

# **C# basiertes multiplayer online Strategiespiel**

Nicolai Schiele  
2012 / 2013

# C# basiertes multiplayer online Strategiespiel



<b>vorgelegt von:</b>	Nicolai Schiele
<b>Klasse:</b>	2BKI/2
<b>Kontaktadresse:</b>	Hauffstraße 29 89077 Ulm nicolai_schiele@web.de
<b>Abgabedatum:</b>	03.05.2013
<b>im Schuljahr:</b>	2012 / 2013
<b>Betreuung:</b>	Herr Michael Mattes

# **Inhaltsverzeichnis**

## **1 Einführung**

## **2 Englische Fassung der Einführung**

## **3 Hauptteil**

### **3.1 Client**

#### **3.1.1 Menü**

#### **3.1.2 Spieloberfläche**

#### **3.1.3 Spielfeld.cs**

#### **3.1.4 Verbindung des Clients**

### **3.2 Server**

#### **3.2.1 Spielfigur.cs**

#### **3.2.2 Spieler.cs**

#### **3.2.3 Client.cs**

#### **3.2.4 Verbindung des Servers**

### **3.3 Abläufe**

#### **3.3.1 Ablauf bis zum Spiel**

#### **3.3.2 Ablauf eines Spiels**

## **4 Fazit**

## **5 Verzeichnisse**

### **5.1 Literaturverzeichnis**

### **5.2 Abbildungsverzeichnis**

### **5.3 Abkürzungsverzeichnis**

## **6 Bestätigung der Selbstarbeit**

## **7 Anhang**

# 1 Einführung

Ich habe mich in den letzten 2 Jahren am 2BKI auch abseits der im Unterricht gestellten Programmieraufgaben immer wieder für das Programmieren von Spielen und Spielmechaniken interessiert und da ich auch vor habe nach dem Berufskolleg in der Medienbranche zu bleiben, lag es für mich nahe, als Projektarbeit ein Spiel zu programmieren, welches meine bisherigen Spiele in ihrer Komplexität aber bei weitem überschreiten sollten. Das Konzept meines Projekts ist es zwei in C# geschriebene Programme mit einer TCP<sup>1</sup> Verbindung in ein Server-Client Verhältnis zu bringen, um damit ein netzwerkfähiges, grafikbasiertes, zwei-spieler Strategiespiel zu entwickeln, welches beide Spieler rundenbasiert gegeneinander antreten lässt.

Jeder Spieler wählt zu Beginn der Partie eine Klasse aus. Es gibt drei Klassen die sich in Angriffs- und Verteidigungsstärke der Figuren unterscheiden. Je nachdem welche Klasse der Spieler wählt, erhält er eine bestimmte Menge Geld und Bewegungspunkte pro Runde. Beides kann er nutzen um Figuren zu erstellen, zu bewegen, gegnerische Figuren anzugreifen oder die eigenen upzugraden. Zusätzlich gibt es drei verschiedene Figurenarten, welche sich wiederum in Angriffs- und Verteidigungsstärke, Lebenspunkten und Bewegungs- und Angriffsareal unterscheiden. Da es sich um ein rundenbasiertes Spiel handelt kann immer nur ein Spieler Züge ausführen, was ein sehr strategisches Vorgehen erfordert. Ziel des Spiels ist es, sämtliche Figuren des Gegners zu zerstören bis dieser keine Ressourcen mehr hat, um neue zu erstellen und somit nicht weiterspielen kann. Das Ganze passiert auf einer Karte mit insgesamt 20x20 Feldern. Neben den normalen Zug-Feldern existieren auch „Spawn“-Felder zum Erstellen der Figuren und Hindernis-Felder, auf die die Figuren nicht ziehen können.

Das erste Programm stellt den Client dar. Dieser beinhaltet die Informationen zur Bedienung und Darstellung des Spiels, ein Auswahlménü, Spieltöne und ein einfaches Tutorial<sup>3</sup>, welches rein bildlich ist. Die Bedienung funktioniert komplett mausgesteuert und soll ein möglichst intuitives Spielgefühl bieten. Zur Darstellung gehören neben dem eigentlichen Spielfeld allgemeine Informationen, sowie die Werte der einzelnen Spielfiguren. Beide Spieler verwenden das selbe Programm, um sich am Server anzumelden. Einer der beiden kann den Server vom Programm aus auf seinem PC starten, sodass sich der andere mit der IP-Adresse des Server-PCs an diesem anmelden kann. Es ist aber auch möglich, den Server auf einem dritten PC zu starten, um diesen als zentralen Knotenpunkt zu nutzen.

Der Server beinhaltet die gesamte Spielmechanik und verwaltet das Spiel. In ihm befinden sich alle Informationen über die Spieler, das Spielfeld und die einzelnen Figuren. Jede Handlung, die ein Spieler macht wird vom Client an den Server geschickt, dort verarbeitet und zurück gesendet. Dies passiert ausschließlich über Anfragen, die der Server an die Clients schickt, was bedeutet, dass ein Client nur dann senden kann, wenn der Server eine Anfrage dafür gesendet hat. Benutzer können nicht auf den Server zugreifen, da dieser eine reine Konsolenanwendung ohne Eingabemöglichkeit ist.

Das Projekt befindet sich zur Abgabe in einem unfertigen Zustand, da ich die zeitlichen Vorgaben von 80 Arbeitsstunden bei weitem überschreiten musste und mir nach ca. 150 Stunden die Zeit bis zum Abgabetermin ausging.

---

1 Transmission Control Protocol

2 Englisch: hervorbringen. Hier: Das erzeugen neuer Figuren

3 Schriftliche oder bildliche Lernanleitung

## 2 Englische Fassung der Einführung

In the past two years in the 2BKI, i have often worked on small game programmes aside from the normal programming lessons and because I am planning on staying in the media after school I instantly thought of doing a game for my project, but this time i wanted it to be far more complex than my previous games. The concept off the game is to have two C# written programmes connected with a TCP connection and build a network-compatible, graphical-based, two-player strategy game on that, which let both players fight a turn-based online match.

