Projekt Bauernopfer

Team: Amon Druffel / Projekt: Bauernopfer

19.10.2017

Inhaltsverzeichnis

Einleitung

1.1 Vorwort

1.2 Beschreibung der Aufgabe

Das Spiel Bauernopfer ist eine Variante des Schachspiels bei dem eine Anzahl an weißen Bauern einen schwarzen Turm fangen müssen. Der Turm bewegt sich standardmäßig, die Bauern können sich vertikal und horizontal jeweils ein Feld bewegen.
Geprüft werden soll, wie viele Züge eine variable Anzahl von Bauern benötigt, um den schwarzen Turm zu fangen.

Lösungsidee

2.1 Technische Umsetzung

Das Projekt wird in der Programmiersprache Java und mit der Java EE Technologie JSF umgesetzt. Das Programm wird in einem Applikationsserver gestartet und ausgeführt.
Im Rahmen dieses Projektes wird auf einem lokalen Wildfly Server entwickelt. Das Programm wird später jedoch bei dem PaaS Anbieter Heroku hochgeladen und innerhalb eines Tomcat Servers deployed.

2.2 Probleme und Lösungen

2.2.1 Probleme

- Schwarz muss zu jederzeit perfekt spielen und es vermeiden in die Nähe eines weißen Bauern zu kommen
- Weiß muss sich dem schwarzen Turm nähern und dafür sorgen das der Turm nicht entkommt.

2.2.2 Lösungen

- Damit Schwarz nicht gefangen wird, wird vor jedem Zug geprüft, ob es in Reichweite ein sicheres Feld gibt. Ein sicheres Feld hat weder auf der horizontalen noch auf der vertikalen Achse eine weiße Figur stehen.
- Vor jedem Zug wird eine optimale Bewegungsrichtung festgelegt, basierend auf der relativen Position zu Schwarz. Dann soll geprüft werden, ob das Feld bereits besetzt ist. Ist dies der Fall, wird die zweitbeste Bewegungsrichtung verwendet.

Umsetzung

3.1 Grafische Oberfläche

Die grafische Oberfläche ist als Weboberfläche realisiert. Verwendet dafür wurde die Java EE Technologie. Als Hilfsmittel wurde die Bibliothek JSF eingebunden.

Das Schachbrett wird mit einer DataTable dargestellt. Diese iteriert über eine Liste mit Objekten, die jeweils eine Zeile des Schachbrettes darstellen. Jede Row beinhaltet eine Liste mit Feldern. Jede Zelle der Tabelle wird mit einem Bild gefüllt, welches im Feld Objekt gesetzt ist.

Im unteren Bereich der Seite, unter dem Schachbrett, befinden sich einige Buttons zur Steuerung der Oberfläche.

3.2 Aufteilung der Weboberfläche

Auf der Startseite der Webanwendung befindet sich ein Navigationsmenü welches zu den jeweiligen Aufgaben verlinkt.

Für jede Aufgabe wird eine eigene View mit dem dazugehörigen BackingBean verwendet. Im oberen Bereich der einzelnen Aufgaben befindet sich das Schachbrett. Unterhalb des Schachbretts befindet sich eine weitere Navigation zur Steuerung der Figur.

Next Führt den nächsten Zug aus und macht ihn auf dem Brett sichtbar.

Apply Speichert die Eingaben aus den Textfeldern und startet die Simulation neu.

Autoplay Führt automatisch die nächsten Züge aus. Lässt sich durch erneutes Anklicken wieder ausschalten.

Restart Startet die Simulation erneut.

3.3 Vorbereiten des Spielbretts

3.3.1 Platzieren der weißen Figuren

Für das Platzieren der weißen Figuren gibt es 3 sinnvolle Möglichkeiten.

1. Nach Zufall auf dem Schachbrett verteilen
2. Horizontal in der Mitte des Schachbrettes
3. Diagonal durch das Schachbrett

Zum Fangen des schwarzen Turms bieten sich die beiden letzteren Strategien eher an, die diese bei einer Standardanzahl von Bauern garantieren, dass der Turm gefangen werden kann. In dem Projekt wird nur die horizontale Strategie verwendet. Bei der ersten Aufgabe wird die Zeile in der die

Figuren aufgestellt werden zufällig gewählt. Bei den späteren Aufgaben werden die Figuren standardmäßig in der Mitte des Schachbrettes auf Zeile 4 platziert.

