatet efter et terningkast. Når en spiller opnår 40 point, og derefter slår 2 ens, har han vundet spillet. Derudover har vi i vores spil valgt at inkludere et par ekstraopgaver. Hvis en spiller slår to 1'ere, vil han miste alle sine point. Hvis en spiller slår 2 ens, vil han få en ekstra tur. Spilleren kan vinde spillet ved at slå 2 seks'ere i træk.
Ud fra vores opgavebeskrivelse, kan vi hurtigt konkludere at vi har med 2 interessenter at gøre, kunden og brugeren af vores spil. I vores situation, vil brugeren af vores spil være en direkte bruger, hvor kunden er en indirekte bruger. Derudover kan vi ud fra kundens vision, opstille de funktionelle og ikke-funktionelle kravspecifikationer, som vores program skal indeholde.

- To spillere skal kunne spille mod hinanden
- En spiller skal kunne slå med to terninger
- En spiller skal kunne se resultatet af et terningekast efter hvert kast
- En spillers kast bliver tilføjet til spillers samlet scorer
- Spilleren mister alle sine point hvis spilleren slår to 1'ere
- Spilleren får en ekstra tur hvis spilleren slår to ens
- Spilleren kan vinde spillet ved at slå to 6'ere, hvis spilleren også i forrige kast slog to 6'ere
- Spilleren skal slå to ens for at vinde spillet, efter at man har opnået 40 point
- Spillet skal kunne spilles på en Windows maskine

Ikke-funktionelle krav
Herudover har vi også identificeret et ikke funktionelt krav.

- Kunden vil gerne have at man kan se et kast med det samme

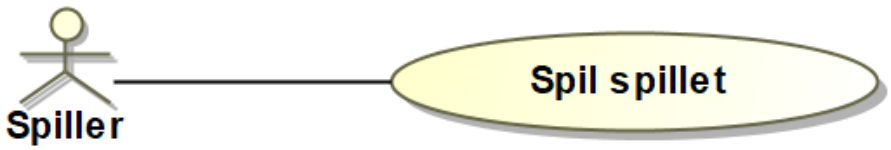
Vi har her valgt at omskrive dette krav, til at det ikke må tage mere end 333 millisekunder, før man kan se resultatet af et kast

Supplerende specifikationer
Derudover har vi ud fra opgave beskrivelsen også fået angivet et krav om, at almindelige mennesker, skal kunde spille spillet uden en brugsanvisning. Vi har her valgt at omskrive dette krav, da det er vanskeligt at imødekomme et krav vi ikke kan måle på. Vi har derfor valgt at opstille følgende krav til vores program, som skal sikre, at programmet er brugervenligt og nemt at spille.

- Programmet skal fortælle hvis tur det er
- Programmet skal fortælle hvad du skal trykke på for at kaste dine terninger

4 | Use case modellering

Da vi i vores tilfælde har med et simpelt terningspil at gøre, har vi her kun én use case i vores systemet, som er: Spil spillet.
Derudover definerede vi tidligere vores interessenter til at være brugeren af spillet, samt kunden. I tilfældet af vores system er det dog kun spilleren, som er direkte bruger, og derfor har vi kun én aktør, som vi i vores use case diagram, har valgt at kalde Spiller.



Da vi kun har én aktør og én use case, har vi herefter valgt at udarbejde et fully dressed brief til vores use case, for at gå i dybden med, hvordan vores aktør kommer til at anvende vores system.