At the beginning off each round, both players choose one off three classes, which differ in attack and defense strength off the individual figures. Depending on which class a player chooses, he gets a different amount of money and moving points. Both can be used to attack the opponent or to spawn, move or upgrade own figures. There are three types of figures with different attack and defense strenght, healthpoints and movment and attack areas. Because its a turn-based game, there is always only one off both playing, what requires a very strategic approach. The goal off this game is it, to destroy all of the enemies units until he has no money left to spawn new ones and cannot go on playing. All off that happens on a 20 by 20 field card, which includes move, obstacle and spawn fields to create new figures.

The first programm represents the client. It contains all informations to to handle and display the game, a menu, playing sounds and a simple tutorial, which comes as an image. The handling of the game is completly mouse-controlled and shall offer an intuitive gameplay. Besides the actual card there are general informations like figure stats, remaining money and moving points. Both players use the same programm to connect to the server whereat one of both can start the server directly from the client, so that the other player can connect to it with the IP adress of the servers PC. It is also possible to start the server on a third PC in order to use it just as a server and play the game from two different PCs on it.

The Server contains all the game mechanics and manages the whole process. It holds information about both playes, the playing field and the individual figures. Every action a player takes will be sent from the client to the server, processed and sent back. All of the traffic between clients and server works with requests which means that a client can only send information to the server when it requested one.

At the time of the deadline the project is in an uncomplete condition, because I had to exceeded the permitted time of 80 workhours by far and reached the deadline after about 150 hours of working on it.

## 3 Hauptteil

Im folgenden erläutere ich die Abläufe meines Programms. Da das Projekt noch nicht vollständig ist werde ich es hier so beschreiben und schildern wie ich es geplant habe.

### 3.1 Client

Der Client ist ein recht einfach strukturiertes Programm, da er lediglich vier Klassen beinhaltet. Es handelt sich hierbei um eine Windows Forms Anwendung, weswegen sich der Großteil der Programmabläufe in der Form1.cs befindet und mit Events abgerufen werden kann.

#### 3.1.1 Menü

Zu Beginn des Programms befindet man sich in einem Auswahlmenü(Abb.1), welches den Benutzer mit einer vererbten Klasse der TabControl, die über Buttons angesprochen wird, durch die einzelnen Menüpunkte führt. Um nicht für jeden hier verwendeten Button einen einzelnen Ereignishandler mit sehr wenig Code verwenden zu müssen, geschieht die gesamte Menüführung über einen einzigen Ereignishandler, den jeder Button mit seinem „Click“ Ereignis aufruft. Je nach gedrücktem Button entscheidet der Ereignishandler *Menü\_btn\_Click* welcher Tab angezeigt werden soll. Im Optionsbereich kann der Spieler über eine CheckBox Spieltöne an- und ausschalten. Dies ist jederzeit möglich, da das Programm bei jedem Aufruf der Methode *PlaySound* überprüft, ob die CheckBox noch aktiviert ist. Das in der Einleitung erwähnte Tutorial ist sehr simpel, da es nur ein, in eine PictureBox eingefügtes Bild ist, welches die im Spiel relevanten Anzeigen und Buttons auf einfachste Art und Weise erklären soll(Abb. 2). Auf der Menüseite „Spiel erstellen“ muss der Spieler Klasse und Spielfeld wählen, mit denen er spielen will und kann anschließend das Spiel starten. Nun startet sich das Programm SingleThreadedServer. Der Client verbindet sich mit der Localhost IP Adresse „127.0.0.1“ mit ihm und wartet darauf, dass sich der zweite Client am Server anmeldet. Hierfür gibt es einen einzelnen Eventhandler, welcher die Methode *Connect* aufruft. Der zweite Client kann sich nun im Menüpunkt „Spiel beitreten“ an diesem Server anmelden und verwendet dafür ebenfalls nur einen Eventhandler, welcher die *Connect* Methode aufruft, für die Verbindung aber natürlich die IP Adresse des Servers benötigt.

Es sei noch erwähnt, dass ich die verwendete Klasse *TablessControl*, welche die Reiter der Tabs versteckt, nicht selbst geschrieben habe. Diese ist so wie sie vorliegt aus einem Forenbeitrag entnommen<sup>4</sup>.

---

<sup>4</sup> Passant, 2011 [online]

### 3.1.2 Spieloberfläche

Nachdem sich beide Spieler am Server angemeldet haben wechselt die *TablessControl* auf den Reiter „Spiel“ und beide Spieler sehen nun die Spieloberfläche (Abb. 3). Der Großteil dieser ist die Karte, welche, je nachdem welches Feld gewählt wurde, unterschiedlich aussehen kann. Ein Klick auf das Feld führt den *pb\_SpielFeld\_MouseClick* Ereignishandler aus, welcher lediglich zwei andere Methoden ausführt. Er übergibt den Rückgabewert der Methode *Feldauswahl* an die *Senden* Methode. Erstere nimmt die aktuelle Mauskoordinaten abzüglich der Position des Programms auf dem Monitor und teilt diese in einen Wert zwischen 0 und 19, womit sich die Koordinaten der einzelnen Felder der Karte beschreiben lassen. Diese Koordinaten werden dann mithilfe der *Senden* Methode an den Server gesendet, welcher dem Client nun mitteilt, um was für ein Feld es sich handelt. Klickt ein Client zum Beispiel auf ein „Spawn“-Feld, erhält er vom Server diese Information und öffnet darauf hin das „Spawn-Menü“. Dieses besteht aus 3 Labels mit jeweils einem Button. Damit lässt sich nun auswählen, welche Art von Figur man erstellen will. Alle 3 Buttons greifen hierbei wieder auf den selben Eventhandler *btn\_Figur\_erstellen* zurück, welcher nun als erstes prüft, ob der Client noch genug Geld und Bewegungspunkte für das Erzeugen der Figur zur Verfügung hat. Wenn ja, zieht er den Geldbetrag von seinem Konto, sowie 2 Bewegungspunkte ab und ruft anschließend die, *Figurzeichnen* Methode auf, welche die für die ausgewählte Figur passende .bmp<sup>5</sup> Datei auf das gewählte Feld zeichnet, die *PlaySound* Methode mit dem „Spawn“ Ton aufruft und den Wert des Feldes ändert, da es jetzt um ein Figuren-Feld handelt. Das Zeichnen passiert im gesamten Spiel immer über die *DrawImage* Methode der Klasse *Graphics*, die eine ausgewählte .bmp Datei an einen bestimmten Punkt auf der Picture Box zeichnet. Bei einem Klick auf eine Figur öffnet sich die zweite Anzeige, welche alle Attribute einer Figur anzeigt. Sie wird über die Methode *ElementeAnzeigen* aufgerufen, die lediglich die Sichtbarkeit aller dafür vorgesehenen Labels auf true setzt. Das besondere hierbei ist, das hierfür ein Delegate<sup>6</sup> notwendig ist, da das Empfangen des Befehls und die Einstellungen für die Labels in zwei unterschiedlichen Threads geschieht. Innerhalb der Anzeige ist es möglich, die gerade gewählte Figur upzugraden, was mit einem Klick auf den „LevelUp“ Button passiert. Sein Eventhandler sendet dem Server nun die Information, die *LevelUp* Methode für diese Figur auszuführen. Unter den Anzeigen befindet sich der Button „Runde beenden“. Dieser sendet dem Server den Befehl, dass dieser nun den anderen Client ansprechen soll.