3.3.2 Platzieren der schwarzen Figur

Die Position des schwarzen Turms wird erst mal zufällig ausgewählt. Die einzigen Richtlinien für das Platzieren des Turms oder der Königin sind:

1. Das Feld darf nicht vorher besetzt sein.
2. Es darf keine weiße Figur direkt an das Feld angrenzen. Mindestens ein Feld dazwischen muss frei sein.

3.3.3 Interne Speicherung der Figuren und Positionen

Nach dem Platzieren einer Figur wird eine Referenz zu dem Feld, in dem die Figur platziert ist, in eine Liste gespeichert. Diese Liste wird benötigt um die Position der einzelnen Figuren auf dem Brett später einfacher und performanter heraus zu finden.

Nach jeder Positionsänderung einer Figur wird die alte Position aus der Liste wieder entfernt, und die neue hinzugefügt. Zudem wird für die erste Aufgabe eine weitere Liste erstellt, in der nach jedem weißen Zug, die ID der Figur gespeichert wird. Damit wird garantiert, dass die Figuren nacheinander ziehen.

3.4 Zug Generator

Der Algorithmus zum Generieren des nächsten Zuges besteht aus mehreren Teilen. Als erstes wird für die Figur eine optimale Bewegungsrichtung ermittelt. Diese verhält sich von Aufgabe zu Aufgabe für Weiß unterschiedlich. Für die erste Aufgabe besteht der Teil lediglich daraus, zu entscheiden ob die Figuren sich nach unten oder nach oben bewegen müssen.
Für die anderen Aufgaben sollen die weißen Figuren die horizontale Richtung vorziehen und versuchen auf den selben Feld-Index zu kommen, wie die schwarze Figur.

Schwarz soll sich bei der ersten Aufgabe erst einmal nur von Weiß entfernen. Für die anderen Aufgaben sucht Schwarz sich ein Feld, auf dem sowohl horizontal als auch vertikal kein Feld von einer weißen Figur besetzt ist.

3.4.1 Zug Generator für Schwarz

Optimale Bewegungsrichtung Das finden der optimalen Bewegungsrichtung verhält sich recht simpel. Es wird überprüft ob die nächste weiße Figur auf der Vertikalen höher oder tiefer liegt. Liegt die weiße Figur nördlich, bewegt sich Schwarz nach Süden, und umgekehrt.

Finden eines sicheren Feldes Das finden eines sicheren Feldes ist dabei etwas komplizierter. Ausgehend von der Position der schwarzen Figur wird jedes Feld auf der horizontalen und vertikalen Achse geprüft, ob es leer ist.
Als nächstes wird geprüft ob das Feld sicher ist. Dazu überprüft er erst ob eines der Felder auf der selben Zeile bereits besetzt ist. Ist kein Feld besetzt wird überprüft, ob ein Feld mit dem selben Feld-Index belegt ist. Ist auch dies der Fall, gilt das Feld als sicher und wird in einer Liste gespeichert.
Aus dieser Liste wird hinterher durch Zufall entschieden, zu welchem sicheren Feld er sich bewegen soll.

Zufalls Zug Zum Generieren eines zufälligen Zuges kommen dieselben Regeln ins Spiel wie die, die bereits zum Platzieren der Figur verwendet wurden. Der einzige Unterschied ist das zusätzliche Berücksichtigen der optimalen Bewegungsrichtung.

3.4.2 Zug Generator für Weiß

Optimale Bewegungsrichtung Da die horizontale Achse bevorzugt wird, wird erst überprüft ob Schwarz westlich oder östlich von der eigenen Position liegt. Als nächstes wird überprüft ob es für weiß möglich ist, sich in die gewünschte Richtung zu bewegen. Ist dies nicht möglich wird überprüft ob Schwarz nördlich oder südlich steht.

3.5 Lösung zu Aufgabe 1

3.5.1 Lösungsweg

> *Acht weiße Bauern können den schwarzen Turm fangen, egal, wie sich dieser bewegt. Beschreibe, wie sie dabei vorgehen sollen. Beschreibe auch, wie der Turm sich bewegen kann, um so lange wie möglich nicht gefangen zu werden. Implementiere die Vorgehensweisen von Bauern und Turm und visualisiere die „Treibjagd“.*

Die Lösung zu Aufgabe 1 ist simpel gelöst. Die 8 Bauern werden mittig entweder horizontal oder vertikal aufgereiht platziert. Da nun keine Lücke zwischen den Bauern ist, durch die der Turm verschwinden könnte, können sich die Bauern einfach jeweils abwechselnd einen Schritt in Richtung Schwarz bewegen, bis dieser komplett eingekreist ist.