Use case sektion:	Beskrivelse:
Navn	Spil spillet
Scope	Terning spil
Level/niveau	Brugermål
Primær aktør	Spiller
Andre interessenter	Kunden: Kunden forventer en terningespil.
Forudsætninger/ preconditions	To spiller sidder klar foran et Windows system, hvor terningespillet ligger på.
Succeskriterier/ postconditions	En af spillerne slår to ens, der ikke er to 1´er efter de har opnået en samlet scorer på over 40 point.
Vigtigste successcenarie	1. En spiller slår to ens, der ikke er to 1´er efter de har opnået en samlet scorer på over 40 point.
Udvidelser	1. Spilleren slår med terningerne og kastet bliver tilføjet til hans samlet scorer. 1.1 Hvis spilleren slår to ens, der ikke er to 1´er 1.1.1 Spilleren får en tur mere. 1.2 Hvis spilleren slår to 1´er. 1.2.1. Spillerens scorer bliver sat til 0 og han får en tur mere 1.3 Hvis spilleren ikke slår to ens 1.3.1 Det er nu den næste spillers tur 2. Spilleren opnår en samlet scorer på over 40 point. 2.1 Hvis spilleren slår to ens, der ikke er to 1´er 2.1.1 Spilleren vinder spillet. 2.2 Hvis spilleren slår to 1´er. 2.2.1 Spillerens scorer bliver sat til 0 og han får en tur mere 2.3 Hvis spilleren ikke slår to ens 2.3.1 Det er nu den næste spillers tur
Specielle krav	En computer med Windows, hvor spillet ligger på.
Frekvens	Hver gang man spiller spillet.

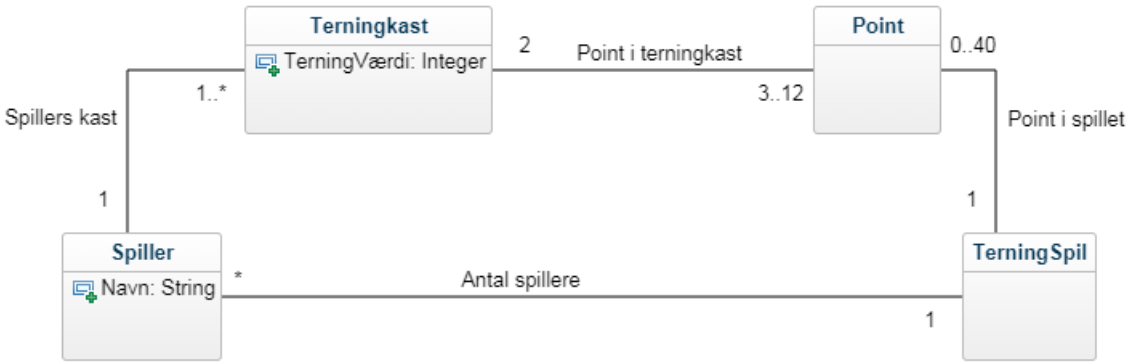
5 | Systemanalyse

Som en led i vores analyse- og designproces, har vi her fortaget en navneordsanalyse af vores fully dressed use case brief. Idéen med vores navneordsanalyse er at brainstorme alle de navneord, som senere hen i vores system, kan blive til klasser og attributes. Der er her vigtigt at understrege, ikke alle navneord som vi finder i analysen, nødvendigvis også kommer til at indgå i vores endelige system, men metoden her er med til at danne et overblik over hvordan vores system skal opbygges.

5.1 Navneordsanalyse og Domænemodel

Spil, terning, bruger, spiller, kunde, terning, point, kast, computer og scorer.

