Beispielprojekt: TicTacToe

Ein Spiel, dass jeder aus seiner Jugend kennt. Mit der richtigen Strategie erreicht der Spieler, der beginnt, immer mindestens ein unentschieden. Bis man diese Strategie jedoch herausgefunden hat, dauerte es etwas. Wie ist es mit einem Computer als Gegenüber? Wie bringt man ihm am schnellsten und einfachsten eine Strategie bei, welche gut spielt, aber nicht zu gut, sodass man immer noch Spaß am Spiel hat?

Dieses Projekt ist sehr anspruchsvoll im Verständnis und in der Programmierung, sodass es eher für Leute gedacht ist, die bereits Programmiererfahrung vorweisen können.

Inhaltsverzeichnis / list of contents

Das Spiel

Das Konzept

Schön und gut. Was soll unser Programm können? Auf einer Oberfläche mit 9 Buttons (3x3 angeordnet) soll abwechselnd ein Spieler und der Computer ein "X" bzw. ein "O" auf den Button setzen, der geklickt wurde. (Dabei soll "X" beginnen) Wer eine volle Reihe, eine volle Spalte oder eine Diagonale bekommt, hat gewonnen und das Spiel endet. Wer anfängt, soll zufällig bestimmt werden.

Die graphische Oberfläche

Nach dieser kleinen Vorüberlegung kann man eigentlich schon loslegen:

Man öffnet Visual Studio und erstellt ein neues Projekt unter das links oben in der Ecke befindliche: **Datei Neu Projekt WPF-Anwendung**

Jetzt kann man das Projekt noch nach belieben benennen und danach sollte man sich vor einem weißen Fenster wiederfinden. Das ist der Baukasten, mit dem man das Fenster, die graphische Oberfläche des Spiels, erstellt. Wer will kann damit ein bisschen herumspielen, aber ich stelle auch einen Quellcode zur Verfügung:

XAML-Code

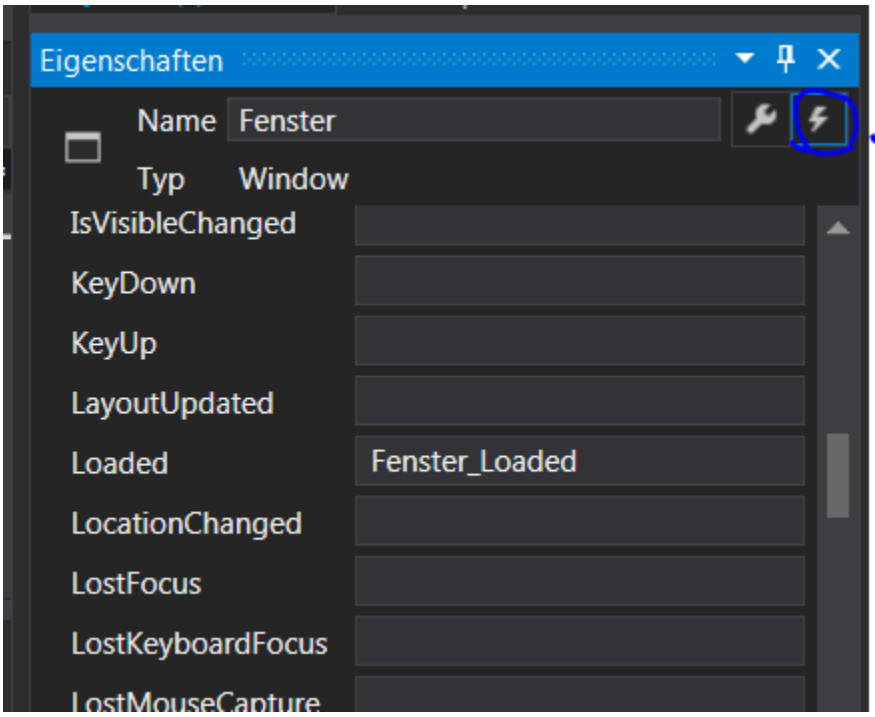
```
<Window x:Name="Fenster" x:Class="PROJEKTNAME_HIER_EINFUEGEN.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:PROJEKTNAME_HIER_EINFUEGEN"
        mc:Ignorable="d"
        Title="MainWindow" Height="500" Width="500" Background="#FFBABABA" Loaded="Fenster_Loaded">
    <Grid x:Name="Grid">
        <Canvas x:Name="Can" HorizontalAlignment="Left" Height="500" VerticalAlignment="Top" Width="500"
Loaded="Can_Loaded"/>
        <Button x:Name="B00" Content="" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
Margin="65,50,0,0" Height="75" Click="B00_Click" FontSize="48"/>
        <Button x:Name="B10" Content="" HorizontalAlignment="Left" Margin="65,191,0,0" VerticalAlignment="
Top" Width="75" Height="75" Click="B10_Click" FontSize="48"/>
        <Button x:Name="B20" Content="" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
Margin="65,332,0,0" Height="75" Click="B20_Click" FontSize="48"/>
        <Button x:Name="B01" Content="" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
Margin="207,50,0,0" Height="75" Click="B01_Click" FontSize="48"/>
        <Button x:Name="B11" Content="" HorizontalAlignment="Left" Margin="207,191,0,0" VerticalAlignment="
Top" Width="75" Height="75" Click="B11_Click" FontSize="48"/>
        <Button x:Name="B21" Content="" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
Margin="207,332,0,0" Height="75" Click="B21_Click" FontSize="48"/>
        <Button x:Name="B02" Content="" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
Margin="348,50,0,0" Height="75" Click="B02_Click" FontSize="48"/>
        <Button x:Name="B12" Content="" HorizontalAlignment="Left" Margin="348,191,0,0" VerticalAlignment="
Top" Width="75" Height="75" Click="B12_Click" FontSize="48"/>
        <Button x:Name="B22" Content="" HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"
Margin="348,332,0,0" Height="75" Click="B22_Click" FontSize="48"/>

    </Grid>
</Window>
```

