**Bericht Projekt M318**

*Projekt Dokumentation*

Verfasser: Fabio Litscher

Version: 1.0

Erstellungsdatum: 06.06.2014

**Inhalt**

[1. Einleitung 3](#_Toc389783366)

[2. Benutzerschnittstelle 4](#_Toc389783367)

[2.1. Aufgabenangemessenheit 4](#_Toc389783368)

[2.2. Selbstbeschreibungsfähigkeit 4](#_Toc389783369)

[2.3. Steuerbarkeit 4](#_Toc389783370)

[2.4. Erwartungskonformität 4](#_Toc389783371)

[2.5. Fehlertoleranz 4](#_Toc389783372)

[2.6. Individualisierbarkeit 4](#_Toc389783373)

[2.7. Lernförderlichkeit 4](#_Toc389783374)

[3. Fehlermeldungen 5](#_Toc389783375)

[4. Benutzergruppe 6](#_Toc389783376)

[4.1. Beurteilung entsprechend Benutzermodell 6](#_Toc389783377)

[4.2. Nötige Anpassungen / Änderungen 6](#_Toc389783378)

[5. Zustandsdiagramm 7](#_Toc389783379)

[6. Codekonventionen 8](#_Toc389783380)

[6.1. Variablen 8](#_Toc389783381)

[6.2. Funktionen 8](#_Toc389783382)

[6.3. Formular-Steuerelemente 8](#_Toc389783383)

[7. Programm Dokumentation 9](#_Toc389783384)

[7.1. Allgemein 9](#_Toc389783385)

[7.2. Wichtige Funktionen / Bestandteile 9](#_Toc389783386)

[7.2.1. Drei dimensionales Array 9](#_Toc389783387)

[7.2.2. Variante mit Pixel-Koordinaten 9](#_Toc389783388)

[7.2.3. Funktion für die Bestimmung der Koordinaten der einzelnen Felder 10](#_Toc389783389)

[7.2.4. Auf welches Feld habe ich geklickt 10](#_Toc389783390)

[7.2.5. Steine anhand von Array zeichnen 11](#_Toc389783391)

[7.2.6. Wichtigste globale Variablen 11](#_Toc389783392)

[LastColor 11](#_Toc389783393)

[LastPosition 11](#_Toc389783394)

[7.3. Ablauf Regelüberprüfung 12](#_Toc389783395)

[7.3.1. Reihenfolge der zu überprüfenden Regeln 12](#_Toc389783396)

[7.4. Erweiterungsmöglichkeiten 12](#_Toc389783397)

[8. Code Review 13](#_Toc389783398)

[9. Testkonzept 14](#_Toc389783399)

[10. Schlusswort 15](#_Toc389783400)

# Einleitung

In diesem Projekt, im Rahmen des Modules 318, geht es darum das Brettspiel Dame in der Sprache C# zu programmieren.

Das Ziel ist, dass man mindestens Spieler gegen Spieler spielen kann, ein Zusatz wäre, wenn man Spieler gegen Computer spielen könnte.

Da ich nach drei Wochen bereits alleine in der Gruppe war, da meinem Partner der Lehrvertrag aufgelöst wurde, wurde das Projekt dann von mir alleine realisiert.

Für mich lag die Schwierigkeit darin, dass ich bisher noch nie ein ähnliches Projekt realisiert habe, sowie ich so gut wie keine Programmiererfahrung im objektorientierten Bereich habe.

Das Spiel Dame ist ein klassisches Brettspiel, welches von jeder Altersgruppe gespielt werden kann. Damit die Bedienung möglichst einfach ist, habe ich mich entschieden, das ganze möglichst schlicht darzustellen und mit wenigen intuitiven Knöpfen die wichtigsten Funktionalitäten zu gewährleisten.

# Benutzerschnittstelle

## Aufgabenangemessenheit

**Gut:**

Unnötige Interaktionen wurden vermieden, indem beispielsweise am Anfang bereits feststeht, dass immer Weiss beginnt.

