ASSIGNMENT 4: GAME PROJECT

Tic Tac Toe

Sammanfattning

En rapport som sammanfattar skapandet av ett av världens mest klassiska brädspel.

Viktor Salmi
A18viksa
Abukar Ahmed
A18abuah

Innehållsförteckning

1. Introduction

Uppgiften som tilldelades gick ut på att skapa ett interaktivt spel med ett grafiskt användargränssnitt. Det första som gjordes var att försöka komma på en bra idé för ett spel. Ett spel som redan finns som dessutom inte är alltför stort eller komplext.
Beslutet landade i Luffarschack, även känt som Tic Tac Toe på engelska.
Första steget efter beslutet om vilket spel som skulle skapas var att bestämma hur spelet skulle kunna se ut, hur det skulle fungera, vilka regler spelet skulle ha samt vilka klasser som skulle kunna vara logiska att använda för ett spel som Tic Tac Toe.
Grundtanken var att sätta upp egna krav som spelet skulle följa samtidigt som även uppgiftskraven uppfylls på ett så korrekt sätt som möjligt.
Rapporten för projektet har som syfte att beskriva och förklara för en utomstående exakt vad projektet går ut på samt hur det har skapats. Detta kommer även förtydligas extra med hjälp av diverse diagram.
Rapporten som innehåller dessutom ett Use-case diagram, en Use-case text och ett klassdiagram som finns till för att beskriva hur programmet är uppbyggt och hur det fungerar.

2. Requirement Specification

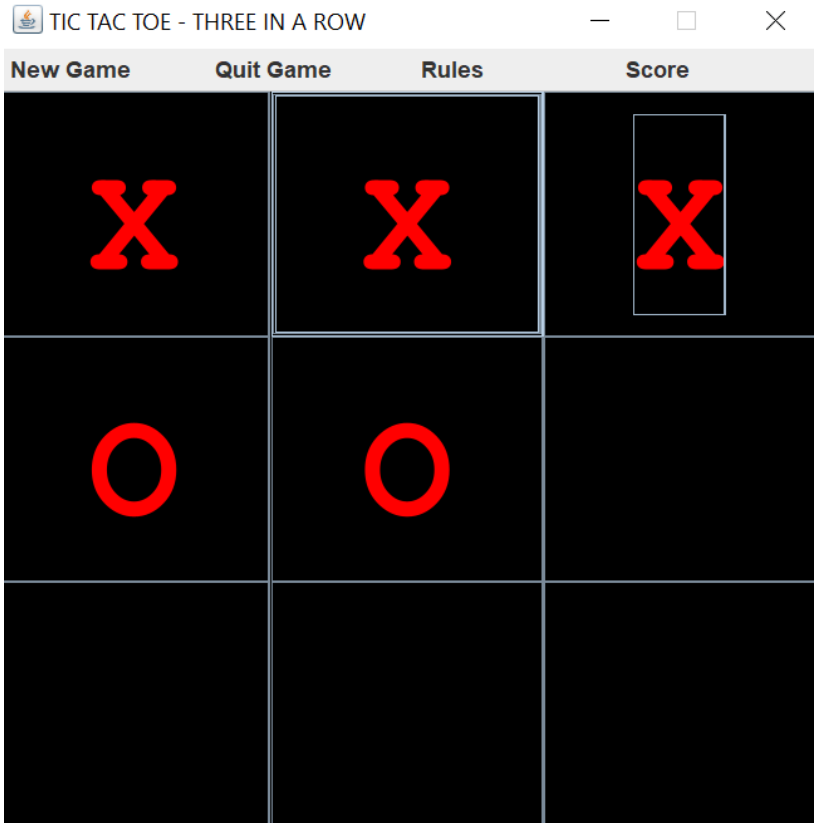
Det måste alltid finnas två spelare i en omgång av Tic Tac Toe. En som kontrollerar X och en som kontrollerar O. Det kan bara sluta med att en av spelarna vinner. Om det blir lika delas inga poäng ut och spelet startas om. Ett Use-case diagram har skapats för programmet där det illustreras exakt vad en spelare kan göra med spelet.

2.1 Presentation

De grundläggande kraven som fastställs för projektet är att det som skapas ska vara ett spel av något slag. Exempelvis ett brädspel eller ett kortspel. Spelet ska kunna interagera med en användare på så sätt att användaren kan få saker att hända när den spelar spelet. Spelet ska även ha ett grafiskt användargränssnitt som gör att spelet blir både snyggare och lättare att interagera med.
Till sist ska spelet även ha ett tydligt mål. Det kan förslagsvis vara att det finns ett sätt att vinna spelet på. Då blir målet helt enkelt att vinna mot en motståndare, vare sig det är en dator som är motståndare eller en annan människa som är motståndare.
Spelet som har skapats är ett väldigt grundläggande luffarschack. Luffarschack är ett spel som utspelar sig på en bräda där det finns nio olika rutor. Varje ruta är från början tom och syftet med spelet är helt enkelt att X möter O. För att vinna måste antingen X eller O placeras i rad tre gånger. När det sker får vinnaren en poäng.

