HTL OTTAKRING

# Sudoku

## Lösen des Puzzles in C#

Lorenz Leutgeb und Moritz Wanzenböck

Arbeitsbeginn am 17.12.2009

Projektabgabe am 15.01.2010

Abgabe an Ing. Dipl.-Päd. Michael Rausch

# Inhaltsverzeichnis

# 1 Angabe

Programmieren Sie ein Programm zum Lösen von Sudokus. Achten Sie besonders auf Ein- und Ausgabemethoden.

## 1.1 Regeln des Sudoku

Das klassische Sudoku besteht aus 81 Feldern. Die Felder sind in neun Zeilen zu je neun Spalten angeordnet. Das Spielfeld wird in neun sogenannte *Boxen* unterteilt. Jede Box ist drei Felder lang und breit.

Das Spielfeld ist mit den Zahlen von Eins bis Neun so zu füllen, dass in jeder Zeile, jeder Spalte und jeder Box, jede Zahl nur ein Mal vorkommt.

Das bedeutet, dass in jeder Zeile nur eine Eins, eine Zwei usw. vorkommen darf. Dies gilt im gleichen Schema für Reihe und Box.

## 1.2 Unlösbarkeit von Sudokus

Ein Sudoku ist nicht lösbar wenn die Angabe einen Fehler aufweist.

Die Angabe weist einen Fehler auf, wenn eine Zahl der Angabe gegen die Regeln nach 1.1 verstößt, oder wenn die Angabe so gewählt ist, dass nicht alle Zahlen eingesetzt werden können.

In der Folge wird die zuerst genannte Art „Unlösbarkeit erster Ordnung“ oder „Fehler erster Ordnung“ bzw. „Unlösbarkeit zweiter Ordnung“, „indirekte Unlösbarkeit“ oder „Fehler zweiter Ordnung“ genannt.