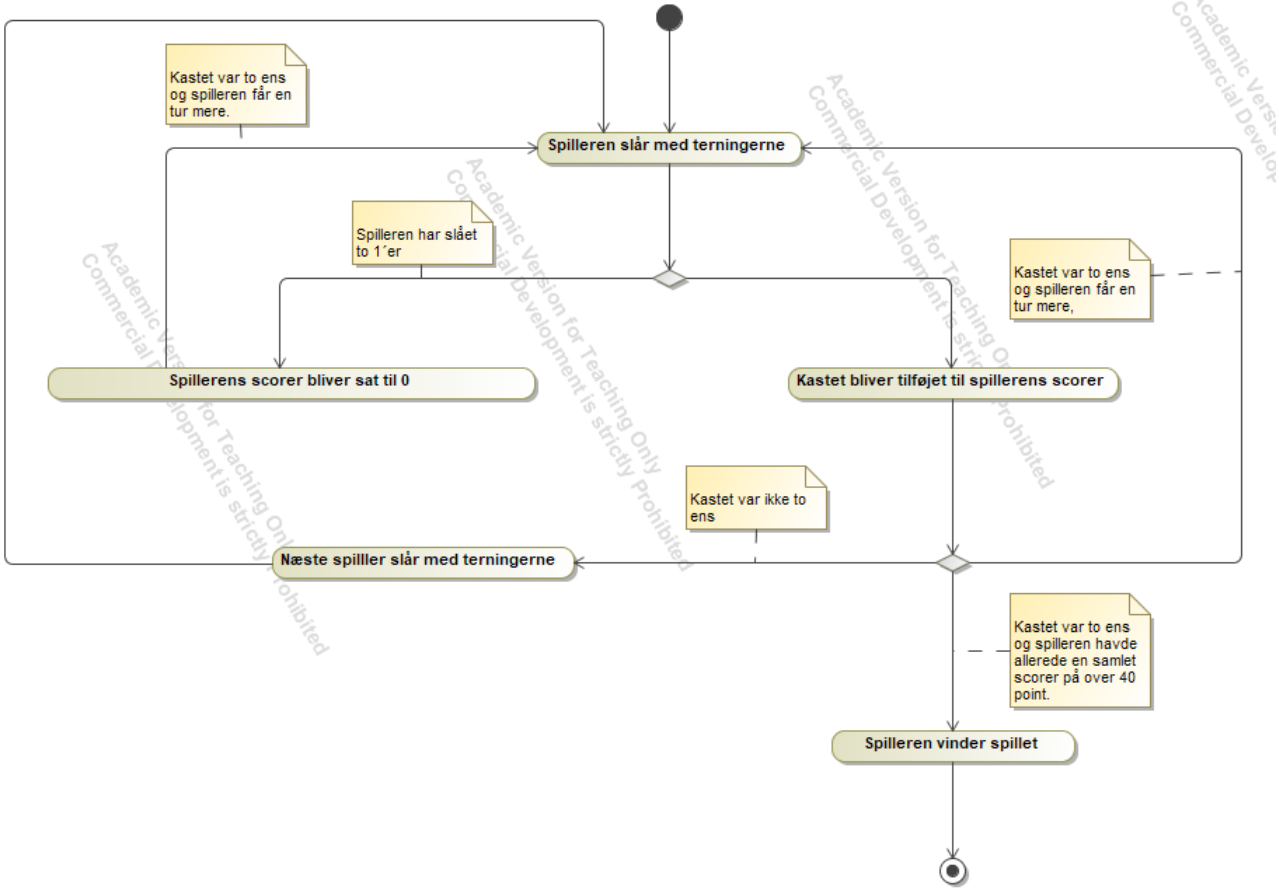
Vi har her på baggrund af vores navneordsanalyse udarbejdet en domænemodel, for hvordan vores system skal fungere. I denne domænemodel har vi angivet multipliciteten for relationerne imellem vores klasser, herudover har vi angivet 2 attributes, navn på spiller samt værdi af terningkast, som vi på forhånd er sikre på kommer til at indgå i vores system.



5.2 Aktivitetsdiagram

Vi har lavet et aktivitetsdiagram over vores eneste use case nemlig "spil spillet", der har følgende pre/post conditions.

Preconditions: To spiller sidder klar foran et Windows system, hvor terningespillet ligger på.
Postconditions: En af spillerne slår to ens, der ikke er to 1'er efter de har opnået en samlet scorer på over 40 point.



6 | Dokumentation

Der er en række interessante klasser i vores kode, dem der kommenteres på her er:

- MatadorMain
- Matador
- Player
- Die
- DiceCup

Kildekode for disse klasser kan findes i Billagene.
Vi har i vores kode fokuseret på at det kan udvides til at være hele matador spillet vi skal lave i løbet af dette CDIO projekt, og vi har derfor sat en nogle ting op som til dels kan være unødvendigt for denne opgave, men som vil gøre projektet lættere senere.