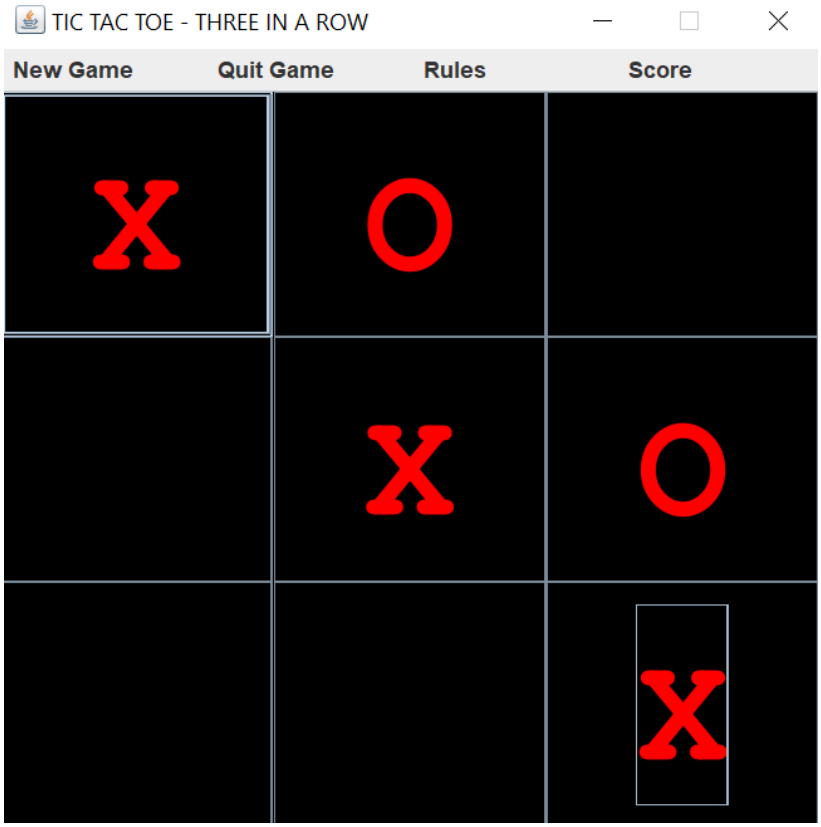
2.2 Rules

För att spelet ska kunna fungera på ett bra och tydligt sätt måste det finnas regler. Utan regler kommer ingen någonsin kunna vinna spelet. Den första och mest grundläggande regeln är att en användare bara får göra ett drag i taget. När den första spelaren lägger ut sitt X blir det den andra spelarens tur att lägga ut sitt O. För att kunna säkerställa att den här regeln följs har det skapats en metod som gör att spelet skiftar mellan X och O efter varje gång en spelare interagerat med spelbrädet.
För att kunna vinna spelet måste det även finnas regler för hur det uppstår en segrare. En spelare vinner som bekant om den får tre X alternativt tre O i rad.
Detta kan ske vertikalt, horisontellt och diagonalt. Det finns med andra ord flera olika varianter en spelare kan använda sig av för att vinna spelet.
Figur 1, 2 och 3 visar alla olika sätt en spelare kan vinna en omgång på.