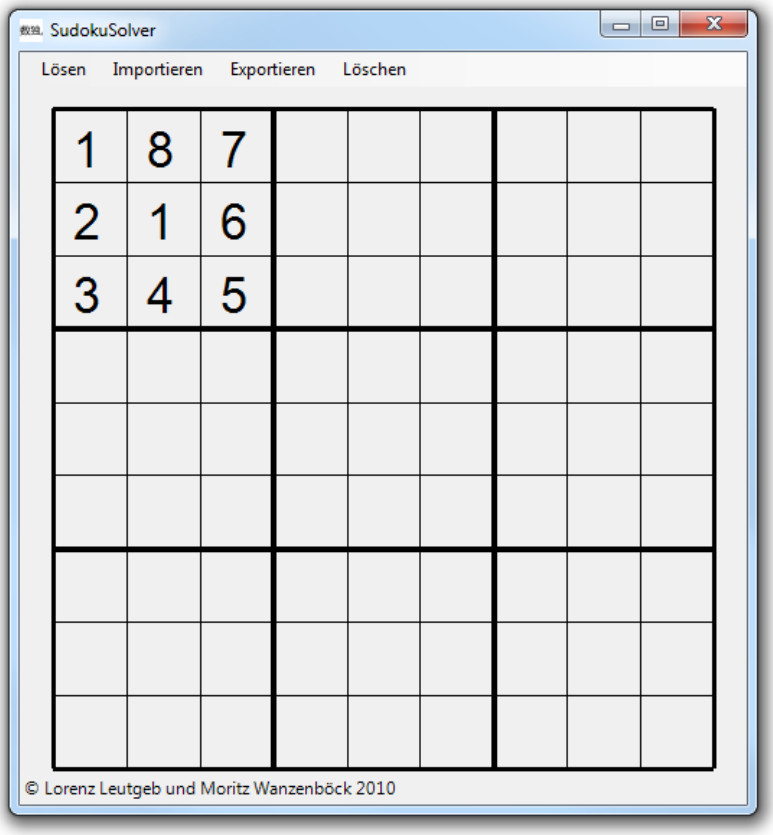


Abb. 1 - Fehler erster Ordnung

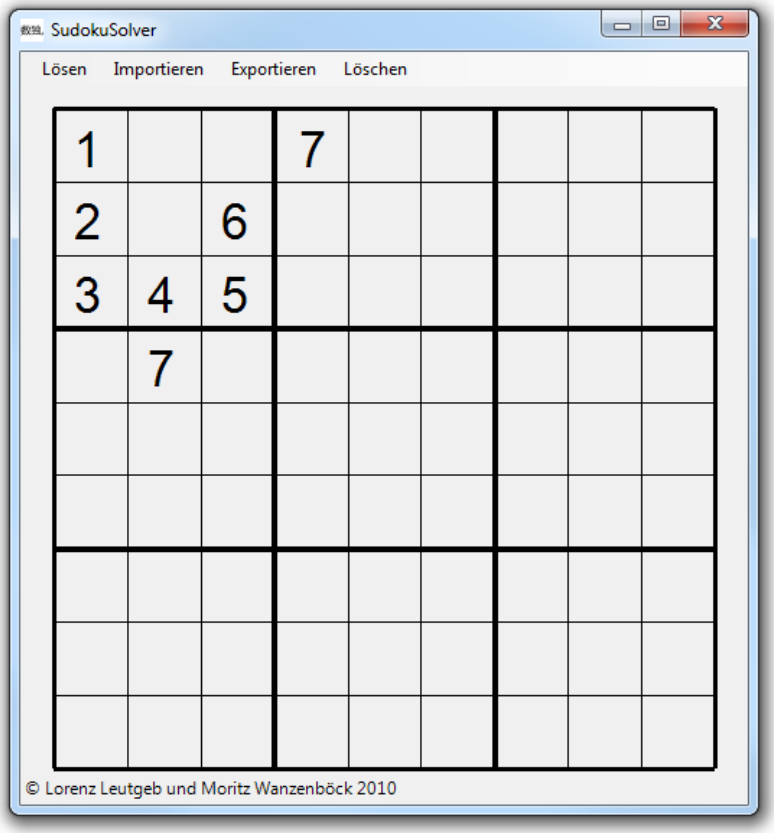


Abb. 2 - Fehler zweiter Ordnung

Die Problematik der Erkennung der Fehler zweiter Ordnung wird schnell deutlich. Eine Überprüfung nach 2.4 kann einen solchen Fehler nicht entdecken. Um diesen Fehler zu erkennen, müssen Zahlen in das Sudoku eingesetzt und deren Plausibilität geprüft werden. Die Überprüfung später im Programm erkennt während der Eingabe nur Fehler erster Ordnung. Beim Abschluss des Eingabevorgangs wird auch auf Fehler zweiter Ordnung geprüft, indem der Lösealgorithmus zur Hilfe genommen wird.

# 2 Überlegungen zum Algorithmus

## 2.1 Grundlegende Überlegungen

Ein Algorithmus, welcher Suokus lösen soll, muss überprüfen können, welche Zahl in welches Feld gesetzt werden darf und welche nicht. Dies muss beim Lösen berücksichtigt werden.

Viele Sudoku-Angaben bestehen aus nur wenigen ausgefüllten Feldern. In diesem Fall ergeben sich oft mehrere Lösungen für die gleiche Angabe. Der Algorithmus muss eine Lösung finden und darf in diesem Szenario nicht gestört werden.

Der Computer ist weitaus flexibler bezüglich der Fehlerkorrektur und rechnet vielfach schneller als der Mensch. Die Quantität der Rechenschritte kann also beim Lösen von Sudokus der Größe $9 \cdot 9$ und heutiger Rechenleistung vernachlässigt werden.