### 3.1.3 Spielfeld.cs

Die Klasse *Spielfeld* ist sowohl im Client, als auch im Server enthalten. In ihr werden die Daten, die jeder Karte hinterlegt sind, in ein Array geschrieben. Jedes Feld des Arrays bekommt einen Zahlenwert, mit dem entschieden wird, welche Handlung ausgeführt werden muss, wenn ein Feld angeklickt wurde. Beim Erstellen einer Figur bekommt das Feld, auf dem dieses geschieht einen neuen Zahlenwert, welcher nach Bewegung der Figur zurück gesetzt wird.

---

<sup>5</sup> Bitmap

<sup>6</sup> vgl. Delegates (Jahr unbekannt) [online]

### 3.1.4 Verbindung des Clients

Das Verbinden zum Server geschieht in der *Connect* Methode. In ihr verbindet sich ein Objekt der Klasse *TcpClient* mit einem angegebenen Endpunkt. Dieser wiederum besteht aus einer IP Adresse und einem Port. Anschließend wird jeweils ein Objekt der Klasse *Streamreader* und *Streamwriter* erzeugt, welche beide den Stream des *TcpClients* verwenden, um ihre Informationen zu senden. All das muss in einem *try/catch* Block liegen, da es durchaus möglich ist, dass der Versuch, sich an einem Server anzumelden, nicht funktioniert und eine *SocketException* ausgelöst wird. Nachdem der Verbindungsprozess abgeschlossen ist, erstellt die *Connect* Methode noch einen neuen Thread, der für das Empfangen von Daten zuständig ist und startet ihn<sup>7</sup>. Das ist sinnvoll, da der Server so, unabhängig von dem was ein Spieler gerade macht, Daten senden kann, die dann auch sofort verwertet werden, ohne dass der Client noch eine Handlung beenden muss<sup>8</sup>.

---

<sup>7</sup> vgl. Amjad (2001) [online]

<sup>8</sup> vgl. *TcpClient* (Jahr unbekannt) [online]



## 3.2 Server

Der Server wird im normalen Spielmodus, bei dem einer der beiden Spieler ein neues Spiel erstellt, vom Client aus geöffnet und verbindet sich anschließend direkt mit diesem. Nun wartet er, bis sich ein zweiter Client anmeldet. Er stellt das vom ersten Client ausgewählte Feld ein und überprüft ob der zweite Client die selbe Klasse wie der Erste ausgewählt hat. Wenn ja sendet er ihm den Befehl, eine neue Klasse zu wählen. Nachdem beide Clients unterschiedliche Klassen haben, sendet er den Spielern den Befehl, das Spiel zu starten. Innerhalb des Spiels sendet er den Clients Sendeabfragen, auf die die Clients antworten, was er daraufhin verarbeitet. Das komplette Programm läuft in nur einem Thread, was auf der einen Seite eine klare Struktur bringt, auf der anderen Seite aber eine Abfrage für jede Handlung, die ein Client machen will, erfordert. Der Server ist zwar nicht für die Ausgabe da, schreibt aber dennoch jede Handlung und Information in die Konsole (Abb. 4), was dem Ersteller des Spiels nach dem Spiel Rückverfolgen lässt, wie der Ablauf des Spiels war.

### 3.2.1 Spielfigur.cs

Es gibt 3 verschiedene Klassen, von der es jeweils 3 verschiedene Figuren gibt. Wird ein Objekt der Klasse Spielfigur erzeugt, werden im Konstruktor die Koordinaten der Figur eingestellt, sowie die Art der Figur. Danach wird ermittelt, um was für eine Figur es sich handelt und die Methode *WerteSetzen* erzeugt je nach Art und Klasse eine Figur mit bestimmten Attributen und Bewegungs- und Angriffsarealen. Die Areale sind kleine 7x7 Arrays, in denen die Werte für den Bewegungs- und Angriffsraum sind. Diese Werte werden bei einem Klick auf eine Figur um die Figur in das Spielfeld Array geschrieben, um dem Spieler die Areale anzeigen zu können. Klickt ein Spieler nun auf ein Feld das im Bewegungsareal ist wird die *FigurBewegen* Methode aufgerufen, die der Figur ihre neuen Koordinaten übergeben. Klickt ein Spieler auf ein Angriffsfeld, wird die *GreiftAn* Methode aufgerufen, welche der übergebenen Figur nach einem bestimmten Angriffsalgorithmus Gesundheitspunkte abzieht. Die *LevelUp* Methode multipliziert Angriffs- und Verteidigungsstärke mit dem neuen Level und die überschriebene *ToString* Methode liefert alle wichtigen Werte mit Kommata getrennt zurück, um diesen String später leicht trennen und Labels zuweisen zu können.