Standardwerte wie beispielsweise „Spieler 1“ und „Spieler 2“ sind bereits gegeben.

## Selbstbeschreibungsfähigkeit

**Durchschnittlich:**

Hilfen wurden keine grossen eingebaut, allerdings ist das Programm sehr verständlich aufgebaut, da es wenige Knöpfe gibt, bei denen man aber sofort weiss für was sie stehen.

## Steuerbarkeit

**Durchschnittlich:**

Da auf Zusatzfeatures verzichtet wurde, und vor Allem die Grundfunktionalität (Spieler vs. Spieler) gewährleitet wird, gibt es keine grossen Einstellungsmöglichkeiten, welche der Benutzer steuern könnte.

## Erwartungskonformität

**Durchschnittlich:**

Der Grossteil der Knöpfe ist selbsterklärend. Das heisst der Benutzer weiss immer was er macht, und macht nicht ausversehen etwas, was er eigentlich nicht wollte.

## Fehlertoleranz

**Durchschnittlich:**

Es existieren nicht viele Fehlerquellen im Programm, da nicht viele Eingabefelder existieren.

Was allerdings überprüft wird, ist ob der Spieler nach Regeln spielt, man kann also keinen Stein auf ein Feld setzen, wenn man das laut Regelwerk nicht darf.

## Individualisierbarkeit

**Weniger berücksichtigt:**

Der Spieler hat keine grossen Anpassungsmöglichkeiten, die Werte sind vorgegeben.  
Es wurde weniger Wert darauf gelegt, da an erster Stelle die Funktionalität stand.

## Lernförderlichkeit

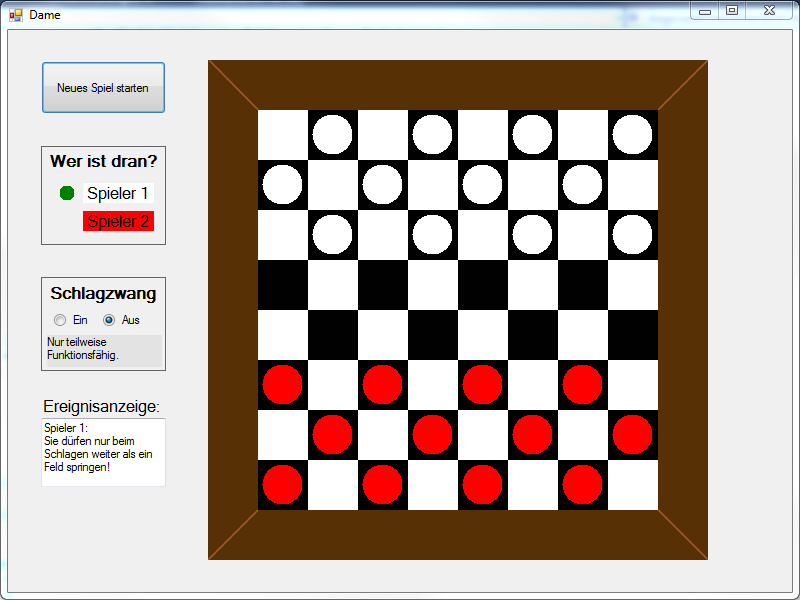
**Gut:**

Das Programm ist so aufgebaut, dass auch Neuanwender keine Probleme haben das Programm zu bedienen.

# Fehlermeldungen

Fehlermeldungen, wurden bei meinem Programm praktisch keine implementiert, bis auf diejenigen, die in Einsatz treten, wenn man einen unzulässigen Spielzug durchführen möchte. Diese werden jeweils links unten neben dem Spielfeld in der Ereignis-Konsole angezeigt.