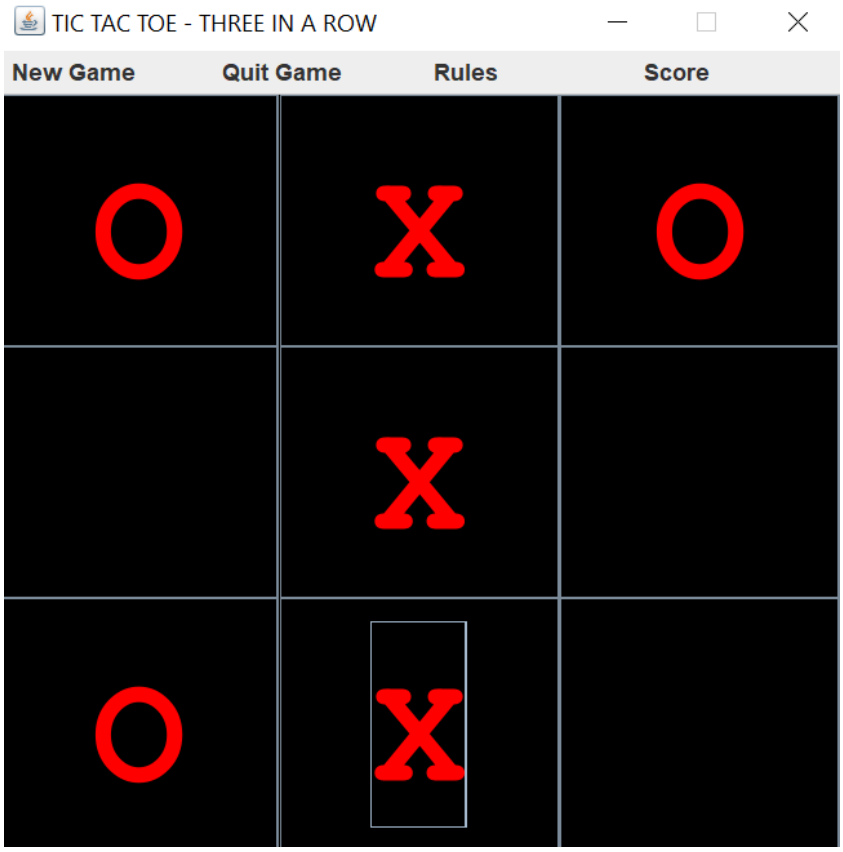
Figur 1: Horisontellt

Spelaren kan vinna omgången genom att få tre i rad horisontellt.



Figur 2: Diagonalt

Spelaren kan vinna omgången genom att få tre i rad diagonalt.



Figur 3: Vertikalt

Spelaren kan vinna omgången genom att få tre i rad vertikalt.

2.3 User Interaction

En användare interagerar med spelet genom att använda datormusen för att klicka på en tom ruta i spelbrädet. En användare kan även starta om spelet/starta ett nytt spel genom att klicka på ”New Game” och det går dessutom avsluta spelet helt och hållet genom att klicka på ”Quit Game”. Om en ny spelare skulle vara lite osäker kring hur spelet fungerar kan den klicka på ”Rules” där det framgår hur spelaren vinner, hur spelets logik ser ut samt hur den får poäng och därmed vinner. När en spelare vinner en runda får den ett poäng och detta går att se ifall ”Score” klickas på. Väl därinne illustreras det hur många poäng X har fått samt hur många poäng O har samlat på sig.

2.4 Functional Requirements

- En spelbräda skapas genom att kompilera koden.

- När en spelare klickar på en tom ruta ska det läggas till en symbol på den ruta där ett klick har skett.

- En spelare ska vinna om den får tre symboler i rad antingen horisontellt, vertikalt eller diagonalt.

- En spelare ska kunna starta ett nytt spel, läsa om spelets regler, titta på den aktuella poängställningen samt stänga ner spelet.

2.5 Non - Functional Requirements

- Spelet ska vara enkelt att förstå.

- Spelet ska ha tydliga regler.

- Det ska vara tydligt vem som vinner.

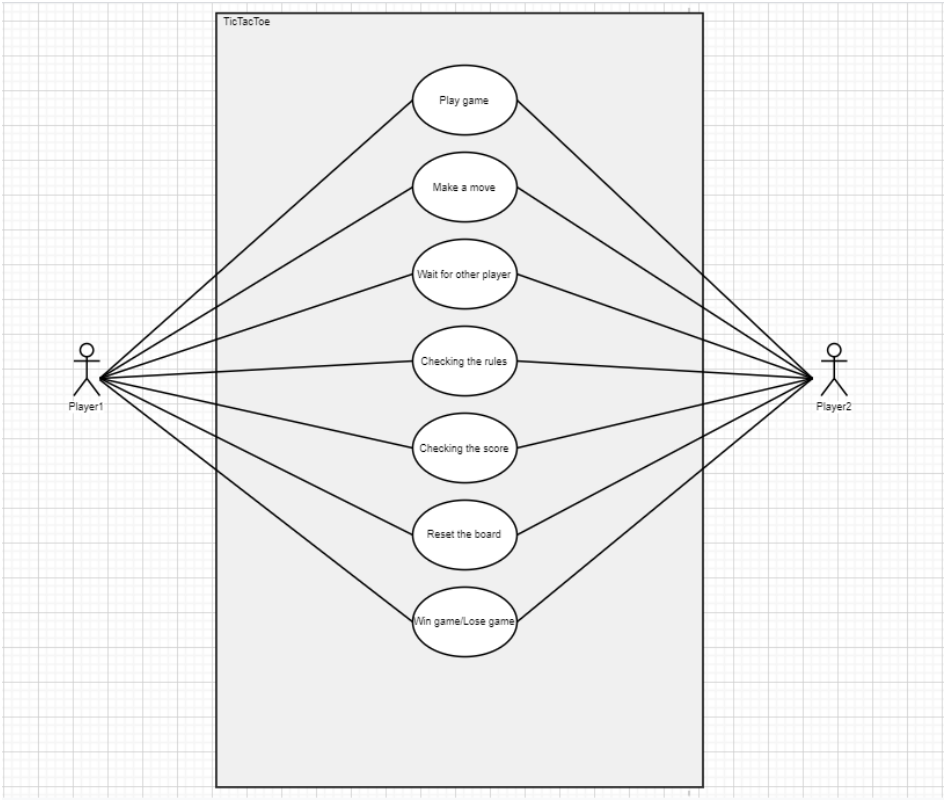
- Poängställningen ska vara enkel att följa för att se vem som leder.