Wer den Code übernimmt:

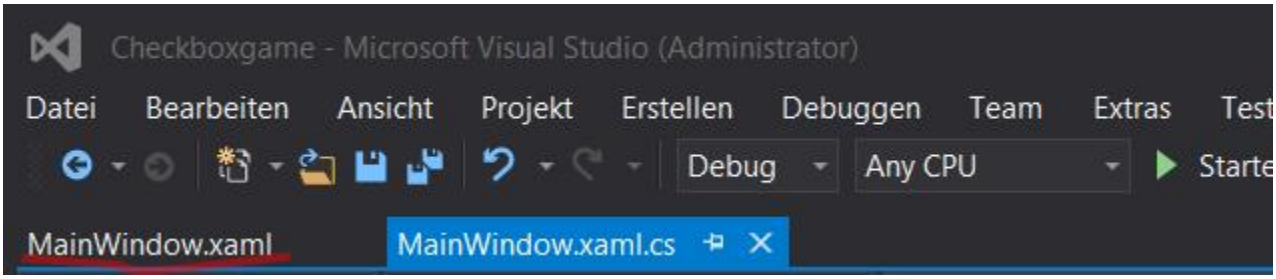
- Beachtet, überall wo "PROJEKTNAME_HIER_EINFUEGEN" steht, dies zu tun
- Farben etc. können nach belieben variiert werden, die Koordinaten der Objekte sollten jedoch gleich bleiben.

Wer selbst herumprobiert:

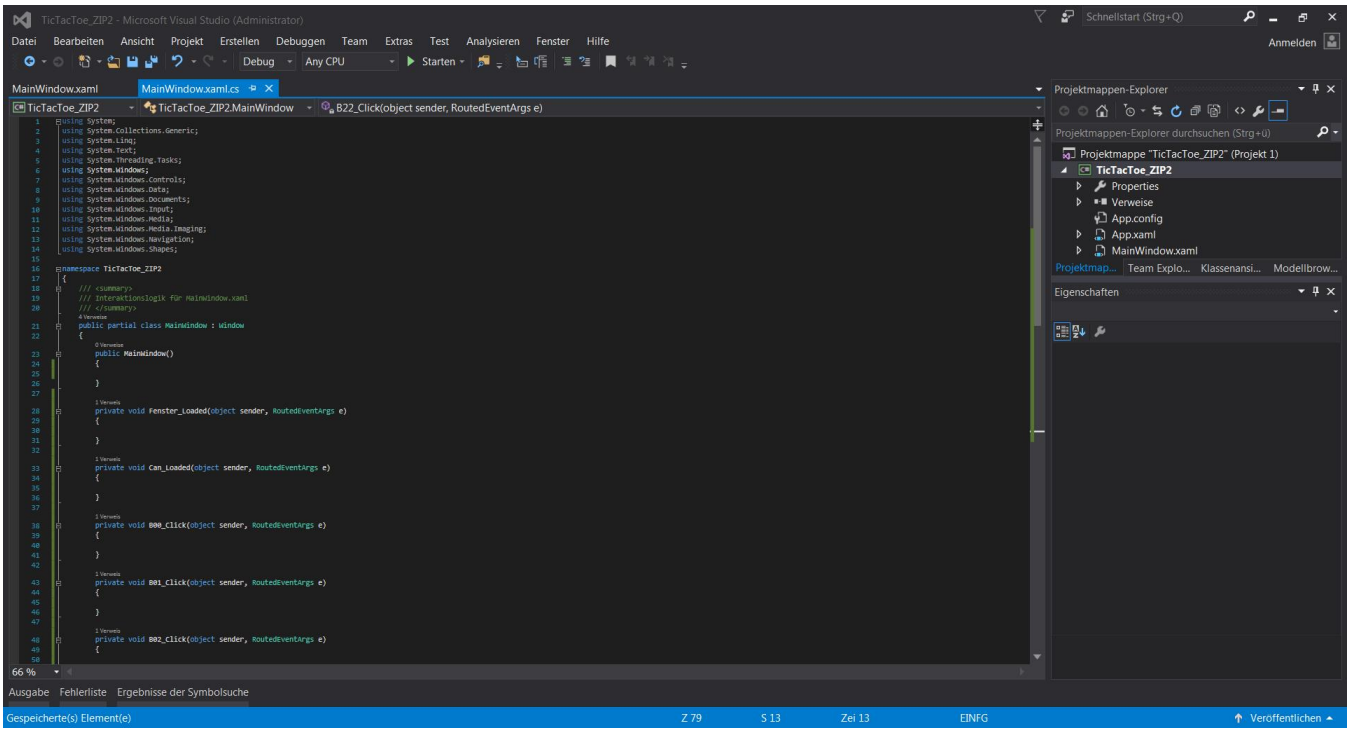
- Beachtet, dass ALLE eure Objekte Titel besitzen
- Euer Fenster sollte neun Buttons als Felder, sowie ein Canvas besitzen.
- In Folgenden Codeblöcken werde ich mich auf die Titel der im oberen Code deklarierten Objekte beziehen.
- Sowohl dem Fenster, als auch dem Canvas sollten die Ereignishandler "FENSTERNAMEN_Loaded", sowie "CANVASNAMEN_Loaded" hinzugefügt werden. Diese können erzeugt werden, indem man, wenn man das Objekt markiert hat, rechts neben der Textbox zum ändern des Titels auf dem Blitz geht und anschließend auf die Textbox neben "Loaded" doppelklickt.
- Selbiges gilt für die Methode "BUTTONNAME_Click" eines jeden Buttons.