### 3.2.2 Spieler.cs

Jedes Objekt der Klasse Spieler enthält eine Liste von Spielfiguren, die bei Aufruf der *NeueFigur* Methode um ein Objekt erweitert wird. Außerdem gibt die Methode die *ToString* Methode an der Konsole des Servers aus, zieht dem Objekt Spieler den Geldwert ab, und liefert die Spielfigur zurück.

### 3.2.2 Client.cs

Jeder Client enthält ein Objekt der Klasse Spieler. Dadurch ist es möglich einzelne Spielfiguren eines Spielers über den Client anzusprechen. In der Klasse Client wird außerdem, genau wie im Programm Client, ein TcpClient, ein Streamreader und ein StreamWriter erzeugt, die die Informationen jedes einzelnen Spieler senden und Empfangen können.

### 3.2.4 Verbindung des Servers

Im Server wird zu Beginn ein Objekt der Klasse `TcpListener` erzeugt, welches die Verbindungen der einzelnen `TcpClients` überwacht und Verbindungsanforderungen jeder IP<sup>9</sup> Adresse erlaubt. Dem `TcpClient` der Klasse `Client` wird daraufhin eine ausstehende Verbindungsanforderung des sich anmeldenden Clients übergeben. Es gibt genau wie im Client-Programm einen `StreamReader` und einen `StreamWriter`, die genau die selben Aufgaben übernehmen. Da es sich sowohl im Server, als auch im Client um eine Simple TCP Verbindung handelt, ist das Spiel ausschließlich in LAN Netzwerken verwendbar.

---

<sup>9</sup> Internetprotokoll

### 3.3 Abläufe

Im folgenden werde ich die doch recht komplexen Abläufe und Sendeeinformationen der beiden Programme an Hand von Codebeispielen erläutern. Zum Verständnis sind alle Codezeilen des Servers blau und alle des Clients gelb hinterlegt.

#### 3.3.1 Ablauf bis zum Spiel

```
clients[0].Spieler.Partei = SendeBefehlLiesAntwort(clients[0], "klas");
```

Sendet dem ersten Client das Schlüsselwort „klas“ und speichert seine Klasse.

```
case "klas":
if (empfangen.Substring(4) != "")
{
    MessageBox.Show("Spieler 1 hat diese Klasse bereits gewählt. Bitte neu Wählen");
    switch (empfangen.Substring(4))
    {
        case "red":
            btn_Red_Bei.Enabled = false;
            btn_Red_Bei.BackColor = Color.Gray;
            break;
        case "green":
            btn_Green_Bei.Enabled = false;
            btn_Green_Bei.BackColor = Color.Gray;
            break;
        case "blue":
            btn_Blue_Bei.Enabled = false;
            btn_Blue_Bei.BackColor = Color.Gray;
            break;
        default:
            break;
    }
}
else
{
    this.Senden(klasse);
}
break;
```

Sendet die Klasse.

```
s = new Spielfeld("seaside");
```

Stellt Standard Spielfeld „Seaside“ ein. Wenn beide Spieler einem extra geöffneten Server beitreten können, stellt niemand ein Spielfeld ein.

```
s.Name = SendeBefehlLiesAntwort(clients[0], "fena");
```

Schlüsselwort „fena“. Rückgabe ruft Spielfeld.Name auf, was Namen setzt und Daten einliest.

```
case "fena":
spielfeld = new Spielfeld(hilfsstring);
sfvorbewegung = new Spielfeld(hilfsstring);
pb_Spielfeld.BackgroundImage = Image.FromFile("..\\..\\..\\data\\field\\" +
                                                hilfsstring +
                                                ".bmp");this.Senden(hilfsstring);
break;
```

Stellt eigenes Spielfeld und Übergabefeld ein und sendet Feldnamen an Server.

```
clients[1].Spieler.Partei = SendeBefehlLiesAntwort(clients[1], "fese" + s.Name);
```

Sendet Schlüsselwort „fese“ an anderen Spieler, um ihm Feld zu übergeben.

```
case "fese":
    spielfeld = new Spielfeld(empfangen.Substring(4));
    sfvorbewegung = new Spielfeld(empfangen.Substring(4));
    pb_SpielFeld.BackgroundImage = Image.FromFile("..\\..\\..\\data\\field\\" +
                                                    empfangen.Substring(4) + ".bmp");
    this.Senden(klasse);
    break;
```

Setzt alle nötigen Feldinformationen.

```
if (clients[1].Spieler.Partei == clients[0].Spieler.Partei)
{
    Console.WriteLine("Neue Klasse einstellen");
    clients[1].Spieler.Partei = SendeBefehlLiesAntwort(clients[1], "klas" +
    clients[0].Spieler.Partei);
}
```

Wenn beide Spieler die selbe Klasse gewählt haben, wird Spieler 2 erneut Schlüsselwort „klas“ gesendet. Im Client wird die Partei des Gegners dann unauswählbar.

```
clients[1].Senden(SendeBefehlLiesAntwort(clients[0], "tabw" + clients[0].ClNummer
+ "" + clients[0].Spieler.Geld + "" + clients[0].Spieler.Bewpnt ));
```

Server sendet Spieler 2 den Rückgabewert von dem was er von Spieler 1 bekommt. Schlüsselwort „tabw“ mit eigener Client Nummer, Geldbetrag und Bewegungspunkten

```
case "tabw":
    this.BeginInvoke((MethodInvoker)delegate { tablessControl1.SelectTab(4); });
    geld = Convert.ToInt32(empfangen.Substring(5,4));
    lbl_Geld_Anzeige.Text = geld.ToString(); ;
    clnummer = Convert.ToInt32(empfangen.Substring(4,1));
    bewpnt = Convert.ToInt32(empfangen.Substring(9));
    lbl_Punkte_Anzeige.Text = bewpnt.ToString();
    Senden("gek1"+klasse);
    break;
```

Tabwechsel zum Spielfeld. Alle empfangenen Werte werden gesetzt und die eigene Klasse wird mit Schlüsselwort „gek1“ erst an Server und dann an Spieler 2 gesendet.

```
case "gek1":
    gegnerklasse = empfangen.Substring(4);
    break;
```

Spieler zwei erhält die gegnerische Klasse.

```
clients[0].Senden(SendeBefehlLiesAntwort(clients[1], "tabw" + clients[1].ClNummer
+ "" + clients[1].Spieler.Geld + "" + clients[1].Spieler.Bewpnt ));
```

Der selbe Prozess genau andersrum, um Spieler 1 auch eine gegnerische Klasse zu senden.

```
string empfangen = SendeBefehlLiesAntwort(dran, "Start");
```

Sendet Start an Spieler 1, was dieser in einer Message Box ausgibt und wartet auf seine erste Aktion.