MatadorMain

Billag A.1: MatadorMain
Dette er klassen hvori vores main funktion er i, som er den der køre som det første i programmet. Den registrere om der er gevet nogle argumenter til programmet som bestemmer antallet af spillere og AI's, så kaldes funktionen $Init()$ fra $Matador$ klassen med det ønskede antal spiller, derefter kaldes $startGameLoop()$ fra $Matador$ klassen som er et loop der køre indtil spillet er slut.

Matador

Billag A.2: Matador
Denne klasse indeholder metoderne der initializere programmet, og main loopet, samt en metode *stop*() som bliver kørt når programmet afsluttes, som ikke bruges endnu.

Metoden *init*() laver et array af vores klasse *Player* med længde summen af antal spillere og antal AI's. Derefter instantiere den det rigtige antal spillere og AI's og lægger dem i arrayen.

Metoden *startGameLoop*() indeholder et while loop som køre så længe der ikke er nogle der har vundet spillet endnu. Hvert loop kaldes metoden *playerRollDice*() på den næste spiller, eller AI, i rækken, og spillet stopper hvis en af spillerne så vinder.

Player

Billag A.3: Player
Denne klasse indeholder 2 constructers, en der instantiere dette instance af klassen som en AI, og en der tager et argument som navnet på spilleren, og så instantiere dem som en reel spiller.

Der er en metode *calcHasWon*() som udregner om spilleren har vundet som kaldes efter hvert kast.

Yderligere er der metoden *playerRollDice*() som ruller tærningen ved at kalde metoden *roll*() fra klassen *DiceCup* for en *Player* eller en AI. Vha. *DiceCup* klassen findes ud af om spilleren har rullet 2 ens, om det var to etter eller 2 seksere og hvor meget summen af de to tærninger de rullede var. Hvis det var 2 ettere sættes *Player*'s score til 0, hvis det var 2 seksere, og det sidste ruld også var to seksere sættes de til at have vundet, hvis de rullede to identske tærninger får de et ekstre ruld, ved at metoden kalder sig selv.

Die

Billag A.4: Die
Denne klasse har 2 attributter, en *size* og en $facevalue$. *size* sættes i constructeren for klassen, og $facevalue$ sættes til et tilfældigt tal af mindst 1 og størst hvad *size* er, hver gang metoden *roll*() kaldes på tærningen.

DiceCup

Billag A.5: DiceCup
Denne klasse er en singleton, som indeholder statiske referencer til to forskellige *Die* med størelse 6. Denne klasse indeholder en række metoder til at håndtere de to tærninger.

roll()
: Denne metoder ruller de to tærninger og retunere hvad de rullede.

getDiceIntValues()
: Denne metode retunere summen af hvad de to tærninger har rullet i øjeblikket uden at rulle dem på ny.

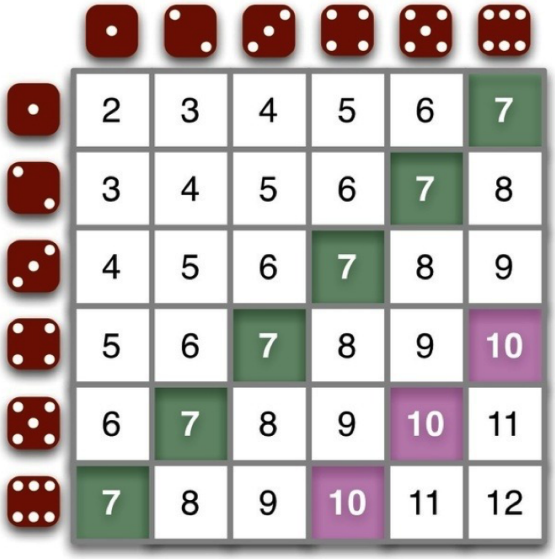
isSame()
: Retunere om *facevalue* på de to tærninger er identiske

isSameAndNum(*intnum*)
: Retunere *isSame*() og om de 2 tærninger har rullet tallet *num*

getDiceStringValues()
: Retunere en stræng der repræsentere værdien af de to tærninger.