Nebenbei, **wenn ihr die Methode mit Doppelklick wählt**: Visual Studio springt anschließend gleich zur entsprechenden Stelle im Code. Um zurück zur graphischen Ansicht zu kommen, könnt ihr ganz einfach oben den Tab zu "MainWindow.xaml" wechseln.



Ansonsten kommt ihr mit einem Druck auf F7 zum Code. Dort müsste danach in etwa folgendes zu finden sein:



Falls ihr die Methoden "Loaded" und "Click" der entsprechenden Objekte nicht findet, ergänzt diese bitte. Die Methodennamen sind immer nach folgendem Schema aufgebaut:

private void OBJEKTNAME_Loaded(object sender, RoutedEventArgs e) oder

private void OBJEKTNAME_Click(object sender, RoutedEventArgs e)

Dies wird bei allen Buttons, beim Fenster und beim Canvas gemacht.

Der Konstruktor der Klasse MainWindow ist meist sehr nützlich, jedoch in diesem Programm unnötig und kann problemlos entfernt werden.

Grundlegendes

Zu aller erst deklarieren wir uns ein zweidimensionales Bytearray "xOro", dass die Werte der Buttons greifbar machen soll. Dieses wird zunächst mit 0 initialisiert. Als nächstes wird eine Zufallszahl zwischen 1 und 2 gezogen, um zu entscheiden, ob der Spieler oder der Computer anfängt. (Der andere bekommt die andere Zahl) Dies hat den Vorteil, dass man nun auch wissen kann, wer die "X" und wer die "O" bekommt. (1 bekommt "X" und "X" macht den Anfang) Die Werte für den Spieler und den Computer werden dann an entsprechenden Stellen in das Array gesetzt, wenn die entsprechende Position von einer der beiden Parteien belegt wird. Außerdem brauchen wir noch ein zweidimensionales Buttonarray, dass mit den Buttons aus der Oberfläche initialisiert wird. Somit ergibt sich folgender Code:

Grundlegende Dinge

```
                public byte[,] xOro = new byte[3, 3];
        Random zgen = new Random();
        int spieler, ki;
        Button[,] but = new Button[3, 3];

                private void Fenster_Loaded(object sender, RoutedEventArgs e)
        {
                        //Ob der Spieler als erstes setzt, wird zufällig ermittelt
            spieler = zgen.Next(2) + 1;

                        //Aus der Startvariable des Spielers, wird die Startvariable des Computers ermittelt
(hier irrtümlicherweise als ki bezeichnet. Das, was hier programmiert wird, ist jedoch keine KI.)
            if (spieler == 1)
            {
                ki = 2;
            }
            else
            {
                ki = 1;
            }

                        //Das Bytearray wird mit 0 initialisiert
            for (byte i = 0; i < xOro.GetLength(0); ++i)
            {
                for (byte j = 0; j < xOro.GetLength(1); ++j)
                {
                    xOro[i, j] = 0;
                }
            }
        }

        private void Can_Loaded(object sender, RoutedEventArgs e)
        {
                        //Dem Buttonarray werden die Buttons der Oberfläche zugewiesen
            but[0, 0] = B00;
            but[0, 1] = B01;
            but[0, 2] = B02;
            but[1, 0] = B10;
            but[1, 1] = B11;
            but[1, 2] = B12;
            but[2, 0] = B20;
            but[2, 1] = B21;
            but[2, 2] = B22;

                        //Wenn der Spieler den Wert "2" zugewiesen bekommt, fängt die "KI" an
            if (spieler == 2)
            {
                KI();
            }

        }
```

Da bei jedem Buttondruck nahezu das gleiche passiert, lohnt es sich hier eine Methode **Change(int executor, int x, int y)** einzuführe n, um den Text am Button in "X "oder "O" zu ändern, zu überprüfen, ob eine der beiden Parteien siegt und einen eventuellen Aufruf der Methode "KI()" auszuführen.

Buttoninhalt

```
                //Buttonmethode des Buttons B00 (an der Stelle 0,0) auch bei den restlichen Button ergänzen,
nur mit den entsprechenden Buttonkoordinaten
                private void B00_Click(object sender, RoutedEventArgs e)
        {
            if (xOro[0, 0] == 0)
            {
                xOro[0, 0] = (byte)spieler;
                Change(xOro[0, 0], 0, 0);
            }
        }
```

und

Change

```
                public void Change(int executor, int x, int y) //Executor: der, der ausführt. (In Zahlen: 1
für Spieler oder 2 für Computer) x und y: Koordinaten des zu ändernden Buttons
        {
                        //Erster Spieler bekommt "X"
            if (executor == 1)
            {
                but[x, y].Content = "X";

            }
            else
            {
                but[x, y].Content = "O";
            }

                        //Liegt ein Sieg vor? Wenn ja, beende
            if (Sieg())
            {
                return;
            }
                        //Hat der Spieler gezogen? Wenn ja, Lass den Computer ziehen
            if (executor == spieler)
            {
                KI();
            }
        }
```

Somit wären die grundlegendsten Dinge abgeschlossen.