- Användargränssnittet ska vara enkelt att använda.

- Spelet ska inte krascha på grund av några större buggar i programmet.

2.6 UML Use-Case Diagram

Ett Use-case diagram skapas för att illustrera vad en aktör(spelare) kan göra med spelet i form av funktionalitet.



Figur 4: Use-case diagram

2.7 UML Use-Case Text

Use-case texter tas fram för att dela ut ID till varje Use-case samt förklara vilken aktör som är kopplad till Use-case diagrammet.
Det finns även en beskrivande text till varje Use-case som beskriver funktionaliteten.

Use-Case ID	UC-1
Use-Case Name	Play game
Actor	Player1 & Player2
Description	Spelaren ska kunna starta igång spelet och sedan spela det.

Use-Case ID	UC-2
Use-Case Name	Make a move
Actor	Player1 & Player2
Description	Spelaren får möjligheten att kunna bestämma sitt nästa drag.

Use-Case ID	UC-3
Use-Case Name	Wait for other player
Actor	Player
Description	Spelaren måste vänta medan den andra speldeltagaren gör sitt drag.

Use-Case ID	UC-4
Use-Case Name	Checking the rules
Actor	Player

Description	Möjligheten att kunna läsa om spelets regler.

Use-Case ID	UC-5
Use-Case Name	Reset the board
Actor	Player
Description	Rensar spelbrädan efter föregående omgång.

Use-Case ID	UC-6
Use-Case Name	Win game
Actor	Player
Description	När spelaren får tre i rad vinner den omgången.

Use-Case ID	UC-6
Use-Case Name	Lose game
Actor	Player
Description	När motståndaren får tre i rad förlorar spelaren omgången.

Use-Case ID	UC-7
Use-Case Name	Checking the score

Actor	Player
Description	Möjligheten att kunna se den aktuella poängstatistiken.

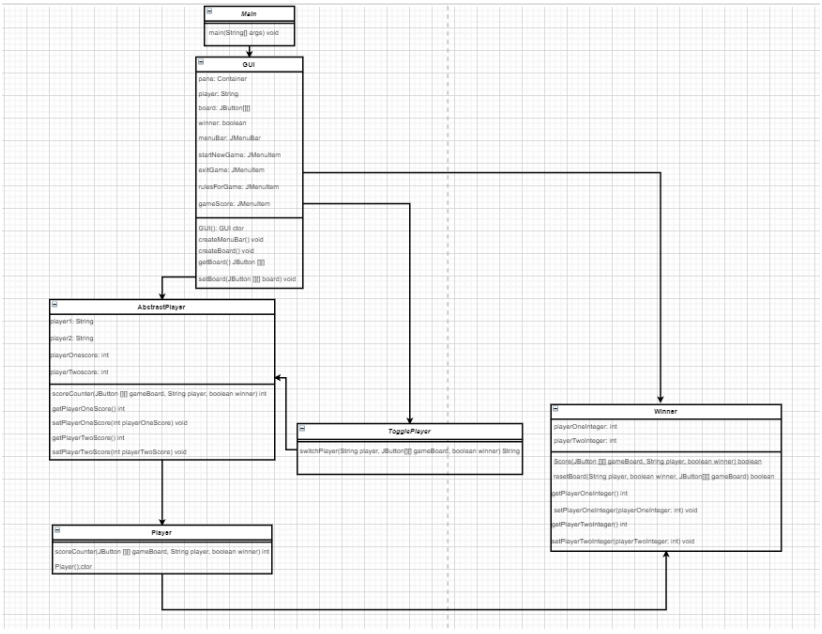
3. Design

3.1 Introduction

Till programmet har det skapats sex olika klasser där varje klass har sitt specifika syfte. Det finns en Main-klass där koden exekveras vilket leder till att programmet startar igång.
Det finns även en abstrakt klass (AbstractPlayer) som agerar som ”parent” till klassen Player.
Utöver de nämna klasserna finns det även klass för att kunna skifta mellan X och O i spelet (TogglePlayer) och det finns dessutom en klass (Winner) som avgör vilken spelare det är som vinner i spelet genom att jämföra positioner på spelbrädet.
Slutligen finns det även en klass som uteslutet existerar för att skapa spelets grafik.