## 2.2 Lösungsmethode

Zum Lösen von Sudokus eignet sich Backtracking sehr gut. Diese Methode setzt entsprechend den Spielregeln *von links oben nach rechts unten* Zahlen in das Sudoku ein. Auf diesem Weg macht der Computer sehr schnell einen Fehler. Er setzt eine Zahl ein, die zwar zum Zeitpunkt des Einsetzens korrekt ist, aber später eine andere Zahl *blockiert* (siehe 1.2 – Fehler zweiter Ordnung). Doch der Computer kann diesen Fehler erkennen und sucht systematisch nach der Ursache. Er bessert ihn aus und versucht nun erneut das Sudoku zu lösen. Nach mehreren Versuchen sind alle Zahlen korrekt angeordnet und das Sudoku gelöst.

## 2.3 Pseudocode

Eine Formulierung des Algorithmus als Pseudocode:

```
funktion loesen(zeile, spalte)
{
        wenn(spalte == 10)
        {
           zeile += 1;
           spalte = 0;
        }

        wenn(zeile == 10)
           ende ok;

        wenn(feldleer(zeile, spalte))
        {
           schleife(i von 1 bis 9)
           {
              wenn(zahlerlaubt(zeile, spalte, i))
              {
                 setzen(zeile, spalte, i)
                 wenn(loesen(zeile, spalte + 1))
                    ende ok;
              }
              löschen(zeile, spalte)
           }
        }
        sonst
        {
           wenn(loesen(zeile, spalte + 1))
              ende ok;
           sonst
              ende fehler;
        }
}
```

Das Lösen ist abgeschlossen wenn versucht wird die 10. Zeile zu lösen. Dies geschieht nur wenn die neunte Zeile gelöst ist, was wiederum bedeutet, dass alle Zeilen des Sudokus gelöst sind und somit ein Ergebnis gefunden ist.

Wenn dies nicht der Fall ist, wird zuerst bestätigt, dass das aktuelle Feld nicht schon ausgefüllt ist, sonst würde die Angabe überschrieben werden. Liegt ein freies Feld vor, wird Schritt für Schritt jede Zahl von Eins bis Neun eingesetzt und für jede Zahl versucht ob das Sudoku lösbar ist. Ist das Feld schon ausgefüllt wird es übersprungen und mit dem Nächsten fortgefahren.

Wenn in der Schleife keine Lösung gefunden wird. Ist keine der Zahlen von 1 bis 9 richtig, und somit eine zuvor platzierte Zahl falsch. In diesem Fall wird das in Bearbeitung befindliche Feld wieder auf den Leerzustand gesetzt und der Fehler korrigiert.

## 2.4 Überprüfungsmethode

Um festzustellen welche Zahl laut den Regeln des Sudoku in welchem Feld platziert werden darf, muss eine entsprechende Formulierung der Überprüfung gefunden werden. Am leichtesten gestaltet sich der Vorgang zur Kontrolle der Zeile bzw. Spalte:

```
schleife(i von 1 bis 9)
   wenn(zahl(i, spalte) == zahl
      ende nichterlaubt;

schleife(i von 1 bis 9)
   wenn(zahl(zeile, i) == zahl)
      ende nichterlaubt;
```

Zwei Schleifen durchlaufen hintereinander die gegebene Zeile und Spalte. Sobald in dieser Iteration die zu prüfende Zahl auftaucht, wird der Vorgang beendet.

Die Ermittlung der Box ist etwas trickreicher. Dividiert man die $x$-Koordinate des zu prüfenden Feldes durch $3$ und vernachlässigt dabei die Nachkommastellen des Ergebnisses (sog. *Integerdivision*), so erhält man eine der Zahlen $0$, $1$ oder $2$. Multipliziert man diesen Wert wieder mit $3$, ergibt dies die kleinste $x$-Koordinate der entsprechenden Box. Dies lässt sich analog auf die $y$-Achse übertragen. Ein Beispiel:

Das Feld $4|5$ liegt in der zentralen Box des Spielfeldes. Die kleinsten Koordinaten der Box sind $3|3$. $\frac{4}{3}$ als Integerdivison ergibt $1$. $1 \cdot 3$ ist die kleinste $y$-Koordinate der Box. $\frac{5}{3} \cdot 3$ ergibt ebenfalls $3$ als $x$-Koordinate. Somit sind die kleinsten Koordinaten der Box gefunden.