Siegesbedingungen

Wann hat man im TicTacToe gewonnen? Wenn man eine Reihe, eine Spalte oder eine Diagonale besitzt. Außerdem gibt es noch das Unentschieden... Das sind ziemlich viele Dinge zum abprüfen, sodass es sich anbietet die einzelnen Bedingungen auszulagern. Die Methode "Sieg()" verwaltet dann die einzelnen Bedingungen und verweist auf die jeweiligen Methoden, welche abfragen. Um Code einzusparen, ist es dabei sinnvoll, zuerst ein enum Orientation einzuführen, da die Methoden zur Abfrage zwischen Reihen und Spalten bzw. zwischen den Diagonalen ziemlich gleich aufgebaut sind:

Enum

```
                //Übersichtlichkeitshalber sollte dies entwerder unmittelbar über die Methode Sieg() (Da
nur von dieser benötigt) oder zu Beginn der Klasse MainWindow eingefügt werden
                public enum Orientation
        {
            Column,
            Row,
            Slash,
            Backslash
        }
```

Sieg

```
                public bool Sieg()
        {
                        //Prüfe, ob eine Spalte gewonnen hat
            if (IsEnd(Orientation.Column))
            {
                            //Beende die Methode
            return true;
            }

                        //Prüfe, ob eine Reihe gewonnen hat
            if (IsEnd(Orientation.Row))
            {
                            //Beende die Methode
            return true;
            }

                        //Prüfe, ob die Diagonale in Richtung eines Backslashs gewonnen hat
            if (IsEndSlashOrBackslash(Orientation.Backslash))
            {
                            //Beende die Methode
            return true;
            }

                        //Prüfe, ob die Diagonale in Richtung eines Slashs gewonnen hat
            if (IsEndSlashOrBackslash(Orientation.Slash))
            {
                            //Beende die Methode
            return true;
            }

                        //Prüfe Unentschieden
            if (IsDraw())
            {
                            //Beende die Methode
            return true;
            }

                        //Kein Sieg liegt vor
            return false;
        }
```

Fangen wir zunächst mit den leichteren Abfragen an: Backlash, Slash und Unentschieden:

SlashOrBackslash

```
            public bool IsEndSlashOrBackslash(Orientation orient)
        {
                        //zwei Bytes werden festgelegt, um zu zählen, wer wie viele Felder in der Diagonale
besitzt
            byte s = 0;
            byte k = 0;
                        //Die Koordinaten der Diagonalen lassen sich (wenn i die Länge einer Dimension sei)
durch [i,i] bzw. durch [i, (LängeEinerDimension-1)-i] beschreiben. Das wird in folgender Schleife abgefragt.
            for (byte i = 0; i < xOro.GetLength(0); ++i)
            {
                                //In den beiden folgenden Abfrage kommt wiederum eine Abfrage vor, ob nach
der Diagonale in Richtung eines Slashs oder in Richtung eines Backslashs abgefragt werden soll.
                                //Dementsprechende Koordinaten werden von der Abfrage zurückgegeben
                if (xOro[i, (orient == Orientation.Backslash ? i : ((xOro.GetLength(0) - 1) - i))] == 1)
                {
                                        //Diese Abfrage zählt die Anzahl der vom Spieler besetzten Felder in
der Diagonale
                    ++s;
                }
                if (xOro[i, (orient == Orientation.Backslash ? i : ((xOro.GetLength(0) - 1) - i))] == 2)
                {
                                        //Diese Abfrage zählt die Anzahl der vom Computer besetzten Felder
in der Diagonale
                    ++k;
                }

                                //Die beiden folgenden Abfragen prüfen, ob einer der beiden Gewonnen hat und
übergeben einer Methode "Spielende" die dafür benötigten Parameter
                if (s == xOro.GetLength(0))
                {
                    Spielende(1);

                                        //Beendet die Methode mit zutreffenden Rückgabewert
                    return true;
                }
                if (k == xOro.GetLength(0))
                {
                    Spielende(2);

                                        //Beendet die Methode mit zutreffenden Rückgabewert
                    return true;
                }
            }

                        //Beendet die Methode mit zutreffenden Rückgabewert
            return false;
        }
```

Unentschieden

```
                public bool IsDraw()
        {
                        //Diese Methode erzeugt einen Byte "u", der hochegzählt wird, wenn das entsprechende
Feld von einer der Parteien besetzt wird
            byte u = 0;
            for (int i = 0; i < xOro.GetLength(0); ++i)
            {
                for (int j = 0; j < xOro.GetLength(1); ++j)
                {
                    if (xOro[i, j] != 0)
                    {
                        ++u;
                    }
                }
            }
                        //Wenn "u" gleich der Anzahl der Felder ist, wird abermals die Methode Spielende mit
entsprechendem Parameter aufgerufen und die Methode mit entsprechendem Rückgabewert beendet
            if (u == xOro.Length)
            {
                Spielende(0);
                return true;
            }

                        //Wenn dies nicht der Fall ist, wird die Methode mit entsprechendem Rückgabewert
beendet
            return false;
        }
```