3.5.2 Ergebnis

Angenommen Schwarz zieht sich bei der ersten Gelegenheit zum Rand des Brettes zurück, benötigt Weiß 25 Züge um Schwarz sicher zu fangen.

3.6 Lösung zu Aufgabe 2

3.6.1 Lösungsweg

> *Untersuche, ob sieben weiße Bauern den schwarzen Turm fangen können. Falls ja, dann beschreibe, wie die Bauern vorgehen sollen. Falls nein, dann beschreibe, wie sich der Turm verhalten soll, um nie gefangen zu werden.*
> *Implementiere das Vorgehen der deiner Meinung nach erfolgreichen Spielpartei und ein möglichst gutes Vorgehen der anderen Spielpartei. Visualisiere das Spiel zwischen den beiden Kontrahenten.*

Für Aufgabe 2 wurde bei der Vorbereitung des Spielbretts eine Änderung vorgenommen. Da nun nur 7 Bauern im Spiel sind bleibt immer ein Feld in der Formation offen. Deswegen wird bei der Platzierung der Figuren bereits durch Zufall entschieden welches Feld denn nun frei bleibt.

Die weißen Figuren handeln nun, wie oben im Zug Generator für Weiß beschrieben. Steht Schwarz einer weißen Figur gegenüber, bewegt sich diese nach vorne. Steht Schwarz nun auf einem sicheren Feld, also wie in dem Fall gegenüber des freigelassenen Feldes zwischen den Bauern, bewegt sich einer der Bauern auf das freie Feld.

3.6.2 Ergebnis

In Aufgabe 2 ließ sich der schwarze Turm nicht von Weiß fangen. Durch das fehlende Feld in der Reihe bewegt sich Schwarz immer wieder auf Position des gerade freigewordenen Feldes. Weiß kann die Lücke nicht schließen. Selbst wenn Weiß versucht sich nach vorne zu bewegen, kann Schwarz auf die andere Seite des Spielfelds durchziehen und entkommen.

3.7 Lösung zu Aufgabe 3

3.7.1 Lösungsweg

> *Betrachte jetzt eine wesentlich allgemeinere Situation: Es gibt k weiße Bauern, von denen sich in jedem Zug l jeweils einen Schritt bewegen. Für welche k und l können die weißen Bauern den schwarzen Turm wohl immer fangen? Was kannst du darüber herausfinden? Fange mit k = 7 an.*

Für diese Aufgabe 3 ist die Ausgangsposition der Bauern entscheidend. Der schwarze Turm bewegt sich weiterhin zur nächsten sicheren Position, wie in dem Zug Generator für Schwarz beschrieben. Deshalb kann eine horizontale Aufstellung mit 7 Bauern nicht funktionieren, egal wie viele Bauern sich gleichzeitig bewegen können.

Die einzige Möglichkeit Schwarz zu fangen ist Weiß diagonal aufzustellen und das leere Feld an eine der Seiten zu legen. Eine weitere Bedingung ist das sich alle Bauern gleichzeitig bewegen können. Weiß muss sich nun mit allen Bauern gleichzeitig seitwärts in Richtung Turm bewegen. Somit wird nach dem ersten Weißen Zug die Lücke in der Reihe geschlossen.

3.7.2 Ergebnis

Bei dieser Aufgabe gibt es die ersten Unterschiede zwischen den Startpositionen. Startet das Spiel mit weniger als 8 Figuren in einer horizontalen Linie, bleibt das Ergebnis unverändert. Schwarz kann immer noch nicht gefangen werden.

Bei 8 oder mehr Figuren in einer horizontalen Linie braucht Weiß nur maximal 4 Züge um den Turm zu fangen, sobald sich alle 8 Figuren gleichzeitig bewegen können. Desto mehr Bauern dazu kommen des so weniger Züge werden natürlich gebraucht.

Stellt sich Weiß nun diagonal mit 8 Bauern auf, gibt es für Schwarz ebenfalls kein entkommen. Bei 7 Bauern, mit der Bedingung das sich alle gleichzeitig bewegen dürfen, lässt sich Schwarz ebenfalls garantiert fangen.
Bei weniger als 7 Bauern ist es nicht möglich Schwarz zu fangen.