Beispielsweise kommt folgende Meldung, wenn der Spieler 1 mit einem seiner weissen Steine, weiter als ein Feld fahren möchte, ohne dass er einen gegnerischen Stein schlagen kann:



# Benutzergruppe

## Beurteilung entsprechend Benutzermodell

Laut den Definitionen der drei verschiedenen Benutzergruppen, entsprechend dem Benutzermodell (Ergonomie\_BalzertLE16.pdf), würde ich mein Programm folgendermassen beurteilen:

|  |  |  |
| --- | --- | --- |
| **Anfänger** | **Gelegenheitsbenutzer** | **Experten** |
| * Einfache Bedienung durch wenige, dafür klare Funktionen (Knöpfe). * Keine grossen Einstellungsmöglichkeiten. | * Bedienung wie bei anderen gewohnten Programmen. * Keine Schwierigkeiten um zu verstehen, was passiert, wenn man beispielsweise auf einen Knopf klickt. | * Wenn man weitere Funktionen implementieren möchte, ist das eine gute Basis. * Überhaupt keine Probleme bei der Bedienung, findet sich blind zurecht. |
| * Keine Hilfe mit einer ausführlichen Erklärung zum Programm. * Keine interaktiven Hilfestellungen. | * Funktionsumfang evtl. etwas knapp, da kein Modus Spieler vs. Computer möglich ist. | * Geringer Funktionsumfang. * Wenige Einstellungs-möglichkeiten. |

Gemäss dieser Bewertung, würde ich sagen, ist mein Programm hauptsächlich für Anfänger und Gelegenheitsbenutzer entworfen.

## Nötige Anpassungen / Änderungen

Um den Experten-Bereich besser abzudecken, wären folgende Zusätze / Anpassungen nötig:

* Grösserer Funktionsumfang
  + Spieler vs. Computer
  + Speichern von Highscore
  + Spielstand in Datei speichern, Spielstand wieder aus Datei laden
* Mehr Einstellungs- / Personalisierungsmöglichkeiten
  + Möglichkeit zum Anpassen des Farbschemas
  + Beliebiger Name eingeben
    - Macht erst Sinn wenn ein Highscore implementiert wird
  + Skalierbarkeit

Folgende Sachen wären mit geringem Aufwand möglich zu implementieren:

* Anpassung der Farbe der Spielsteine
  + Mit Hilfe einer kleinen Auswahl aus 4 - 5 fix festgelegten Farben.
* Anpassung des Spieler Namens (anstatt Spieler 1)
  + TextBox anstatt Label machen, dann kann man den Namen bearbeiten.
* Skalierbarkeit
  + Wollten wir anfangs umsetzen, dann aber auf Eis gelegt, alles auskommentiert.
* Spielstand in Datei speichern
  + Alle Array Daten in ein Textdokument ausgeben.

# Zustandsdiagramm

# Codekonventionen

## Variablen

Weil bei mir die Verständlichkeit und Orientierung im Programm sehr wichtig war, dass ich mich selber zurechtfinde, habe ich sowohl englische als auch deutsche Bezeichnungen verwendet.

Anfangs wollte ich alles auf englisch machen, habe dann aber gemerkt, dass es für mich einfacher ist, wenn ich das verwende, das mir gerade in den Sinn kommt, sonst muss ich immer überlegen, wie z.B. etwas auf Englisch heisst.

Die Variablen beginnen immer mit einem Kleinbuchstaben.

Es gibt keine Prefixes für globale Variablen oder member Variablen einer Klasse.

Beispiele:

* lastPositionX

Dies können also globale Variablen oder auch member Variablen einer Klasse sein.

* feldX
* fields[]

## Funktionen

Auch bei den Funktionen habe ich sowohl englische als auch deutsche Namen vergeben, auch hier wieder wegen der Verständlichkeit und persönlichen Orientierung in meinem Programm.

Beispiele:

* setFixFields()
* berechneKoordinaten()

## Formular-Steuerelemente

Bei den verschiedenen Steuerelementen habe ich für die verschiedenen Typen die jeweiligen Abkürzungen / Prefixes verwendet.

Der Name ist folgendermassen aufgebaut: *prefix\_name*

Der Name kann sowohl englisch als auch deutsch sein.