3.2 UML Class Diagram

Ett klassdiagram skapas för att kunna se hur alla de klasser som har skapats är relaterade till varandra. Klassdiagrammet skapas för att ge en tydligare bild över hur programmet ser ut och hur allting hänger ihop.



Figur 5: Klassdiagram

3.3 Components

Huvudklassen Main startar igång spelet och kopplas direkt till klassen GUI. Väl i klassen GUI skapas spelets grafik såsom spelbrädet.
GUI kopplas till klasserna Winner, TogglePlayer och AbstractPlayer då GUI tillkallar metoder ifrån dessa.
AbstractPlayer kopplas till Player då den är en abstrakt klass som agerar som ”parent” till Player. Player är i sin tur kopplad till klassen Winner då den tillkallar en metod därifrån.

Main

Syftet med Main-klassen är att exekvera programmet som gör att en spelare kan spela spelet som har skapats.

GUI

Syftet med GUI-klassen är att skapa ett grafiskt användargränssnitt där exempelvis spelbrädet, menyn samt alla knappar skapas.
Den här klassen tillkallar de metoder som har med spelets logik att göra.

Winner

Det finns två mål med klassen Winner. Det finns en metod som räknar ut vem som vann spelomgången och det finns även en metod som rensar spelbrädet genom att tilldela en tom sträng till alla ifyllda rutor.

TogglePlayer

Syftet med klassen TogglePlayer är att den ska skifta mellan spelare 1 och spelare 2 i varje omgång som spelas. Första draget ska med andra ord bli ett X medan nästa drag ska bli ett O. Denna logik ska följas genom alla spelomgångar.

PlayerAbstract

Denna abstrakta klass är till för att skapa både abstrakta och icke abstrakta metoder. Abstraktionen används för att gömma undan detaljerna bakom implementationen och enbart visa funktionalitet.
I den abstrakta klassen har det skapats metoder som sedan ärvs av klassen Player.

Player

Klassen Player har som syfte att räkna ut poängen till player 1 och player 2 under alla omgångar som spelas.
I denna klass initieras även namnen (X och O) för player 1 och player 2 som sedan skrivs ut på spelbrädet.

4. Execution and Analysis

När spelet väl har skapats och känns helt färdigt måste det analyseras ifall alla krav som har fastställts har blivit uppfyllda.

Det första kravet som behandlas i koden är det krav som säger att spelet ska kunna exekveras och därmed bli spelbart. Detta sker när klassen GUI tillkallas i run-metoden som existerar i Main-klassen.

```
import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args) {
        // SwingUtilities takes care of updates that happens in the GUI
        SwingUtilities.invokeLater(new Runnable() {

            @Override
            // the method run starts the GUI
            public void run() {
                new GUI();

            }
        });


    }

}
```

Figur 6: Main

För att kunna få fram menyraden i JFramen måste SwingUtilities.invokeLater användas då det ser till att hantera all uppdatering som sker i det grafiska användargränssnittet. I och med detta skapas även en ny Runnable som skapar en run-metod där GUI kallas på.

Ett annat krav som togs fram var att spelaren ska kunna utföra ett drag på spelbrädet.

```
if (((JButton) e.getSource()).getText().equals("") && winner == false) {

    // giving the return value of the method machine to the player string
    player = machine.switchPlayer(player, gameBoard, winner);
    // setting the value of the player string to the JButton
    button.setText(player);

    // calling the method that calculates the score and gives it to player1 and player2
    players.scoreCounter(getBoard(), player, winner);
}
```

Figur 7: If-sats

En if-sats skapas för att ta reda på vem som har utfört det senaste klicket på spelbrädet genom användningen av getSource som returnerar det senaste klicket.

Sedan testas det ifall det är en tom sträng för att på så sätt kunna ifall nästa drag är valid.
I if-satsen returneras även värdet som kommer ifrån metoden switchPlayer som skapas i klassen TogglePlayer. Samtidigt tilldelas knappen som klickas på värdet av strängen player.
Dessutom tillkallas metoden scoreCounter som räknar ut resultatet och tilldelar det till varje spelare.