3.8 Lösung zu Aufgabe 4

3.8.1 Lösungsweg

> *Untersuche eine weitere Variante des Spiels, zum Beispiel: Was ändert sich, falls der schwarze Turm durch eine schwarze Dame ersetzt wird? Sie darf zusätzlich diagonal ziehen*

Weiß muss horizontal mit 8 Figuren aufgestellt werden. Die diagonale Herangehensweise funktioniert hier wahrscheinlich nicht, da die Dame durch die Formation durch ziehen kann.

3.8.2 Ergebnis

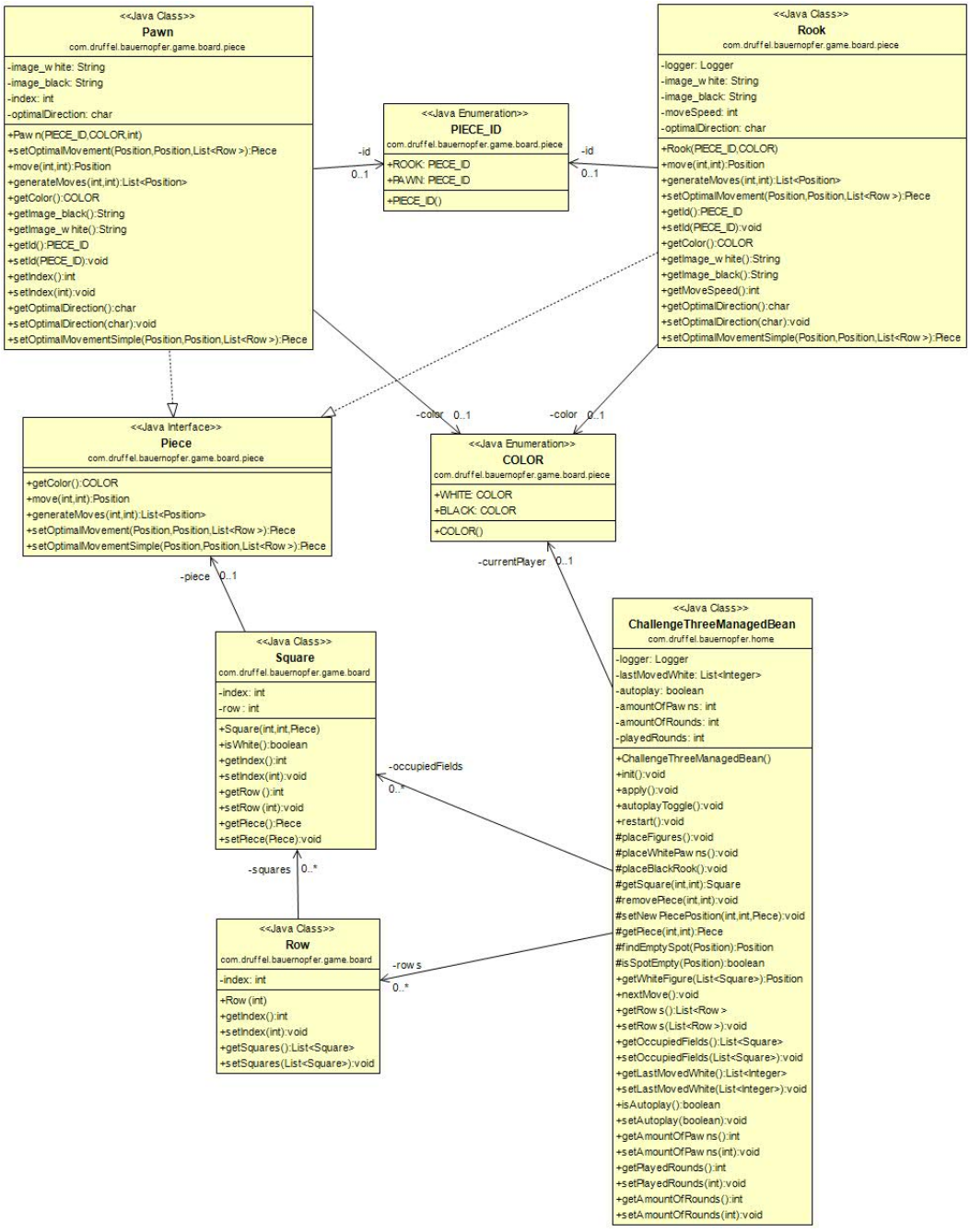
Zu Aufgabe 4 gibt es leider keine Lösung, da dieser Teil der Aufgabe aus zeitlichen Gründen nicht mehr gelöst werden konnte.