Ist dies erledigt werden alle Felder innerhalb der Box überprüft:

```
integer minx = x / 3 * 3, miny = y / 3 * 3;

schleife(i gleich miny, solange i < miny + 3)
       schleife(j gleich minx, solange j < minx +3)
         wenn(zahl(i, j) == zahl)
           ende nichterlaubt;
```

Falls keine der Überprüfungen anschlägt, darf die geprüft Zahl gesetzt werden ohne die Regeln zu verletzen.

# 3 Das Programm

Der wesentlichste Teil des Programms ist die Funktion zum Lösen von Sudokus. Um die Eingabe eines Sudokus zu erleichtern, kann man unter dem Menüpunkt *Importieren* > *Manuell* direkt Zahlen eingeben. Die Eingabe wird automatisch überprüft und auftretende Fehler markiert. Zusätzlich besitzt das Programm eine Schnittstelle zum Lesen Von CSV und eigenen XML-Dateien.

## 3.1 Optionen

Mit einem Klick auf *Lösen* wird das aktuell importierte Sudoku wie unter Punkt 2 beschrieben gelöst.

Dazu wird die eigens formulierte Klasse `Sudoku` benützt, auf welche näher unter Punkt 5 eigegangen wird.

## 3.2 Exportieren

Das Programm kann Sudokus wahlweise in XML- oder CSV-Dateien exportieren. Der Benutzer wird in beiden Fällen aufgefordert den Dateinamen in welchen das Programm exportieren soll anzugeben.

## 3.3 Importieren

Sudokus können aus zuvor erstellten CSV- und XML-Dateien gelesen werden. Zusätzlich kann das Sudoku manuell eingegeben werden.

# 4 Programmcode

## 4.1 Lösen

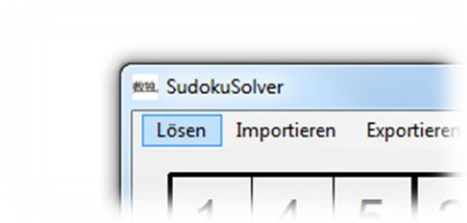


Abb. 3 - Menüpunkt "Lösen"

Wählt der Benutzer den Menüpunkt *Lösen*, wird die Methode `Solve()` aufgerufen. Es wird versucht das angezeigte Sudoku mithilfe von `Sudoku.Solve()` zu lösen und gelöst anzuzeigen. Gelingt dies nicht, wird eine Fehlermeldung angezeigt.

## 4.2 Import

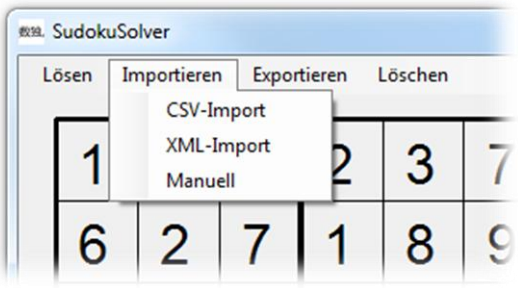


Abb. 4 - Menüpunkt "Importieren"

### 4.2.1 CSV

`CsvImport()` initialisiert einen Puffer, und öffnet einen Stream zur gewünschten Datei. Der Inhalt der Datei wird an `Sudoku.FromString()` übergeben, dort in `int[][]` umgewandelt und wieder im Puffer abgespeichert. Zuletzt wird das Sudoku auf seine Lösbarkeit überprüft und angezeigt.

### 4.2.2 XML

Die Methode zum Importieren von XML-Dateien, `XmlImport()`, öffnet einen Stream zur Datei, welche der Benutzer im Dialog ausgewählt hat und übergibt den Stream an

`Sudoku.FromXml()`, deren Rückgabewert im Puffer abgespeichert und auf seine Lösbarkeit geprüft wird.