Die Methoden der Reihen und Spalten sind auch nicht wesentlich komplizierter. Es sind nur zwei Schleifen, verbunden mit der "doppelten" if-Abfrage in der Methode der Diagonalen. Im Prinzip muss man mit einer doppelten Schleife über das Feld laufen, das entweder nach [i,j] und einmal nach [j,i]:

Reihen und Spalten

```
                public bool IsEnd(Orientation orient)
        {
                        //Gleiches Prinzip, wie in der Diagonalenabfrage
            byte s = 0;
            byte k = 0;
            for (byte j = 0; j < xOro.GetLength(1); ++j)
            {
                for (byte i = 0; i < xOro.GetLength(0); ++i)
                {
                                        //erwähnte Abfrage
                    if (xOro[(orient == Orientation.Column ? j : i), (orient == Orientation.Row ? j : i)] ==
1)
                    {
                        ++s;
                    }
                    if (xOro[(orient == Orientation.Column ? j : i), (orient == Orientation.Row ? j : i)] ==
2)
                    {
                        ++k;
                    }
                    if (s == xOro.GetLength(0))
                    {
                        Spielende(1);
                        return true;
                    }
                    if (k == xOro.GetLength(0))
                    {
                        Spielende(2);
                        return true;
                    }
                }
                s = 0;
                k = 0;
            }
            return false;
        }
```

Um die Methode "Sieg()" zu vervollständigen, fehlt nur noch die Methode Spielende(byte gewinner), welche nur abfragen soll, wer denn nun gewonnen hat.

Spieler oder Computer?

"X" oder "O"?

Am Ende dieser Methode kann man nun eine Methode "Restart()" erstellen oder das Programm einfach beenden.

Spielende

```
                public void Spielende(byte gewinner)
        {
                        //In der ersten Abfrage wird abgefragt, welche Nummer der Spieler besitzt
            if (spieler == 1)
            {
                                //in den beiden inneren Abfragen wird ermittelt, wer denn der Gewinner ist
und dementsprechend wird eine Nachricht ausgegeben
                if (gewinner == 1)
                {
                    MessageBox.Show("Spieler (X) gewinnt!");
                }
                else if (gewinner == 2)
                {
                    MessageBox.Show("Computer (O) gewinnt!");
                }
                else
                {
                    MessageBox.Show("Untentschieden");
                }
            }
            else if (spieler == 2)
            {
                if (gewinner == 1)
                {
                    MessageBox.Show("Computer (X) gewinnt!");
                }
                else if (gewinner == 2)
                {
                    MessageBox.Show("Spieler (O) gewinnt!");
                }
                else
                {
                    MessageBox.Show("Untentschieden");
                }
            }
                        //Restart oder Ende
                        //Environment.Exit(0); würde das Programm schließen
            Restart();
        }
```

Optional: Neustart nach Spielende

Zunächst soll die Methode Restart() den Spieler fragen, ob er es denn nochmal versuchen will. Dafür gibt es bereits ein vorgefertigtes Fenster, das mit nur wenigen Zeilen abgerufen werden kann.

Restart

```
                public void Restart()
        {
                        //Eventuell ist für System.Windows.Forms ein Assemblyverweis nötig. Dazu später mehr.
            System.Windows.Forms.DialogResult result = System.Windows.Forms.MessageBox.Show("Retry?",
"Confirmation", System.Windows.Forms.MessageBoxButtons.YesNo);
            if (result == System.Windows.Forms.DialogResult.Yes)
            {
                                //Die Methode "Reset()" setzt alle Variablen zurück auf Anfang.
                                //Diese Methode ist noch nicht ganz vollständig, da wir später in der "KI()"-
Methode evtl. weitere Variablen brauchen, die wir hier zurücksetzen müssen.
                                //Ich werde sie an entsprechenden Stellen markieren
                Reset();
            }
            else if (result == System.Windows.Forms.DialogResult.No)
            {
                Environment.Exit(0);
            }
        }
```

Einen Assemblyverweis fügt man unter **"Projekt Verweis Hinzufügen"** hinzu. Dann nur in der Linken Spalte unter **"Assemblys Framework"** System.Windows.Forms suchen und hinzufügen.

Reset

```
                public void Reset()
        {
                        //Spieler wird neu gesetzt
            spieler = zgen.Next(2) + 1;

                        //ki wird neu gesetzt
            if (spieler == 1)
            {
                ki = 2;
            }
            else
            {
                ki = 1;
            }

                        //Buttontexte und Werte in xOro werden neu gesetzt
            for (int i = 0; i < xOro.GetLength(0); ++i)
            {
                for (int j = 0; j < xOro.GetLength(1); ++j)
                {
                    xOro[i, j] = 0;
                    but[i, j].Content = "";
                }
            }

                        //evtl. fängt der Computer an
            if (spieler == 2)
            {
                KI();
            }
        }
```

Die Logik (Überprüfen von Reihen)

Der mit Abstand Schwierigste Teil kommt jetzt. Da es 255.168 Möglichkeiten gibt, dieses Spiel zu spielen, ist klar, dass wir abstrahieren müssen, um einen Algorithmus zu entwickeln. Eine Abstraktion in Reihen würde sich anbieten, um den Spielspaß nicht aus dem Auge zu verlieren. Wenn man Lust hat, kann man auch einbauen, sodass, die Strategie, in der man, wenn man beginnt, einfach in eine Ecke setzt und (gegen einen Computer, der auf Reihen acht gibt und sonst Random setzt) meistens gewinnt in ein (fast) sicheres unentschieden für den Computer verwandeln.