3.9 Planung des Programms

3.9.1 Klassendiagramm

Diagramm

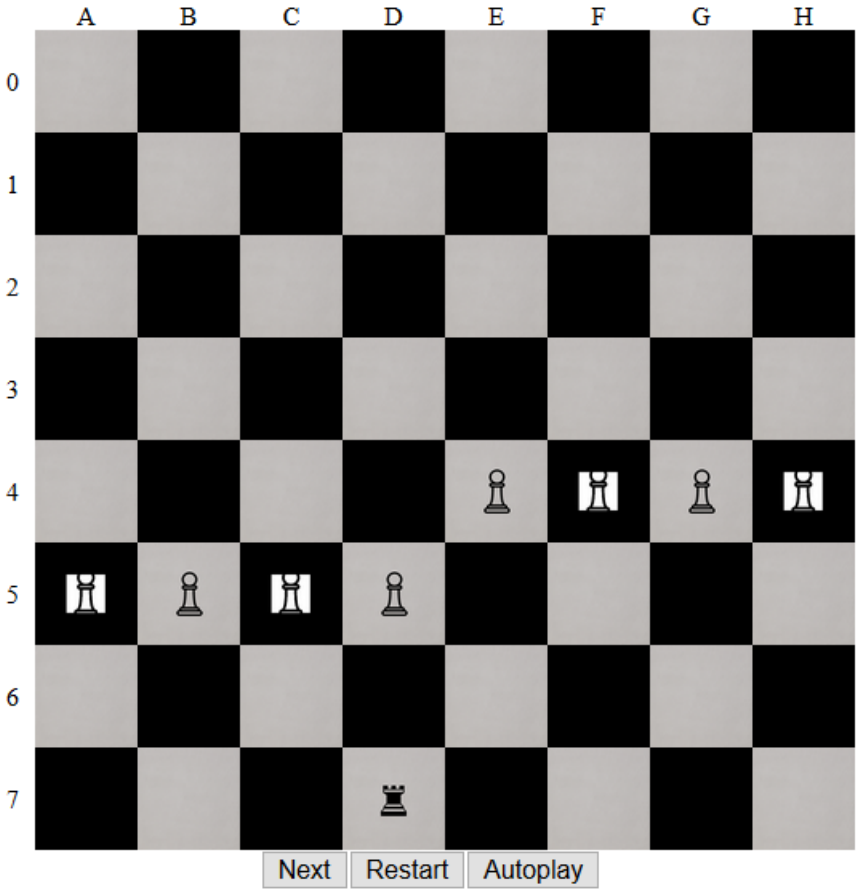
Einfaches Klassendiagramm des Projektes.