Det finns även ett krav som säger att den ena spelaren måste vänta på sin tur medan den andra spelaren utför sitt nästa drag.

```
import javax.swing.JButton;

public class TogglePlayer {

    // method that shifts between X and O for every turn and returns a String value
    public String switchPlayer(String player, JButton[][] gameBoard, boolean winner) {
        // parent class creating objects of the child classes Player1 and Player2
        AbstractPlayer players = new Player();


        if (player == players.player1) {
            player = players.player2;
        } else {
            player = players.player1;
        }

        return player;
    }

}
```

Figur 8: TogglePlayer

I klassen TogglePlayer skapas det en metod som heter switchPlayer som byter symbol mellan player1 och player2 beroende på vems tur det är att spela.
Därmed elimineras möjligheten att kunna utföra två drag direkt efter varandra.
Spelets logik ska se till att varje spelare bara får göra ett drag i taget.

I Use-caset har det även tagits med en funktionalitet angående hur spelarna ska kunna ta reda på spelets regler.

```
        // JMenuItem for the rules of the game
        rulesForGame = new JMenuItem("Rules");
        rulesForGame.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                // message dialog containing rules and explanation
                JOptionPane.showMessageDialog(null,
                        "Rules For The Game \n \n" + "To win you have to get three in a row.\n"
                                + "You win if you fill the board Vertically, Horizontally or Diagonally. \n"
                                + "For every win you will get one point.\n"
                                + "The loser of the round begins the next round.");

            }
        });
```

Figur 9: Rules

När knappen ”Rules” klickas på skapas det en messageDialog som innehar text som har med spelets regler att göra.

Ett krav har även tagits fram för att kunna föra poäng för varje omgång av spelet.

```
    // method that returns an integer of the score from PlayerOne
    public void scoreCounter(JButton[][] gameBoard, String player, boolean winner) {
        // creating an object of Winner

        winner = gameScore.score(gameBoard, player, winner);

        setPlayerOneScore(gameScore.getPlayerOneInteger()  );
        setPlayerTwoScore(gameScore.getPlayerTwoInteger()  );

        System.out.println("score1: " + getPlayerOneScore() + "score2:" + getPlayerTwoScore());

    }
```

Figur 10: scoreCounter

I klassen Player har det skapats en metod som heter scoreCounter.
Det som metoden gör är att först tillkalla en metod ifrån klassen Winner som räknar ut poäng för respektive spelare. Efteråt tilldelas poängen till den spelare som vunnit poängen. Poängen tilldelas med hjälp av setPlayerOneScore samt setPlayerTwoScore.

Det ska även finnas en möjlighet för spelaren att se den aktuella poängställningen. Detta sker i klassen GUI.

```
    gameScore = new JMenuItem("Score");
    gameScore.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {

            // message dialog that contains the current score
            JOptionPane.showMessageDialog(null,
                    "Player O: " + players.getPlayerOneScore() + "\nPlayer X: " + players.getPlayerTwoScore());

        }
    });

    // adds all the menu items to the menuBar
    menuBar.add(startNewGame);
    menuBar.add(exitGame);
    menuBar.add(rulesForGame);
    menuBar.add(gameScore);
    setJMenuBar(menuBar);
}
```

Figur 11: Score

Med hjälp av en ActionListener skapas det en klickbar knapp som öppnar en messageDialog som innehåller poängställningen.
I denna ActionListener finns getPlayerOneScore och getPlayerTwoScore som har blivit initialiserade i metoden scoreCounter som finns i klassen Player och som även blir tillkallad i klassen GUI.

För att kunna spela flera omgångar av spelet måste det finnas ett sätt att rensa brädan på innehåll efter varje omgång har avslutats.

```
public boolean resetBoard(String player, boolean winner, JButton[][] gameBoard) {
    player = "X";
    winner = false;
    // nested for-loop that clears the board with the use of an empty value
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            gameBoard[i][j].setText("");
        }
    }
    return winner;
}
```

Figur 12: resetBoard

I klassen Winner har det skapats en metod vid namn resetBoard vars funktion är att rensa spelbrädet. Det som sker är att med hjälp av nästlade for-loopar skapa nio nya knappar på spelbrädet och därefter tilldela knapparna en tom sträng.
Variabeln winner tilldelas false så att spelomgången inte fortsätter efter att någon av spelarna har vunnit. Denna boolean ser till att spelet inte fortsätter efter att någon av spelarna har vunnit. Detta returneras även som en boolean för att värdet ska ändras i klassen GUI.

```
startNewGame = new JMenuItem("New Game");
startNewGame.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // calling the resetBoard method from Winner that resets the board and gives the
        // return value of the boolean to winner boolean
        winner = toggle.resetBoard(player, winner, getBoard());
        System.out.println("new game  " + winner);
    }
```

Figur 13: Starta nytt spel

Till detta skapas även en knapp som ligger i menyraden. Knappens funktion är att starta ett nytt spel. Här tillkallas metoden resetBoard från klassen Winner samt även returnerar boolean-värdet till klassen Winner.