Zusammengefasst wird der Computer drei Dinge in folgender Reihenfolge prüfen:

- Kann ich gewinnen?
- Kann der Spieler gewinnen?
- Hat der Spieler in die Ecke gesetzt? (in der ersten und zweiten Runde) (Optional)

Desweiteren können wir dem Computer lernen, auch mal etwas zu "übersehen", um ihn menschlicher zu machen. Die Wahrscheinlichkeit dieser "Fehler" des Computers kann jeder in bestimmtem Maße selbst bestimmen. Bei mir liegt sie bei etwa 5%. Die Methode "KI()" die bereits des Öfteren mal vorkam, soll dies verwalten und dem Computer sagen, was wann zu tun ist.

"KI"

```
        public void KI()
        {
                        //Wenn eine zufällige Zahl zwischen 0 und 19 größer als 18 ist (sprich 19), dann
wird zufällig gesetzt, ansonsten soll der richtige Algorithmus aufgerufen werden.
                        // Dieser prüft, ob jemand gewinnen kann, entscheidet dementsprechend und (optional)
kontrolliert auch die Ecken
            if (zgen.Next(20) > 18)
            {
                KI_Random();
            }
            else
            {
                CheckRows();
            }
        }
```

Da "KI_Random()" die bei Weitem einfachere Methode ist, werde ich mit dieser anfangen.

KI_Random

```
                public void KI_Random()
        {
                        //Zwei Zufallszahlen werden gezogen. Anschließend wird überprüft, ob an den
Koordinaten der beiden Zahlen frei ist und wenn ja, wird an entsprechender Position gesetzt.
                        //Wenn nicht, wird die Methode erneut aufgerufen, so lange, bis an der zufälligen
Position frei ist und gesetzt werden kann.
            int i = zgen.Next(3), j = zgen.Next(3);

            if (xOro[i, j] == 0)
            {
                xOro[i, j] = (byte)ki;
                Change(ki, i, j);
            }
            else
            {
                KI_Random();
            }
        }
```

Aber bevor wir zu "CheckRows()" kommen, müssen wir eine Klasse Rows einfügen, um die Reihen zu speichern. Diese Werte können in diesen Reihen so verarbeitet werden, dass wir die in unserem "CheckRows()"- Algorithmus besser nutzen können. Sie soll drei Integer- ARRAYs verwalten, die jeweils drei Zahlen speichert, sodass es insgesammt neun Zahlen sind. Dies ist notwendig, um einerseits die den Wert von "xOro" an den Koordinaten x und y zu speichern, ohne die Koordinaten zu verlieren. Eines der drei Arrays hat somit den Aufbau: {xOro[x,y], x, y}. Dies wird drei mal gemacht, für jede Koordinate in der Reihe im TicTacToe-Feld einmal.

Rows

```
         public class Rows
    {
                //Die drei erwähnten Arrays der Form: {xOro[x,y], x, y} für jede Koordinate der Reihe
        int[] x1, x2, x3;

                //Diese Felder werden bereits im Konstruktor übergeben
        public Rows(int[] x, int[] y, int[] z)
        {
            x1 = x;
            x2 = y;
            x3 = z;
        }
        }
```

Im 3x3 TicTacToe gibt es 8 Reihen, drei vertikale, drei horizontale und beide Diagonalen. Somit instanzieren wir in der Methode "CheckRows()" sogleich ein Rows-Array, sowie dessen Inhalt.

(1) CheckRows

```
                 //gehört wieder in die Klasse "Main Window"
                public void CheckRows()
        {
                        //Das Array, sowie dessen Elemente werden instanziert
            Rows[] ro = new Rows[8];
                        //Es werden jeweils 3 neue Arrays, sowie deren Werte erzeugt.
            ro[0] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[0, 1], 0, 1 }, new int[3] {
xOro[0, 2], 0, 2 });
            ro[1] = new Rows(new int[3] { xOro[1, 0], 1, 0 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[1, 2], 1, 2 });
            ro[2] = new Rows(new int[3] { xOro[2, 0], 2, 0 }, new int[3] { xOro[2, 1], 2, 1 }, new int[3] {
xOro[2, 2], 2, 2 });
            ro[3] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 2], 2, 2 });
            ro[4] = new Rows(new int[3] { xOro[0, 2], 0, 2 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 0], 2, 0 });
            ro[5] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[1, 0], 1, 0 }, new int[3] {
xOro[2, 0], 2, 0 });
            ro[6] = new Rows(new int[3] { xOro[0, 1], 0, 1 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 1], 2, 1 });
            ro[7] = new Rows(new int[3] { xOro[0, 2], 0, 2 }, new int[3] { xOro[1, 2], 1, 2 }, new int[3] {
xOro[2, 2], 2, 2 });

                        //Da kommt noch etwas
                }
```

Außerdem braucht die Klasse Rows wir eine Methode, die uns die Daten, die wir brauchen in geeigneter Form zurückgibt. Das Array, das uns übergeben wird, hat den Aufbau

{AnzahlDesÜbergebenenWertes(spieler)InEinerxOro-Reihe/-Spalte/-Diagonale, x-KoordinateDesLetztenFreienFeldes, y-KoordinateDesLetztenFreienFeldes}

In diesem Feld steht alles, was wir brauchen, um unsere Abfragen zu machen.