Beispiele:

* Textbox 🡪 txt\_console
* Button 🡪 btn\_newRound
* Label 🡪 lbl\_console
* Picturebox 🡪 pic\_spielfeld
* Panel 🡪 pnl\_schlagzwang
* RadioButton 🡪 rdb\_schlagzwangEin

# Programm Dokumentation

## Allgemein

Allgemein ist zu sagen, dass sich alle Funktionen in einer Klasse befinden. Der Grund ist folgender:

Da ich bisher noch nie gross objektorientiert programmiert habe und ich deswegen das ganze Wissen erst aufbauen musste, hat mir Herr Vogel geraten, das Ganze in nur einer Klasse zu programmieren, da so bei Fehlern auch eine Fehlerquelle weniger vorhanden ist und es für mich evtl. etwas verständlicher ist.

Ausserdem ist mein Code eher einfach geschrieben. Das heisst, ich hätte noch einige Vereinfachungen machen können. Da ich das aber eher gegen Ende des Projektes realisiert habe, als ich schon ein wenig mehr vom objektorientierten Programmieren verstand und der Zeitaufwand vieles neu zu machen zu gross war, habe ich alles so belassen. Dafür verstehe ich genau was ich programmiert habe, da ich alles selber gemacht habe und zwar so, dass ich es mit meinem Fachwissen verstehe.

## Wichtige Funktionen / Bestandteile

Hier sind nur einige Funktionen und Variablen aufgeführt. Ich habe diese Sachen herausgenommen, die ich persönlich wichtig finde, und die auch meine Grundsteine waren, damit ich überhaupt so weit gekommen bin bei diesem Programm.

### Drei dimensionales Array

Das Herzstück meines Programms ist ein drei dimensionales Array namens fields[].

Die drei Dimensionen:

1. X - Achse
2. Y - Achse
3. Daten

Das heisst das erste Feld links oben wäre das Feld fields[0, 0].

Folgende Daten werden in der dritten Dimension gespeichert:

1. X - Koordinaten 🡪 in Pixel
2. Y - Koordinaten 🡪 in Pixel
3. Gültiges Feld 🡪 0 = zulässig 1 = unzulässig
4. Damenstein 🡪 0 = nein 1 = ja

### Variante mit Pixel-Koordinaten

Da als Spielfeld ein Bild verwendet wird, welches einfach in einer PictureBox ist, und ich irgendwie herausfinden muss, auf welches Feld man klickt, habe ich mich entschlossen, das ganze mit Pixel-Koordinaten zu machen

Der erste Schritt ist herauszufinden, auf welche Koordinaten man klickt, da habe ich anfangs *Mouseposition.X* und *Mouseposition.Y* verwendet. Das Problem bei dieser Variante war, dass die Pixel immer vom Bildschirmrand aus gemessen wurden, das heisst, wenn das Fenster weiter rechts ist, ist die X-Koordinate grösser, als wenn das Fenster ganz links ist.

Zuerst habe ich versucht das auszugleichen mit folgender Formel:

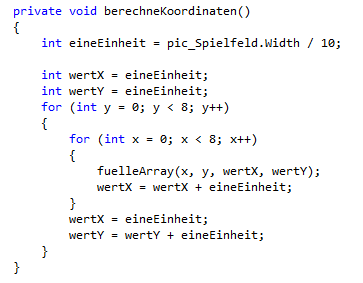
Mouseposition.X - Fensterkoordinate.X - PictureBox.X

* Hätte theoretisch funktionieren sollen, hat praktisch aber immer um einige Pixel nicht gestimmt.

### Funktion für die Bestimmung der Koordinaten der einzelnen Felder

Wenn das Programm geladen wird, werden in folgender Funktion die Koordinaten der einzelnen Felder errechnet und im Array hinterlegt.

Die Variable **eineEinheit** stellt die Seitenlänge eines einzelnen Feldes dar, diese ist genau 1/10 so gross wie die Seitenlänge der gesamten Picturebox (Da der Rand gleich breit ist wie eine Feld).