För att spelet ska kunna ha ett tydligt mål måste det finnas ett sätt att avgöra vilken spelare som har vunnit samt vilken spelare som har förlorat.

```
public boolean score(JButton[][] gameBoard, String player, boolean winner) {
    // if getText is equal to the current player
    // also comparing indexes
    if (gameBoard[0][0].getText().equals(player) && gameBoard[1][0].getText().equals(player)
            && gameBoard[2][0].getText().equals(player)) {
        JOptionPane.showMessageDialog(null, "Player " + player + " has won this round");
        playerintcalculator( player,  winner);
        winner = true;


    } else if (gameBoard[0][1].getText().equals(player) && gameBoard[1][1].getText().equals(player)
            && gameBoard[2][1].getText().equals(player)) {
        JOptionPane.showMessageDialog(null, "Player " + player + " has won this round");
        playerintcalculator( player,  winner);
        winner = true;
```

Figur 14: Bestäm vem som vinner

För att avgöra vinnaren av varje omgång har det skapats en metod som heter score. Där i implementeras flertalet if-satser vars funktion är att gå igenom varje index som

spelbrädet består av. If-satsen kollar på spelbrädets värde och även spelarens värde och jämför dessa. Ifall spelaren lägger ut sina symboler på indexen [0][0], [1][0] och [2][0] innebär det att hela den första raden är fylld av samma symboler vertikalt.
Det har dessutom skapats likadana if-satser fast varje if-sats har i uppgift att titta på sina tilldelade index för att kunna bestämma en vinnare i varje scenario där det förekommer tre av samma symboler i rad.
Boolean winner tilldelas true för att avsluta spelet i de fall som en spelare har fått tre i rad och därmed vunnit.

```
winner = toggle.score(getBoard(), player, winner);
System.out.println("toggle   " + winner);
// getSource returns the click and checks who made the click
// testing if it's an empty string in order to check if the next move is valid
if (((JButton) e.getSource()).getText().equals("") && winner == false) {
```

Figur 15: Rensa spelbrädet innan nästa omgång kan spelas

Först tillkallas funktionen i klassen GUI för att kunna räkna ut vinnaren av omgången. Metoden returnerar boolean-värdet som tilldelas till winner boolean. Detta gör att det inte går fortsätta spela efter att en spelare har vunnit. Därför måste spelbrädet rensas innan nästa omgång kan starta.

Överlag har alla fastställda krav blivit uppfyllda då kraven som togs fram var väldigt tydliga och enkla. Designen som togs fram är lämplig för sitt ändamål då allting är tydligt beskrivet och dessutom finns med i de olika diagrammen som har skapats.
De funktionella kraven i Use-case diagrammet är ett exempel på det.
I det stora hela har allting fungerat bra när det kommer till designen och utförandet. Däremot har kraven behövt omarbetas under arbetets gång då saker och ting inte har fungerat eller inte varit lämpliga för ändamålet. Ett exempel på det är en AI som från början fanns med i kraven. Det skulle med andra ord finnas två aktörer i Use-case diagrammet. En spelare och en AI.
Det som har fungerat mindre väl har varit att koppla samman de olika klasserna så lösningen inte bara blir objektorienterad utan även att metoder och klasser får en låg coupling och high cohesion.

Det som hade gjorts annorlunda ifall vi fått göra om hela projektet hade varit att vi börjat med en gång och på så sätt utnyttjat tiden till max. Nu har det blivit lite stressigt och det har med andra ord påverkat arbetet negativt. Det lades alldeles för mycket tid på vilket spel som skulle skapas. Men när vi väl satte igång gick det för det mesta smidigt och bra.