### 3.3.1 Ablauf eines Spieles

Da noch nicht alle Abläufe des Spiels programmiert sind, werde ich die Fehlenden nur beschreiben.

Spieler 1 Klickt auf ein Spawnfeld.

```
private void pB_SpielFeld_MouseClick(object sender, MouseEventArgs e)
{
    this.Senden("feld" + this.FeldAuswahl());
}
```

Klick auf das Spielfeld sendet Schlüsselwort „feld“ mit Koordinaten.

```
case "feld":
    aktuellekoordinaten = (empfangen.Substring(4).Split(','));
    empfangen = SendeBefehlLiesAntwort(dran, "fewe" +
    Convert.ToString(s.Feld[Convert.ToInt32(aktuellekoordinaten[0])],
    Convert.ToInt32(aktuellekoordinaten[1])) + parameter);
    break;
```

Ermittelt die Koordinaten des angeklickten Feldes und sendet den Feldwert 9 zurück.

```
case "9":
    if (btn_Level_UP.Enabled)
    {
        FeldÜberschreiben(sfvorbewegung, spielfeld);
        ElementeVerbergen();
        ArealLöschen();
    }
    if (clnummer == 0)
    {
        figurx = xkoord;
        figury = ykoord;
        ElementeAnzeigen();
    }
    break;
```

Die angeklickten Koordinaten werden erneut in Variablen gespeichert und die Buttons zum erzeugen einer Figur erscheinen. Der Server wartet jetzt auf eine Eingabe des Spielers.

```
private void btn_Figur_Erstellen_Click(object sender, EventArgs e)
{
    Button b = (Button)sender;
    if (geld < Convert.ToInt32(b.Text) || bewpnkt < 1)
    {
        MessageBox.Show("zu wenig Ressourcen");
    }
    else
    {
        geld -= Convert.ToInt32(b.Text);
        lbl_Geld_Anzeige.Text = geld.ToString(); ;
        bewpnkt -= 2;
    }
}
```

```

        lbl_Punkte_Anzeige.Text = bewpnkt.ToString();
        FigurZeichnen(b.Text, klasse, figurx, figury);
        this.Senden("neuf" + b.Text);
    }
    ElementeVerbergen();
}

```

Wie in 3.1.2 schon beschrieben, erzeugt der Eventhandler nach Aufruf durch Klick eine Figur. Sie sendet das Schlüsselwort „neuf“ mit der Art der Figur.

```

case "neuf":
    Spielfigur figur = dran.Spieler.NeueFigur(Convert.ToInt32(empfangen.Substring(4,
3)),
    Convert.ToInt32(aktuellekoordinaten[0]),
    Convert.ToInt32(aktuellekoordinaten[1]));
    s.Feld[Convert.ToInt32(aktuellekoordinaten[0]),
    Convert.ToInt32(aktuellekoordinaten[1])] = 6+last;
    clients[1 - last].Senden("neuf" + figur.ToString());
    empfangen = SendeBefehlLiesAntwort(dran, "neuf" + figur.ToString());
    break;

```

Der Server erstellt eine neue Figur und setzt den Wert des Spielfeldes auf 6. Außerdem sendet er die Information der Figur an beide Clients mit dem Schlüsselwort „neuf“.

```

case "neuf":
    figurdaten = empfangen.Substring(4).Split(',');
    if (spielfeld.Feld[Convert.ToInt32(figurdaten[6]), Convert.ToInt32(figurdaten[7])]
    == 9 || spielfeld.Feld[Convert.ToInt32(figurdaten[6]),
    Convert.ToInt32(figurdaten[7])] == 8)
    {
        FigurZeichnen(figurdaten[0], gegnerklasse, Convert.ToInt32(figurdaten[6]),
        Convert.ToInt32(figurdaten[7]));
    }
    Senden("weit");
    break;

```

Stellt die Figurdaten der Figur ein. Wenn das Feld an der Stelle immer noch einen Spawn wert hat, was nur bei dem anderen Client möglich ist, wird dort auch eine Figur erstellt. Sendet Schlüsselwort „weit“

```

case "weit":
    empfangen = SendeBefehlLiesAntwort(dran, "weit");
    break;

```

Client soll einen weiteren Zug machen.

Client 1 Klickt nun auf die gerade erzeugte Figur und erhält den Feldwert 6.

```

case "6":
    PlaySound("OnFigur");
    if (btn_Level_UP.Enabled)
    {
        FeldÜberschreiben(sfvorbewegung, spielfeld);
        ElementeVerbergen();
        ArealLöschen();
    }
    else
    {
        int zähler = 0;

```

```

        FeldÜberschreiben(spielfeld, sfvorbewegung);
        ElementeVerbergen();
        figurx = xkoord;
        figury = ykoord;
        if (clnummer == 0)
        {
            for (int i = 0; i < 7; i++)
            {
                if (i + figury - 3 >= 0 && i + figury - 3 < 20)
                {
                    for (int j = 0; j < 7; j++)
                    {
                        if (j + figurx - 3 >= 0 && i + figury - 3 < 20)
                        {
                            if (spielfeld.Feld[j + figurx - 3, i + figury - 3] ==
                                9 || spielfeld.Feld[j + figurx - 3, i + figury - 3] ==
                                8)
                            {
                                spielfeld.Feld[j + figurx - 3, i + figury - 3] = 5;
                            }
                            if (spielfeld.Feld[j + figurx - 3, i + figury - 3] ==
                                5)
                            {
                                spielfeld.Feld[j + figurx - 3, i + figury - 3] -=
                                    Convert.ToInt32(empfangen.Substring(5).Substring(zähler, 1));
                            }
                        }
                    }
                    zähler++;
                }
            }
        }
        else
        {
            zähler += 7;
        }
    }
    ArealZeichnen();
}
ElementeAnzeigen(empfangen.Substring(55).Split(','));
}
break;

```

Bewegungs- und Angriffsareal werden gezeichnet. Hierbei wird in 2 Schleifen geprüft, ob das zu zeichnende Feld überhaupt existiert, also größer 19 oder kleiner 0 ist und ob es sich um ein begehbares Feld handelt.