GetData

```
                //in Klasse "Rows" schreiben
                public int[] GetData(int spieler)
        {
                        //Feld, das später zurückgegeben werden soll
            int[] ret = new int[3];

                        //Erster Wert des Feldes wird als Zählvariable genutzt, um die Anzahl des
übergebenen Parameters in der Reihe zu bestimmen.
            ret[0] = 0;

            if (x1[0] == spieler)
            {
                ++ret[0];
            }
            else
            {
                                //Da uns nur Reihen interessieren, welche 2-Mal den übergebenen Parameter
beinhalten, werden die [1] und [2]-Werte gleich der dritten "vielleicht freien" Koordinate gesetzt.
                ret[1] = x1[1];
                ret[2] = x1[2];
            }
            if (x2[0] == spieler)
            {
                ++ret[0];
            }
            else
            {
                ret[1] = x2[1];
                ret[2] = x2[2];
            }
            if (x3[0] == spieler)
            {
                ++ret[0];
            }
            else
            {
                ret[1] = x3[1];
                ret[2] = x3[2];
            }

            return ret;
        }
```

Da der Code der abfrägt, ob der Computer oder der Spieler im nächsten Zug gewinnen können und den Computer zu entsprechenden Handlungen bringt sehr ähnlich sind, bietet es sich hier an, sie auszulagern. In dieser Methode geht es schließlich darum, in einer Reihe, in welcher bereits zwei Felder besetzt sind, das dritte zu besetzen. Besser gesagt unterscheiden sie sich in genau einem Parameter.

PruefeGewinnOderVerlustKI

```
        //ab jetzt wieder alles in Klasse "Main Window"
                public bool PruefeGewinnOderVerlustKI(Rows[] ro, int wer) //Das Rows-Array, das in CheckRows
bereits instanziert wurde wird übergeben
        {
                        //Für alle Werte im Rows-Feld wird folgender Code ausgeführt
            for (int i = 0; i < ro.Length; ++i)
            {
                                //Das zurückgegebene Feld der GetData-Methode des Rows-Feldes wird
gespeichert
                int[] rf = ro[i].GetData(wer);

                                //Sind von der Partei "wer" zwei Felder in der Reihe besetzt und ist das
dritte Feld frei?
                if (rf[0] == 2 && xOro[rf[1], rf[2]] == 0)
                {
                                        //Wenn Ja: besetze dieses Feld und beende die Methode mit true "Ja,
Ich habe gesetzt"
                    xOro[rf[1], rf[2]] = (byte)ki;
                    Change(ki, rf[1], rf[2]);
                    return true;
                }
            }

                        //Ansonsten beende die Methode mit false "Nein, ich habe nicht gesetzt"
            return false;
        }
```

daraus ergibt sich für die CheckRows-Methode:

(2) CheckRows

```
        public void CheckRows()
        {
            Rows[] ro = new Rows[8];
            ro[0] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[0, 1], 0, 1 }, new int[3] {
xOro[0, 2], 0, 2 });
            ro[1] = new Rows(new int[3] { xOro[1, 0], 1, 0 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[1, 2], 1, 2 });
            ro[2] = new Rows(new int[3] { xOro[2, 0], 2, 0 }, new int[3] { xOro[2, 1], 2, 1 }, new int[3] {
xOro[2, 2], 2, 2 });
            ro[3] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 2], 2, 2 });
            ro[4] = new Rows(new int[3] { xOro[0, 2], 0, 2 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 0], 2, 0 });
            ro[5] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[1, 0], 1, 0 }, new int[3] {
xOro[2, 0], 2, 0 });
            ro[6] = new Rows(new int[3] { xOro[0, 1], 0, 1 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 1], 2, 1 });
            ro[7] = new Rows(new int[3] { xOro[0, 2], 0, 2 }, new int[3] { xOro[1, 2], 1, 2 }, new int[3] {
xOro[2, 2], 2, 2 });


            //Prüfe Gewinn der KI
            if (PruefeGewinnOderVerlustKI(ro, ki))
            {
                            //beende die Methode, wenn gesetzt wurde, um doppeltes Setzen zu vermeiden
                return;
            }
            //Prüfe Verlust der KI
            if (PruefeGewinnOderVerlustKI(ro, spieler))
            {
                            //beende die Methode, wenn gesetzt wurde, um doppeltes Setzen zu vermeiden
                return;
            }

                    //Wenn keiner der beiden Fälle eintritt: setze Random
                    KI_Random();

                }
```

Damit sollte das Programm bereits spielbar sein. Ich werde im Folgenden noch schildern, wie man den Trick mit der Ecke umgeht.

Die Logik (Überprüfen der Ecken) (optional)

Im Prinzip ist für diesen Teil nur die erste und zweite Runde relevant. Um den "Eckentrick" zu umgehen, muss man, wenn der Spieler in die Ecke gesetzt hat, in die Mitte setzen. Wenn der Spieler anschließend in der gegenüberliegenden Ecke setzt, setzt man irgendwo am Rand. Den Rest übernimmt unser Reihenalgorithmus. Somit brauchen wir zwei bool r1 = true, r2 = false; , die zu Beginn der Klasse MainWindow instanziert werden, welche uns sagen, ob denn die erste Runde ist, oder nicht.