### 4.2.3 Manuell

Wünscht der Benutzer ein Sudoku manuell über die Tastatur einzugeben, wird `StartImportMaunal()` aufgerufen. Die Funktion verändert über einen Aufruf von `SwitchMenu()` die Ansicht des Menüs und blendet die `TextBox`en zur Eingabe ein.

Wählt der Benutzer Bestätigen, wird der Inhalt der `Textbox`en in `int[][]` konvertiert und auf seine Lösbarkeit überprüft. Scheitert die Überprüfung, muss die Eingabe fortgesetzt werden.

## 4.3 Export

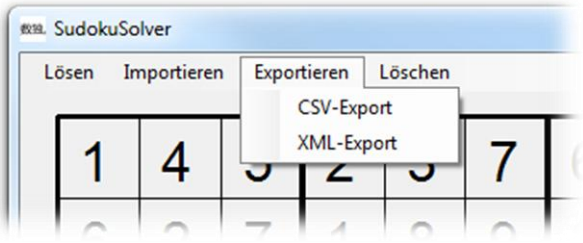


Abb. 5 - Menüpunkt "Exportieren"

### 4.3.1 CSV

`Sudoku.ToString()` konvertiert das Sudoku in einen CSV-kompatiblen `string`, welcher in die gewählte Datei geschrieben wird.

### 4.3.2 XML

Das aktuelle Sudoku wird mithilfe von `Sudoku.ToXml()` in einen Stream zur vom Benutzer ausgesuchten Datei geschrieben.

# 5 Die Klasse Sudoku

Um die Organisation des Programms zu verbessern wurde eine Art Bibliothek an Methoden, die während der Arbeit gebraucht wurden, erstellt und gesammelt. Das

Ergebnis ist eine Klasse, welche den Im- und Export von Sudokus als `int[][]` regelt, sowie Methoden zum Lösen und Prüfen von Sudokus enthält. Diese objektorientierte Lösung des Problems erleichtert die Übersicht und ist sehr portabel.

Die Klasse fängt nahezu keine Exceptions auf, da sie eher als Sammlung an Methoden fungiert und von außen überwacht werden sollte.

## 5.1 Solve

`Sudoku.Solve()` ist mit zwei verschiedenen Zugriffsmodifizierern und geringem Unterschied verfügbar. Die öffentliche Version dient zum Anstoßen der Rekursionen des Lösealgorithmus, innerhalb welcher die private Methode aufgerufen wird. Dies verhindert falsche Indices beim Starten des Lösungsvorgangs außerhalb der Klasse und vereinfacht außerdem den Zugriff auf den Algorithmus.

Das Grundgerüst der Methode ist eine konkrete Form des Codes aus Punkt 2. Der Rückgabewert entspricht „Sudoku gelöst.“ bei true und „Sudoku nicht gelöst.“ bei false. Trifft Zweiteres zu, liegt ein Fehler im Sudoku, d.h. in der Angabe, vor.

## 5.2 Check

`Sudoku.Check()` kombiniert verschiedenste Überprüfungen, die zur Laufzeit des Programms benötigt werden. Die wichtigste Form ist die Überprüfung ob die Zahl $n$ am Index $y|x$ eingesetzt werden darf. Diese Version wird im Lösealgorithmus häufig gebraucht und gibt `true` zurück, wenn die Zahl eingesetzt werden darf.

`Sudoku.Check(int[][] sudoku)` überprüft das übergebene Sudoku auf Fehler. Dazu wird zuerst jede Zahl überprüft nach den Grundregeln überprüft. Danach wird versucht eine Kopie des Sudokus zu lösen um Fehler zweiter Ordnung zu finden. Der Rückgabe wert ist `true`, wenn das Sudoku lösbar ist.

`Sudoku.Check(int[][] sudoku, SudokuBoxes boxes)` prüft das Sudoku nach den Grundregeln und markiert Fehler in den `TextBox`en des Objekts `boxes`.