Das Erzeugen und Anklicken löst bei beiden Spieler die selben Methoden aus, nur dass Spieler 2 8 statt 9 als Spawnwert hat und 7 statt 6 als Figurenwert.

Das Beenden einer Runde:

Spieler 1 Klickt auf „Runde beenden“ Button.

```

private void btn_Runde_Beenden_Click(object sender, EventArgs e)
{
    DialogResult antwort = MessageBox.Show("Wollen Sie die Runde
beenden?", "Runde beenden", MessageBoxButtons.YesNo);

    if (antwort == DialogResult.Yes)
    {
        Senden("rube");
    }
}

```

```
}
```

Eine Message Box öffnet sich, die den Spieler fragt, ob er die Runde beenden will. Wenn ja, wird das Schlüsselwort „rube“ gesendet.

```
case "rube":  
    last = 1 - last;  
    dran = clients[last];  
    empfangen = SendeBefehlLiesAntwort(dran, "losg");  
    break;
```

Die Variable, die den Client wählt, wird auf 1 gesetzt und der Server sendet „losg“ Befehl an Spieler 2.

```
case "losg":  
    MessageBox.Show("Runde startet");  
    break;
```

Message Box öffnet sich, danach kann der Spieler Züge machen.

Bewegung, Angriff, LevelUp und das „Sterben“ einer Figur sind noch nicht programmiert, doch der Ablauf wäre folgender:

### **Bewegung:**

Spieler sendet Feldkoordinaten und bekommt 4 als Feldwert. Daraufhin schreibt er die gerade ausgewählte Figur auf das neue Feld und löscht die alte Figur. Er sendet dem Server die Information, das alte Feld zurück und das neue auf „Figur“ zu setzen. Der Server sendet Spieler 2 nun die selbe Information und die neuen Feldwerte an beide Clients.

### **Angriff:**

Nach einer erfolgreichen Überprüfung, ob sich die anzugreifende Figur im Angriffsbereich befindet, löst ein Sendebefehl an den Server *GreiftAn* mit der Übergeben Figur aus.

### **Level Up:**

Sendet dem Server den Befehl die *LevelUp* Methode auszuführen und dieser Sendet die neuen Figurenwerte an den Spieler.

### **Stirbt:**

Wenn die übergebene Figur der *GreiftAn* Methode unter 0 Gesundheitspunkte hat, wird sie vom Feld gelöscht, genau wie bei einer Bewegung.



## 4 Fazit

Das Spiel ist zwar zur Abgabe nicht fertig geworden, aber ich bin dennoch nicht komplett unzufrieden damit. Mein Ziel war es einen Einblick in komplexere Spielmechaniken zu bekommen und den hatte ich auf jeden Fall.

Der Entschluss, für das Senden einfache Streams statt Events zu verwenden, schien mir anfangs als sinnvoll, doch im Nachhinein wäre ich damit wahrscheinlich wesentlich weiter gekommen und vieles hätte sauberer programmiert werden können. Trotzdem bin ich mit der Verwendung von Windows Forms und TCP für die Verbindung recht zufrieden, da mir das relativ wenig Probleme bereitet hat.

Das wohl Schwierigste bei diesem Projekt war es, zu erkennen wie etwas programmiert sein muss, damit es auch noch funktioniert, wenn Methoden darauf zugreifen, an die ich zu diesem Zeitpunkt noch nicht mal gedacht hatte. Das ist auch der Grund, warum manche Sachen im Programm wahrscheinlich gar nicht nötig wären und nur aus Vorsicht programmiert wurden. Ich hatte zwar sehr Vieles im Voraus geplant und versucht alle Abläufe von vornherein festzulegen, aber fast alles musste mehrmals umgeschrieben werden, um zu funktionieren. Ich habe mir zwar zu jedem Teil dieses Projektes kleinere Programme geschrieben, die mir gewisse Probleme klarer machen sollten, was auch oft funktionierte, aber wenn man dann alles zu einem großen Ganzen zusammenbringt, merkt man erst, wo die wahren Probleme auftauchen.

Natürlich ist als nächstes erst mal die Fertigstellung geplant, aber selbst danach gibt es noch unzählige Dinge, die das Spiel verbessern oder erweitern würden. Mehrere Karten, weitere Klassen und Spielfiguren und Hintergrundmusik wären relativ leicht implementierbar, aber auch größere Erweiterungen, wie ein 3 oder 4 Spieler Modus, sind nicht undenkbar.

Abschließend sei noch erwähnt, dass ich trotz langer, nervenaufreibender Nächte, viel Spaß an diesem Projekt hatte und es gut finde, dass das 2BKI seinen Schülern die Chance gibt, so ein Projekt in kompletter Eigenarbeit durchzuführen.