(**Achtung!**: Wenn der Computer anfängt, soll r1 wieder false werden! Dies ist zu ändern, wenn in **Can_Loaded(...)** und **Reset()** zufällig entschieden wird, ob die Methode "KI()" ausgeführt wird. Bedenkt auch, r1 und r2 in der Methode Reset() wieder deren Ursprungswerten zuzuweisen.)

Als nächstes brauchen wir eine Methode "EckenTrick()", welche abfragen soll, ob die erste Runde ist und ob der Spieler in eine Ecke gesetzt hat. Und danach ob die zweite Runde ist.

EckenTrick

```
        public bool EckenTrick()
        {
                        //ist die erste Runde UND hat der Spieler in eine Ecke gesetzt
            if (r1 && (xOro[0, 0] == spieler || xOro[0, 2] == spieler || xOro[2, 0] == spieler || xOro[2, 2]
== spieler))
            {
                                //Setze in die Mitte
                xOro[1, 1] = (byte)ki;
                Change(ki, 1, 1);
                                //Runde 1 vorbei, Runde 2 beginnt!
                r1 = false;
                r2 = true;
                return true;
            }

                        //ist zweite Runde?
            if (r2)
            {
                                //Setze Random an einen Rand
                RandomRand();
                return true;
            }
                        return false;
                }
```

Die Methode "RandomRand()" soll eine zufällige Zahl zwischen 0 und 3 wählen und die Koordinaten des Randstücks einer Methode "SetzeRandomRand(int i, int j)" übergeben werden, die anschließend überprüft, ob das Feld Frei ist. Wenn ja, soll dort gesetzt werden. Wenn nein, soll wieder die Methode "RandomRand()" aufgerufen werden, um neue Koordinaten zu ermitteln. Dies stellt quasi eine Rekurrsion über zwei Methoden dar.

RandomRand und SetzeRandomRand

```
 public void RandomRand()
 {
                //Ermittle eine Zufallszahl für einen zufälligen Rand
    switch (zgen.Next(4))
    {
                        //Übermittle SetzeRand(...) einen RandomRand
            case 0:
                SetzeRandomRand(0, 1);
                break;

            case 1:
                SetzeRandomRand(2, 1);
                break;

            case 2:
                SetzeRandomRand(1, 0);
                break;

            case 3:
                SetzeRandomRand(1, 2);
                break;
                }
}




public void SetzeRandomRand(int i, int j)
{
                //Überprüfung, ob das Randfeld frei ist
    if (xOro[i, j] == 0)
    {
                        //Wenn ja: setze auf das Randfeld
        xOro[i, j] = (byte)ki;
        Change(ki, i, j);

                        //Runde 2 vorbei
        r2 = false;
    }
    else
    {
                        //Wenn nein: ermittle neue Randkoordinaten
        RandomRand();
    }
}
```

Das wäre eigentlich schon die Logik dazu. Jetzt müssen wir dann ganze nur noch den Algorithmus in "CheckRows()" einfügen:

(3) CheckRows

```
        public void CheckRows()
        {
            Rows[] ro = new Rows[8];
            ro[0] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[0, 1], 0, 1 }, new int[3] {
xOro[0, 2], 0, 2 });
            ro[1] = new Rows(new int[3] { xOro[1, 0], 1, 0 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[1, 2], 1, 2 });
            ro[2] = new Rows(new int[3] { xOro[2, 0], 2, 0 }, new int[3] { xOro[2, 1], 2, 1 }, new int[3] {
xOro[2, 2], 2, 2 });
            ro[3] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 2], 2, 2 });
            ro[4] = new Rows(new int[3] { xOro[0, 2], 0, 2 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 0], 2, 0 });
            ro[5] = new Rows(new int[3] { xOro[0, 0], 0, 0 }, new int[3] { xOro[1, 0], 1, 0 }, new int[3] {
xOro[2, 0], 2, 0 });
            ro[6] = new Rows(new int[3] { xOro[0, 1], 0, 1 }, new int[3] { xOro[1, 1], 1, 1 }, new int[3] {
xOro[2, 1], 2, 1 });
            ro[7] = new Rows(new int[3] { xOro[0, 2], 0, 2 }, new int[3] { xOro[1, 2], 1, 2 }, new int[3] {
xOro[2, 2], 2, 2 });


            //Prüfe Gewinn der KI
            if (PruefeGewinnOderVerlustKI(ro, ki))
            {
                return;
            }
            //Prüfe Verlust der KI
            if (PruefeGewinnOderVerlustKI(ro, spieler))
            {
                return;
            }

            if (EckenTrick())
            {
                return;
            }
                        //Runde 1 vorbei
            r1 = false;
            KI_Random();
        }
```

Fertig!

Natürlich kann das Programm noch verfeinert werden. Einige Ideen dazu:

- Ein optimierter (schwierigerer) Algorithmus
- Schwierigkeitsgrade
- Ein n*n-TicTacToe, wo der Spieler selbst die Seitenlänge bestimmt
- Ein n^k-TicTacToe mit mehr, als zwei Dimensionen (Darstellung wird für k>3 schwierig)
- Eine "WinStreak" Funktion, die die Spiege in Folge speichert
    - evtl. mit Highscoresystem