5. Summary

Tiden vi hade på oss för att göra denna uppgift var inte alltför lång vilket innebar att vi blev tvungna att på en gång försöka komma på en idé för ett spel. Vi ville skapa någonting som var relativt enkelt men samtidigt utmanade att skapa på ett objektorienterat vis. Vi visste dessutom att det skulle bli en hel del problematik med att samtidigt skapa en grafik till spelet.
Det första som gjordes efter att vi hade bestämt vilket spel vi skulle skapa var att bestämma vilka regler spelet skulle kunna tänkas ha. De regler vi kom fram till var att det ska finnas olika sätt att vinna på. Med andra ord att få tre i rad horisontellt, vertikalt och diagonalt. För att göra spelet logiskt behövdes även en regel för spelordningen som säger att en spelare spelar i taget och måste vänta på sin tur medan den andra spelaren utför sitt nästa drag.
Efter uppsättningen av regler började vi föreställa oss hur en design till spelet skulle kunna se ut. Detta lades ingen större tankekraft på detta eftersom det logiska sättet att skapa Tic Tac Toe på är att skapa en spelbräda bestående av 9 rutor. Till detta bestämde vi även att det skulle finnas en menyrad där det finns olika knappar som var och en har sitt eget syfte.

Innan vi började implementera spelet behövde vi först och främst skapa ett Use-case diagram med tillhörande textbeskrivning för att illustrera vem det är som ska använda programmet samt vad den användaren kan göra med programmet. Där skapade vi en aktör, spelaren, och kom på olika typer av funktionalitet som spelet behövde ha.
Ett klassdiagram skapades även för att lägga grunden för programmet, vilka klasser programmet skulle kunna ha samt hur de olika komponenterna är kopplade till varandra. Problemet med klassdiagram är att det har behövts justeras hela tiden då nya ändringar i koden har gjort. Det slutgiltiga klassdiagrammet består helt enkelt av alla klasser samt dess variabler och metoder.

När grunden för allting var fastställd var det dags att implementera det programmets kod. Under tidens gång har det kommit till nya tankar vilket har lett till en ny kodstruktur, nya klasser och även nya metoder.
Programmeringen av programmet har flutit på rätt smidigt men emellanåt blir det svårt att riktigt veta ifall helhetslösningen är objektorienterad och ifall den abstrakta klassen har implementerats på ett korrekt sätt. På grund av en del osäkerhet har vi lagt ner onödigt mycket tid på att hålla oss till det som faktiskt är objektorienterad programmering. Men tack vare den extra tiden som har lagts på det har förståelsen för objektorienterad programmering bara ökat.

Projektets utveckling har utförts relativt agilt då det hela tiden har existerat en bra kommunikation i gruppen där saker och ting har kunnat diskuteras på ett

bra och tydligt sätt vilket har tillåtit oss att kunna gå tillbaka och ändra saker och ting i koden och kraven men samtidigt bibehålla en bra struktur som inte har skapat alltför många nya problem.

Det slutgiltiga målet med spelet var att det skulle snyggt, lättbegripligt och framförallt bestå av minimalt med buggar för att kunna säkerställa att spelet hela tiden fungerar som det ska.
Däremot fanns det hela tiden med från början att det skulle finnas en spelare och en artificiell intelligens. Den artificiella intelligensen fick helt enkelt slopas då varken tid eller kunskap räckte till.
De idéer som fanns för en AI var att antingen skapa en mindre smart AI där den bara ser efter om det finns en tom ruta och därefter ockuperar den. Problemet med detta är att den AI som skapas inte besitter någon tydlig logik då den bara väljer en tom ruta med hjälp av en randomisering.
Den andra mer komplexa lösningen som hade skapat en AI med mer logik var att implementera en Minimax-algoritm som är en rekursiv algoritm som används för att ge tillåtelse till datorn att bestämma sitt nästa steg på ett logiskt sätt och inte med hjälp av en randomisering.
Detta hade skapat en utmaning för den mänskliga spelaren då Minimax-algoritmen faktiskt leder till att datorn kan vinna mot människan. Detta hade förhöjt spelupplevelsen extremt mycket.

I det stora hela känner vi att spelet fungerar på ett bra sätt utan större problem där programmet dessutom är skapat på ett objektorienterat vis. Som tidigare nämnt har detta simpla spel inte kommit utan sina problem vilket har gjort att projektet blivit en rolig men samtidigt krävande utmaning.

6. Collaboration

Detta projekt har utförts i par. Det som vi har gjort under projektets gång är att tillsammans komma överens hur vi ska lägga upp arbetet. Exempelvis har vi kodat tillsammans genom att använda ett tillägg som gör att två personer kan jobba i samma program samtidigt. Därefter har vi delat upp arbetet lite utefter vad vi själva tycker vi är bra på.
När det kommer till koden, diagrammen och själva planeringen har vi uteslutet arbetat tillsammans både på samma sak och enskilda saker. Men däremot har bara en av oss skrivit rapporten. Upplägget i rapportskrivningen blev som så att en skrev medan den andra tillade viktiga saker.
Genom hela projektets gång har vi kompletterat varandra på ett bra sätt där vi hela tiden haft bra kommunikation med varandra.