## 5 Verzeichnisse

### 5.1 Literaturverzeichnis

Passant, Hans (2011) *Remove/Hide Tab Header(Switcher) of C# TabControl* [online]  
URL: <http://stackoverflow.com/questions/6953487/remove-hide-tab-headerswitcher-of-c-sharp-tabcontrol> [03.04.2013]

vgl. TcpClient (Jahr unbekannt) *TcpClient-Klasse* [online]  
<http://msdn.microsoft.com/de-de/library/system.net.sockets.tcpclient.aspx> [12.04.2013]

vgl. Delegates (Jahr unbekannt) *Delegates mal ganz einfach* [online]  
<http://www.hinzberg.net/csharp/csharp/csharp/delegates.html> [20.04.2013]

vgl. Amjad, Zeeshan (2001) *Multithreaded Programming Using C#* [online]  
<http://www.codeproject.com/Articles/1083/Multithreaded-Programming-Using-C>  
[20.04.2013]

Freesound.org (Jahr unbekannt) [online]  
<http://www.freesound.org> [10.04.2013]

### 5.2 Abbildungsverzeichnis

Abb. 1 Ansicht des Menüs

Abb. 2 Das Tutorial

Abb. 3 Allgemeine Ansicht Spieloberfläche

Abb. 4 Screenshot der Serverkonsole

### 5.3 Abkürzungsverzeichnis

TCP, Transmission Control Protocol

BMP, Bitmap

IP, Internetprotokoll

## 6 Bestätigung der Selbstarbeit

Ich versichere, dass ich die vorliegende Arbeit (Dokumentation und eigentliche Projektarbeit) mit dem Titel „*C# basiertes multiplayer online Strategiespiel*“ selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit, die anderen Werken im Wortlaut oder dem Sinn nach entnommen wurden, sind mit Angaben der Quellen kenntlich gemacht. Die beigelegten Darstellungen (Zeichnungen, Bilder) wurden von mir gefertigt, sofern sie nicht aus angegebenen Quellen übernommen wurden.

Der einzige Code der nicht von mir stammt, ist die Klasse TablessControl.cs. Sie wurde von Hans Passant als Antwort auf die Frage nach solch einem Code gepostet und ich habe sie dem Forenbeitrag unverändert entnommen. Die URL ist im Literaturverzeichnis als erste Quelle angegeben. Außerdem sei erwähnt, dass alle im Spiel enthaltenen Töne von einer Internetseite sind, auf der lizenzfreie Audiofiles zum freien Download angeboten werden. Auch diese URL ist im Literaturverzeichnis angegeben.

02.05.2013

Nicolai Schiele

## 7 Anhang

Abb1.