## 5.3 Copy

Im Laufe der Entwicklung des Programms musste öfters ein Sudoku kopiert werden. Es wurde eine Methode zur Vollständigen Dereferenzierung nötig. Da `Object.Clone()` bei `int[][]` nicht alle Referenzen entfernt, kam diese Art der Referenzierung zum Einsatz.

## 5.4 ToString

`Sudoku.ToString()` konvertiert ein Sudoku in einen CSV-kompatiblen `string`. Die Methode wird zum Exportieren verwendet.

## 5.5 FromString

Um ein von `Sudoku.ToString()` konvertiertes Sudoku wieder zurück nach `int[][]` zu übersetzen wurde diese Methode integriert. Sie splittet den übergebenen `string` sehr simpel nach Zeilen und Spalten. Dabei werden `CRLF` und „`;`“ als Seperatoren verwendet.

## 5.6 ToXml

Um Sudokus in XML abzuspeichern und zu exportieren, wird ein Stream zur gewünschten Datei übergeben. Daraufhin wird das Sudoku per `XmlWriter` in den `Stream` geschrieben und dieser geschlossen.

## 5.7 FromXml

`Sudoku.FromXml()` ist das Gegenstück zu `Sudoku.ToXml()`. Ein `Stream` wird ausgelesen und als XML geparst. `XPath` ermöglicht einen einfachen Zugriff auf alle Zahlen im Sudoku, indem alle Elemente die zur Maske `/sudoku/row/cell/` passen iteriert und in einen Puffer geschrieben werden, welcher wieder zurückgegeben wird.

## 5.8 Buffer

`Sudoku.Buffer` ist eine Property, welche immer ein neu initialisiertes `int[][]` enthält. Dieser Vorgang wird des Öfteren gebraucht um leere Sudokus und Pufferspeicher zu deklarieren.

# 6 Die Klasse SudokuBoxes

Die manuelle Eingabe ist am einfachsten mit `TextBox`en zu lösen. Die Verwaltung von 81 `TextBox`en stellt dabei die größte Hürde dar. Diese einzeln anzusprechen würde einen immens langen Code fordern und sehr unübersichtlich enden. Um eine Ordnung in ein solches System zu bringen wurde `SudokuBoxes` geschrieben.

## 6.1 Konstruktor

Mit der Initialisierung werden alle 81 `TextBox`en in einem quadratischen Array initialisiert und geringfügig angepasst. Die `Form` in welche die `TextBox`en eingehängt werden, sowie das `Panel`, in welchem der Raster gezeichnet wird, werden bei der Konstruktion mit übergeben und gespeichert. Dem Resize-Event des `Form`-Objekts wird eine weitere Methode – `SudokuBoxes.Refresh()` – zugeordnet um die `TextBox`en neu zu platzieren und deren Größe anzupassen sobald sich die Skalierung des Rasters ändert.

## 6.2 Highlight

Um die Eingabe zu erleichtern, können über die Methode `SudokuBoxes.Highlight()` Zeile, Spalte so wie Box einer `TextBox` farbig hervorgehoben werden. Die Methode arbeitet sehr ähnlich wie `Sudoku.Check()` und färbt alle Felder ein anstatt sie zu überprüfen.