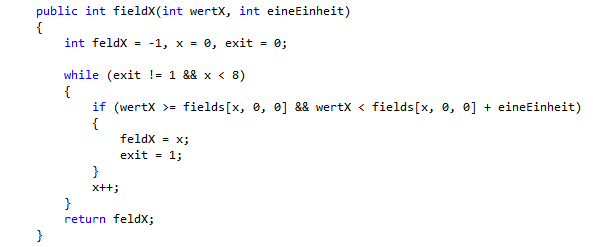


In der for-Schlaufe werden für alle Felder die X- und Y-Koordinaten errechnet, indem immer eine Einheit addiert wird. Die Funktion **fuelleArray()** füllt die Daten dann ins Array ab.

### Auf welches Feld habe ich geklickt

Nun haben wir die Koordinaten des Mausklicks und die Daten der einzelnen Felder sind im Array gespeichert.

Mit folgender Funktion wird berechnet, wo auf der X-Achse man geklickt hat.



Der Funktion werden der X-Wert des Mausklicks sowie die Länge einer Einheit mitgegeben.

Nun wird immer abgefragt, ob der X-Wert im Bereich eines Feldes liegt, wenn ja geht es aus der Schlaufe raus, wenn nein, wird der Zähler x um eins erhöht, und es wird abgefragt, ob der Klick im Bereich des nächsten Feldes war. Das passiert solange bis ein Feld zutrifft.

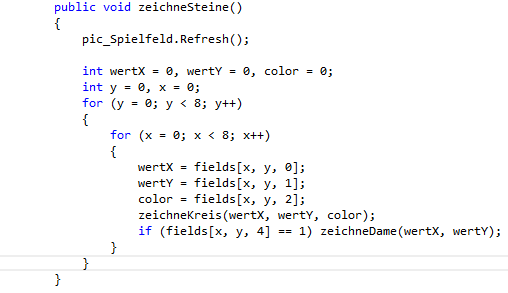
Dasselbe passiert natürlich auch noch in Y-Richtung.

### Steine anhand von Array zeichnen

Nach der Funktion für die Bestimmung des Feldes, war der nächste Schritt, dass ich Steine anhand der Angaben im Array zeichnen konnte.

Diese Funktion sieht folgendermassen aus:

Hier wird immer das ganze Array durchlaufen und die Steine werden entsprechend den Werten im Array gesetzt, indem die Funktion **zeichneKreis()** für das Zeichnen des Spielsteins aufgerufen wird und die Funktion **zeichneDame()** für das Zeichnen des D auf dem Stein, falls es sich um einen Damenstein handelt.



### Wichtigste globale Variablen

LastColor

Mit Hilfe der Variable **lastColor** weiss das Programm immer, was für ein Stein man angehoben hat, da diese Variable immer gesetzt wird, wenn der Spieler der am Zug ist einen Stein anhebt um ihn zu verschieben.

Wenn der Stein dann auf einem leeren Feld (auf welches der Stein laut Regeln gesetzt werden darf) abgesetzt wird, wird diese Variable benötigt, dass dem Feld die richtige Farbe zugeteilt wird.

LastPosition

Die Variablen **lastPositionX** & **lastPositionY** werden immer gesetzt, wenn man auf ein Feld mit einem Stein drauf klickt, damit man weiss, wo der angehobene Stein vorher war.

Diese Variable wird bei den meisten Regelüberprüfungen verwendet.