7 | Test

Figur 7.1: Sandsynlighedstabel for 2 terninger. [1]



Vi vælger at øge test-kastene fra 1000 til 100000 for a få en mindre fejlmargin når vi skal lave en test. I den teoretiske verden, vil de forskellige summe forekomme:

"2" , 2777.7 $\sim$ 2778 gange
"3" , 5555.5 $\sim$ 5556 gange
"4" , 8333.4 $\sim$ 8343 gange
"5" , 11111.1 $\sim$ 11111 gange
"6" , 13888.8 $\sim$ 13889 gange
"7" , 16666.6 $\sim$ 16667 gange
"8", 13888.8 $\sim$ 13889 gange
"9", 11111.1 $\sim$ 11111 gange
"10", 8333.3 $\sim$ 8333 gange
"11", 5555.5 $\sim$ 5556 gange
"12", 2777.7 $\sim$ 2778 gange

Sum	2	3	4	5	6	7	8	9	10	11	12
Forventet antal gange	2778	5556	8333	11111	13889	16667	13889	11111	8333	5556	2778

Helt præcise tal får vi nok ikke, men vi skulle gerne få noget der var meget tæt på.

Vi gør det at vi med en JUnit test, ruller terningerne 100000 gange og får skrevet hvor meget hver sum forekommer og ser om de stemmer over ens med de teoriske med en fejlmargin på 4%.

Sum	2	3	4	5	6	7	8	9	10	11	12
Forventet antal gange	2806	5547	8114	11039	14169	16554	13840	11372	8272	5550	2737

Vi sætter tallene mod hinanden.

Sum	2	3	4	5	6	7	8	9	10	11	12
Forskel i hele tal	28	9	219	72	280	113	89	261	61	6	41
Forskel i procent	1.0079%	0.1622%	2.699%	0.6522%	2.016%	0.6826%	0.354%	2.349%	0.7374%	0.1801%	1.498%

Vi kan se at vi ikke har en differens på er over 4% "forkert". Dermed kan vi konkludere at programmeret fungerer godt med hensyn til de teortiske sandsynligheder.

For at være helt sikre, har vi kørt vores tests flere gange. Selv med en kast testet på 100000, sker det (ca. 1/100) at der vil forkomme en fejlmargin på over 4%. Dette skyldes at vi arbejder med tilfældigheder. Det kunne forekomme at vi slog 100000 * 2 (dvs. to ettere hundrede tusinde gange), men sandsynligheden er ubeskrivelig lille.

8 | Konklusion

Vi er nu noget til projektets ende og kan nu konkludere på vores endelige resultat. Vi har formået at tilfredsstille de krav og forventninger kunden havde til vores produkt. Samtidigt fik vi fortolket og revideret på kundens kravspecifikationer, så vi kunne levere det bedst mulige produkt.

Vi kom i vores analyse frem til den fremgangsmåde vores spil skulle bygge på, og fik styr på vores interessenter, aktører klasser osv.

Til sidst fik vi så udviklet vores spil, og udførte test for at se om det fungerede som vi forventet. Testene gik som håbede om faldt indenfor vores fejlmargin på 4% dog observerede vi også at der er meget sjældne tilfælde skete et lille udfald i forhold til vores fejlmargin, dog skyldes dette at vi arbejder med tilfældigheder, som godt kan ligge uden for det forventede dog er sandsynligheden meget lille.

Bibliografi

[1] Tim Erickson. *an empirical approach to diceprobability*. 2011. URL: https://bestcase.wordpress.com/2011/01/08/an-empirical-approach-to-dice-probability/.