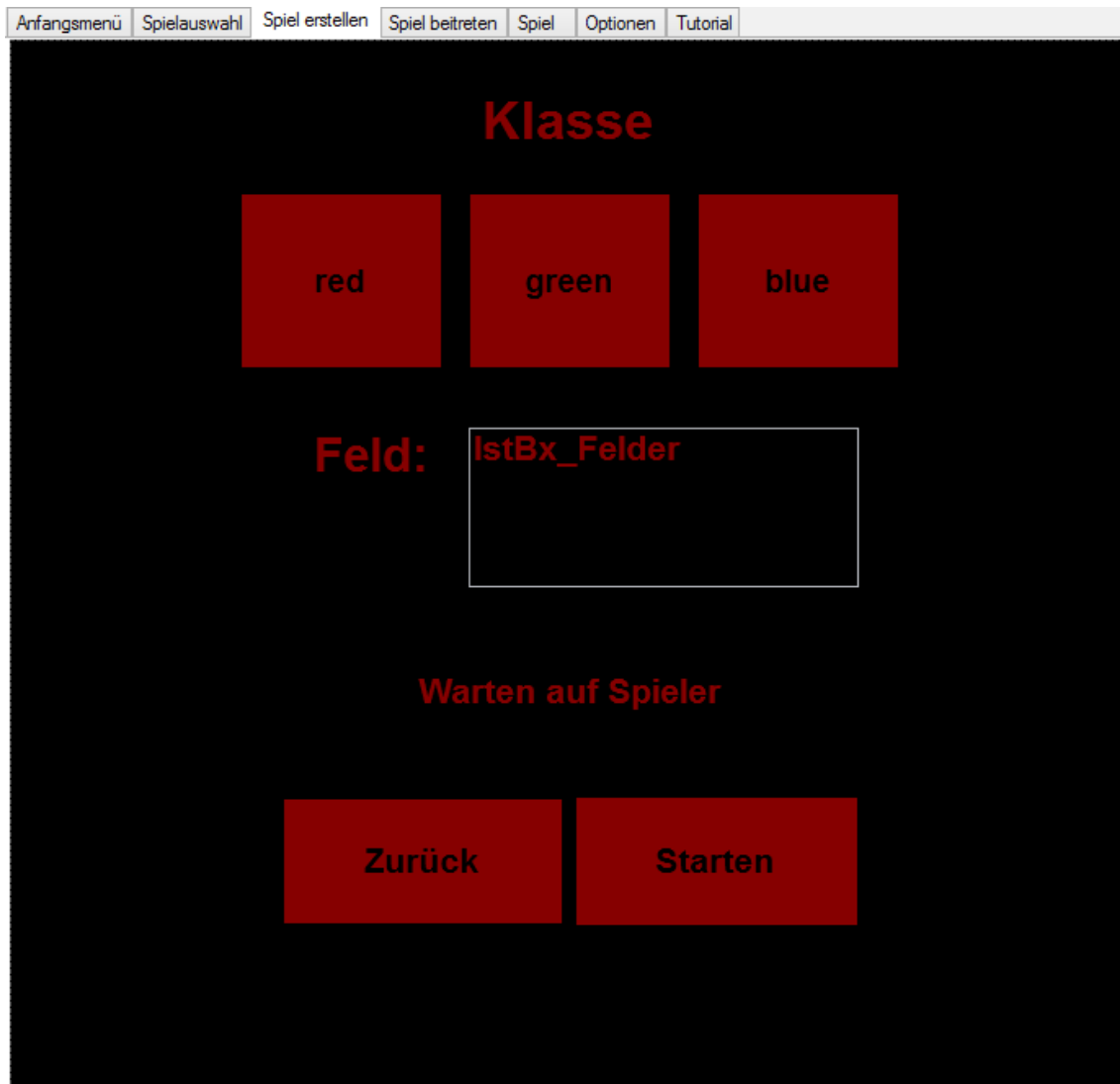


Abb. 2

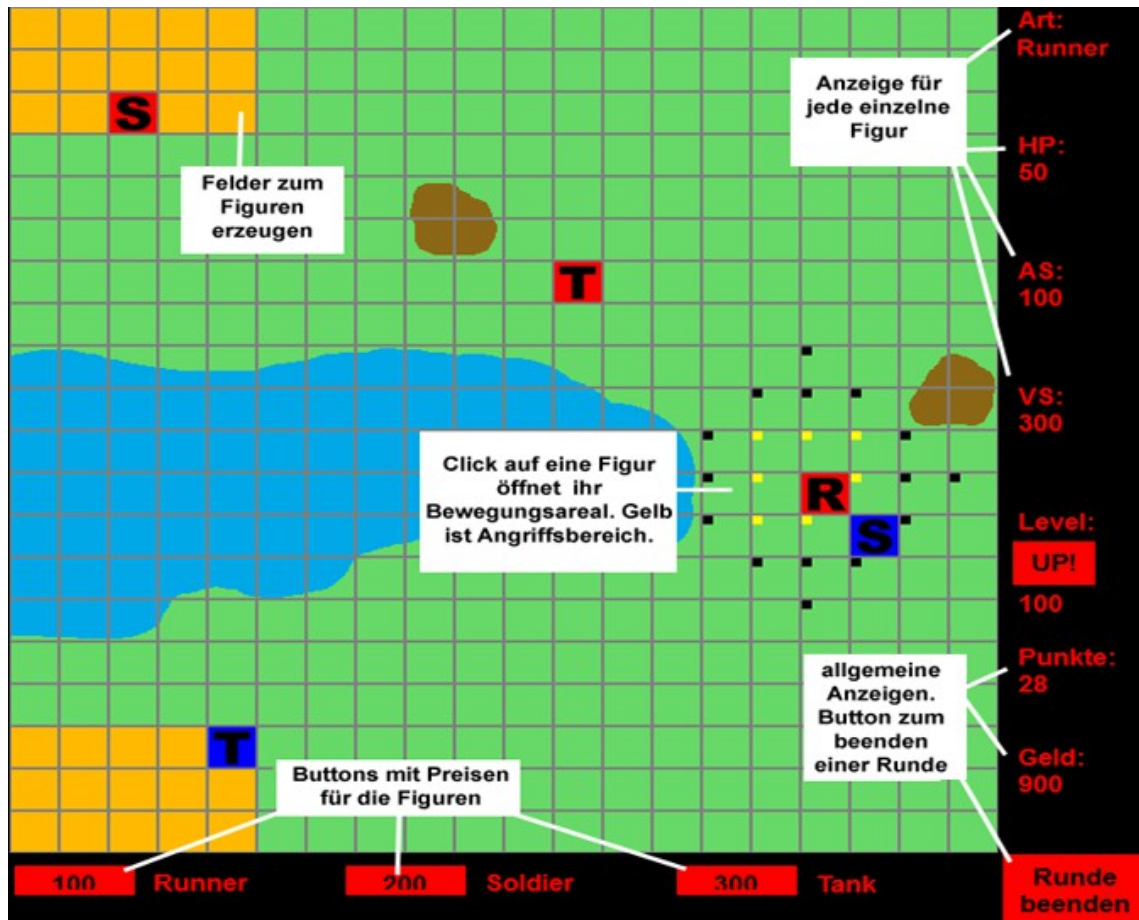
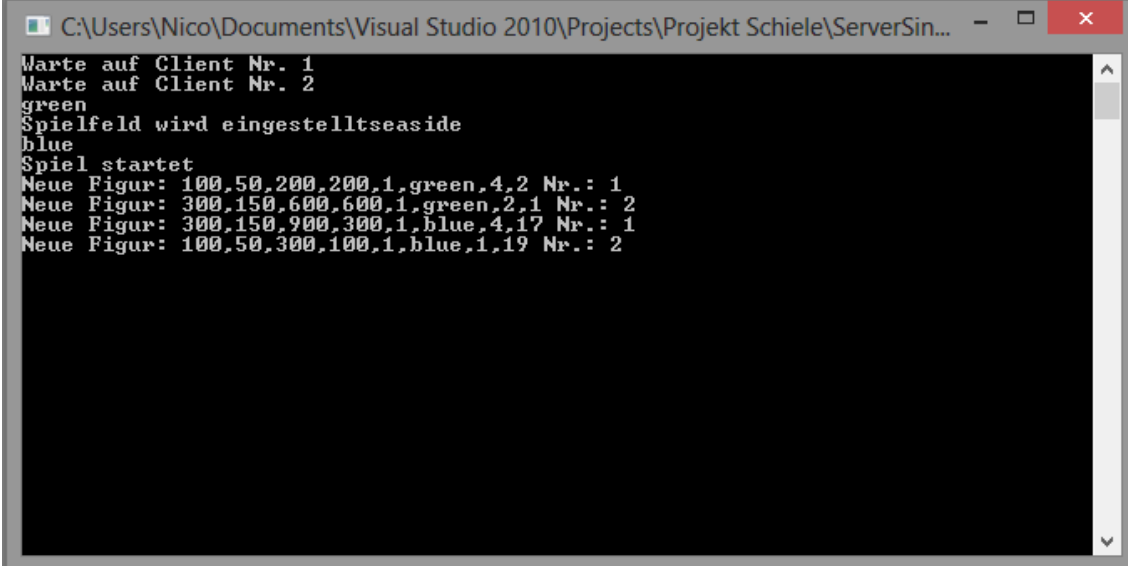


Abb. 3



Abb. 4



```
C:\Users\Nico\Documents\Visual Studio 2010\Projects\Projekt Schiele\ServerSin...  
Warte auf Client Nr. 1  
Warte auf Client Nr. 2  
green  
Spielfeld wird eingestellt  
seaside  
blue  
Spiel startet  
Neue Figur: 100,50,200,200,1,green,4,2 Nr.: 1  
Neue Figur: 300,150,600,600,1,green,2,1 Nr.: 2  
Neue Figur: 300,150,900,300,1,blue,4,17 Nr.: 1  
Neue Figur: 100,50,300,100,1,blue,1,19 Nr.: 2
```