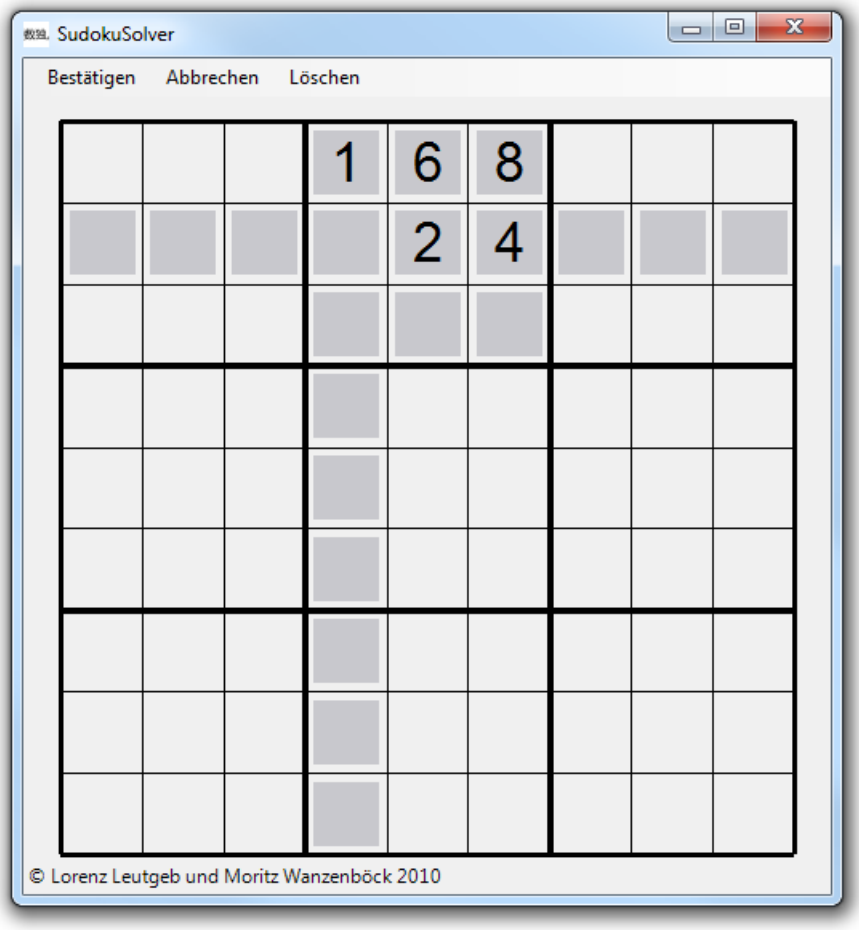


Abb. 6 - Demonstration von SudokuBoxes.Highlight()

## 6.3 SetForeColor

Diese Methode dient dem Ändern der Schriftfarbe einer `TextBox` im Array von außerhalb und leitet die Zuweisung gekapselt weiter.

## 6.4 Clear

`SudokuBoxes.Clear()` löscht den Inhalt aller `TextBox`en.

## 6.5 Refresh

Falls die `Form` ihre Größe ändert, müssen die `TextBox`en an die neue Größe angepasst werden. Dies geschieht mit `SudokuBoxes.Refresh()`.

## 6.6 GetIndex

Um in den Event-Methoden auf den Index der `TextBox` zu kommen, welche das Event ausgelöst hat, wurde diese Methode entworfen. Sie vergleicht den `sender` des Events mit allen `TextBox`en im Array und übergibt den Index der `TextBox`.

## 6.7 Sonstige Events

Die folgenden restlichen Events haben eine untergeordnete Bedeutung. Sie kümmern sich darum, dass nur Ziffern in die `TextBox`en eingetragen werden können und ermöglichen das Durchschalten der `TextBox`en über die Pfeiltasten der Tastatur.

- `SudokuBoxes.Enter()`
- `SudokuBoxes.KeyDown()`
- `SudokuBoxes.KeyUp()`
- `SudokuBoxes.TextChanged()`

## 6.8 Scale

`SudokuBoxes.Refresh()` und `SudokuBoxes()` benötigen einige Angaben zur Größe des `Panel`s um die `TextBox`en korrekt platzieren zu können. Der Datentyp `float` wurde gewählt, weil `int` in diesem Fall zu ungenau wäre und eine inkorrekte Platzierung zur Folge hätte.

## 6.9 Array

Diese Property wird benutzt um den Inhalt der `TextBox`en einfach nach `int[][]` zu konvertieren.

## 6.10 Visible

Die Property `SudokuBoxes.Visible` wird verwendet um alle `TextBox`en auf ein Mal aus- oder einzublenden. Falls die `TextBox`en eingeblendet werden sollen, werden diese auch in den Vordergrund geholt.

# Abbildungsverzeichnis