A | Kildekode

A.1 MatadorMain

```
public class MatadorMain {
        /**
         * The function that runs when the program is executed,
         * it registrers wheter any arguments are given,
         * then initializes the game with given arguments if there are any,
         * otherwise it uses the default values,
         * then the main game loop is started,
         * and when that exits it closes the program
         * @param args
         * @throws NumberFormatException
         * @throws IOException
         */
        public static void main(String[] args)
                                throws NumberFormatException, IOException {
                boolean defaultMode = args.length < 1;
                int defaultPlayers = 2;
                boolean defaultAIMode = args.length < 2;
                int defaultAIPlayers = 0;
                try {
                        Matador.init(defaultMode ?
                                defaultPlayers : Integer.valueOf(args[0]),
                        defaultAIMode ?
                                defaultAIPlayers : Integer.valueOf(args[1]));
                } catch (Exception e) {
                        Matador.init(defaultPlayers, defaultAIPlayers);
                }
                Matador.startGameLoop();
                Matador.stop();
        }
}
```

A.2 Matador

```
public class Matador {
        private CustomStreamTokenizer cst = new CustomStreamTokenizer();
        private static boolean playing = true;
```

```
private static Player[] players;

/**
 * The init function initializes the game with given number of players, AI's
 * and initializes any static classes necessary
 * @param numPlayers
 * @param AIs
 * @throws IOException
 */
public static void init(int numPlayers, int AIs) throws IOException {
        CustomStreamTokenizer.initTokenizer();

        players = new Player[numPlayers + AIs];
        for (int i = 0; i < numPlayers; i++) {
                System.out.print("Indtast navn paa spiller "+(i+1)+": ");
                players[i] = new Player(CustomStreamTokenizer.nextString());
        }
        for (int i = 0; i < AIs; i++) {
                players[numPlayers + i] = new Player();
        }
}

/**
 * The main game loops, that indefinitely runs through each player
 * until one of the players (or AI's) has won
 * @throws IOException
 */
public static void startGameLoop() throws IOException {
        int turns = 1;
        int currPlayer = 0;
        while (playing) {
                players[currPlayer].playerRollDice();
                if (players[currPlayer].hasWon())
                        endGame();
                if (++currPlayer >= players.length) {
                        turns ++;
                        currPlayer = 0;
                }
        }

}

/**
 * The stop function that runs as the very last thing in the game
```

```
         * in case any objects needs to be closed or anything similar.
         */
        public static void stop() {
        }

        /**
         * The function that is run when a player has won the game
         * and the game loop needs to stop.
         */
        private static void endGame() {
                playing = false;
        }
}
```

A.3 Player

```
public class Player {
        private String name;

        private boolean isAI = false;
        private static int aiNum = 1;

        private boolean lastTwoSixes = false;
        private boolean hasWon = false;
        private boolean lastOverWinscore = false;
        private static final int winScore = 40;
        private int score = 0;

        /**
         * @param Player name
         * @return A {@code Player} instantiated as an actual player
         */
        public Player(String name) {
                this.name = name;
        }

        /**
         * @return A {@code Player} instantiated as an AI
         */
        public Player() {
                this.name = "AI " + aiNum;
                aiNum++;
                this.isAI = true;
```

```
}

public void setAI(boolean isAI) {
        this.isAI = isAI;
}

public String getName() {
        return name;
}

public boolean hasWon() {
        return hasWon;
}

/**
 * Calculates whether {@code This} instance of the {@code Player} has won
 * and set's their hasWon tag respectively.
 */
private void calcHasWon() {
        if (score >= 40) {
                if (!lastOverWinscore) {
                        lastOverWinscore = true;
                }
                else if (DiceCup.isSame()) {
                        System.out.println(name + " has won the game");
                        hasWon = true;
                }
                if (!hasWon)
                        System.out.println
                            (name + " now has a score of over 40 and
                            only need to roll two of the same dice to win");
        }
}

/**
 * The function that should be called every time an action is required of a
 * {@code Player}, also works if the {@code Player} is an AI.
 * @param msg
 * @throws IOException
 */
public void playerRollDice() throws IOException {
        System.out.println(" ");
        int roll = DiceCup.roll(); //rolls the dice and saves the value
        score += roll;
```

```
if (DiceCup.isSameAndNum(1)) //If both dice shows 1 score gets reset
        score = 0;
if (isAI) { //If player is an AI rolls automatically
        System.out.println(name + " is rolling their dice");
} else { //If player is an actual player
         //it waits for an input from the player
        System.out.print
            ("It's your turn to roll "+name+" press enter to roll");
        CustomStreamTokenizer.waitForInput();
}
//Prints information to the player
System.out.println
    (name+" rolled "+DiceCup.getDiceStringValues()+
    " for a total of "+DiceCup.getDiceIntValues());
System.out.println(name + " now has a total score of " + score);
if (DiceCup.isSameAndNum(1)) {
    //Prints information to player if their score got resat
        System.out.println
            (name + " rolled two one's
            and gets their score reset to 0");
        lastOverWinscore = false;
}
//calculates whether the player has won now
calcHasWon();
if (DiceCup.isSameAndNum(6)) {
        //If both dice rolled a 6  it checks for if they rolled two
        //sixes last time and if they did they won
        if (lastTwoSixes) {
                System.out.println
                    (name + " has rolled two sixes
                    twice in a row, and hereby wins the game");
                hasWon = true;
        }
        else {
                System.out.println
                    (name + " has rolled two sixes in one roll,
                    if they do it again they win the game");
                lastTwoSixes = true;
        }
} else
        lastTwoSixes = false;

//gives player an extra turn if they haven't won,
//and they rolled two identical Dice
```

```
                if (DiceCup.isSame() && !hasWon) {
                        System.out.println
                            (name + " rolled two identical dice
                            and get's another roll");
                        playerRollDice();
                }
        }
}
```