## Ablauf Regelüberprüfung

### Reihenfolge der zu überprüfenden Regeln

Die Regeln werden in folgender Reihenfolge überprüft:

1. Richtiger Spieler am Zug?
2. Ist das Feld gesperrt?
   * Alle weissen Felder sind gesperrt (In Array hinterlegt).
3. Ist die Fahrtrichtung diagonal?
4. Ist das gewünschte Feld bereits besetzt?
5. Falls es sich um KEINEN Damenstein handelt, ist die Fahrtrichtung vorwärts?
6. Stimmt die diagonale Distanz?
   * Darf nur ein Feld gefahren werden, ausser beim Schlagen wenn ein Stein übersprungen wird.
   * Hier wird auch überprüft, ob man wenn man weiter als ein Feld gefahren ist, einen gegnerischen Stein übersprungen hat. Deshalb gibt es für Damensteine eine separate Funktion, da Damenstein auch rückwärtsfahren und schlagen dürfen.
7. Danach wird der Schlagzwang überprüft.
   * Funktioniert noch nicht zu 100%, deshalb lässt er sich ein- bzw. ausschalten.
8. Wenn man bereits einmal geschlagen hat, wird überprüft, ob man noch weiter schlagen kann.
   * Variable **schongeschlagen** wird auf 1 gesetzt, wenn man schlägt, so kann überprüft werden, ob bereits geschlagen wurde.

Wenn eine Regel greift, sprich verletzt wird, wird der Stein anhand der lastPosition und der lastColor in der entsprechende Farbe auf das Feld gesetzt, von dem er angehoben wurde.

Die Regeln sind ausprogrammiert im Code ersichtlich. Sie wurden hier nicht aufgeführt, da diese Funktionen zum Teil sehr gross sind und es die Dokumentation somit unübersichtlich gestalten würde.

## Erweiterungsmöglichkeiten

Hier sind einige Erweiterungsmöglichkeiten die machbar wären:

* Grösse des Spielfelds der Fenstergrösse anpassen.
  + Auskommentierte Funktionen sind vorhanden, da wir das anfangs fix implementieren wollten. Haben wir dann aber nicht gemacht, da andere Funktionalitäten wichtiger waren.
* Farben der Spielstein anpassbar.
  + Wären nur einige Zeilen Code mehr, nicht schwierig zu implementieren, wenn man beispielsweise 5 Farben zur Auswahl stellt.
* „Reallife Modus“
  + Das wäre meiner Idee nach ein Modus, indem alle Regeln ausgeschaltet sind und man spielt wie wenn man auf einem richtigen Spielbrett spielt.

# Code Review

Da ich keinen Projektpartner habe um das Code Review durchzuführen, habe ich meine Mitlernende in meinem Betrieb, welche im 3. Lehrjahr ist gefragt, ob sie das mit mir machen könnte, da es meiner Meinung nach so viel mehr Sinn macht, als wenn ich das alleine machen würde.

# Testkonzept

Folgende Sachen wurden getestet:

* Einfacher Zug nach vorne sollte fehlschlagen
* Einfacher Zug auf leeres Feld
* Zug auf Feld ausserhalb des Spielfelds sollte fehlschlagen
* Einfacher Zug rückwärts sollte fehlschlagen
* Einfacher Zug mit Dame rückwärts sollte funktionieren
* Einfacher Zug auf Feld in letzter Linie sollte Stein in einen Damenstein umwandeln
* Einfacher Zug auf besetztes Feld schlägt Fehl
* Angriffszug sollte zwei Felder nach vorne und Feindlichen Spieler töten
* Angriffszug über freundlichen Spieler sollte nicht funktionieren
* Angriffszug über leeres Feld sollte nicht funktionieren
* Angriffszug auf Feld in letzter Zeile ergibt eine Dame
* Nach Angriffszug sollte ein weiterer Angriffszug möglich sein
* Nach Angriffszug sollte kein einfacher Zug mehr möglich sein
* Wenn ein Spieler eine Angriffszug verpasst, sollte dieser gefressen werden können, jedoch nicht wenn der Spieler einen anderen Angriffszug gemacht hat

Die Tests wurden ausgeführt, indem ich die verschiedenen Sachen einfach ausprobiert habe. Zuerst sind die einzelnen Regeln separat ausprobiert worden und dann auch im Zusammenspiel mit den anderen Regeln.

# Schlusswort