A.4 Die

```
public class Die {
        private int size;
        private int faceValue;

        /**
         * Initializes a {@code Die} with a set size and rolls a value for the die
         * @param size
         */
        public Die(int size) {
                this.size = size;
                roll();
        }

        /**
         * Rolls a new random value for the die based on the our
         * {@code CustomRandom} class
         * @return Returns a random number which the die can roll
         */
        public int roll() {
                faceValue = CustomRandom.randInt(size);
                return faceValue;
        }

        public int getSize() {
                return size;
        }

        public int getFaceValue() {
                return faceValue;
        }

        public void setFaceValue(int faceValue) {
```

```
            this.faceValue = faceValue;
        }

        @Override
        public String toString() {
            return
                "This dice has " + size + " sides and
                currently has rolled a " + faceValue;
        }
}
```

A.5 DiceCup

```
public class DiceCup {
        private static final DiceCup INSTANCE = new DiceCup();

        private static Die d1 = new Die(6);
        private static Die d2 = new Die(6);

        private DiceCup() { }

        public static DiceCup getInstance() {
            return INSTANCE;
        }
        /**
         * Rolls the two dice in the DiceCup
         * @return Returns the sum of the two rolls
         */
        public static int roll() {
            return d1.roll() + d2.roll();
        }

        /**
         * @return Returns the sum of the current rolls on the two dice
         * without re-rolling them
         */
        public static int getDiceIntValues() {
            return d1.getFaceValue() + d2.getFaceValue();
        }

        /**
         * @return Returns whether the value shown on the two dice are the same
         */
```

```
        public static boolean isSame() {
                return d1.getFaceValue() == d2.getFaceValue();
        }

        /**
         * @param num
         * @return Returns whether the value on the two dice
         * and the same and have the specified {@code num} shown on them
         */
        public static boolean isSameAndNum(int num) {
                return isSame() && d1.getFaceValue() == num;
        }

        /**
         * @return Returns the current value of the first {@code Die}
         */
        public static int getD1Val() {
                return d1.getFaceValue();
        }

        /**
         * @return Returns the current value of the second {@code Die}
         */
        public static int getD2Val() {
                return d2.getFaceValue();
        }

        /**
         * @return Returns a string representing the value of the two dice
         */
        public static String getDiceStringValues() {
                return "a " + d1.getFaceValue() + " and a " + d2.getFaceValue();
        }
}
```