Beispiele

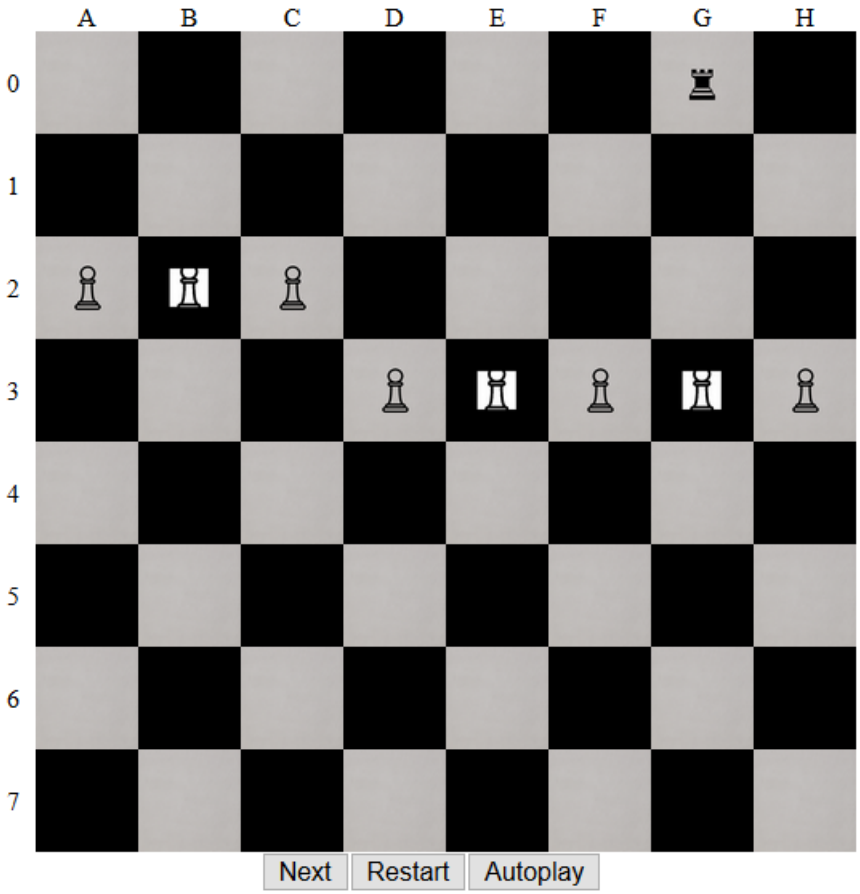
4.1 Aufgabe 1

4.1.1 Turm südlich



Bauern bewegen sich nacheinander nach Süden bis der Turm eingekreist ist.

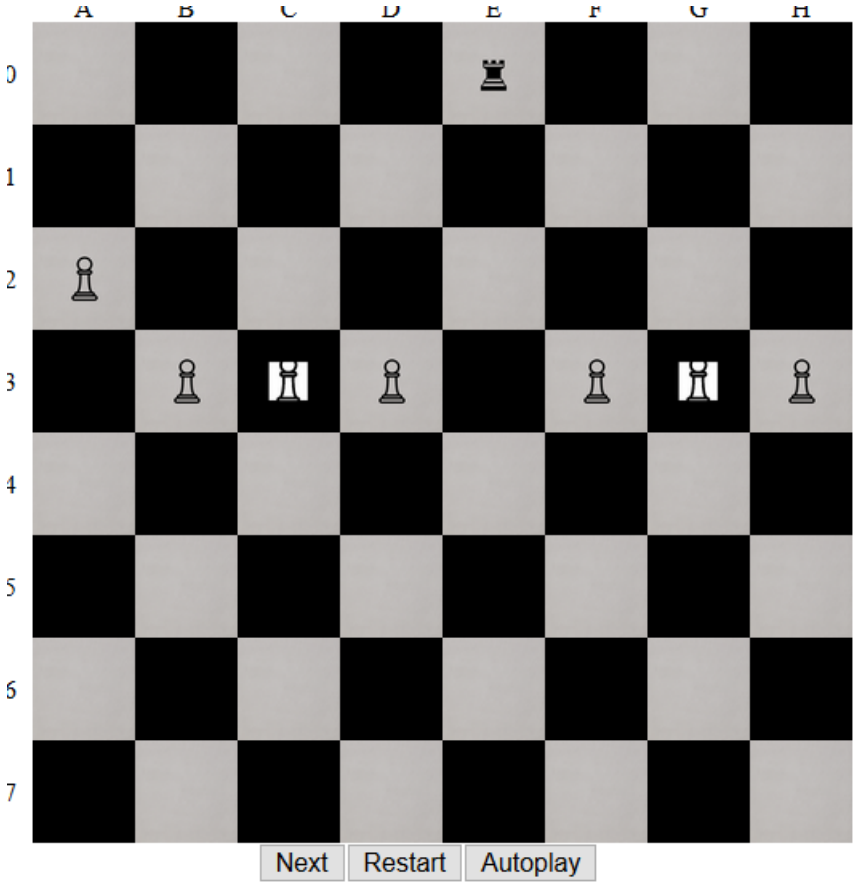
4.1.2 Turm nördlich



Bauern bewegen sich nacheinander nach oben bis der Turm eingekreist ist.

4.2 Aufgabe 2

Turm findet den safe spot



Quellcode

5.1 Safe Spot

```
protected Position findEmptySpot(Position currentPos)
    {
        List<Position> emptySpots = new ArrayList<>();

        for (int i = currentPos.getX() + 1; i < 8; i++)
        {
            if (isSpotEmpty(new Position(i, currentPos.getY())))
            {
                emptySpots.add(new Position(i, currentPos.getY()));
            }
            else if (Movement.isFieldOccupiedByWhite(new Position(i, currentPos.getY()),
                rows))
            {
                break;
            }
        }

        for (int i = currentPos.getX() - 1; i > -1; i--)
        {
            if (isSpotEmpty(new Position(i, currentPos.getY())))
            {
                emptySpots.add(new Position(i, currentPos.getY()));
            }
            else if (Movement.isFieldOccupiedByWhite(new Position(i, currentPos.getY()),
                rows))
            {
                break;
            }
        }

        for (int i = currentPos.getY() + 1; i < 8; i++)
        {
            if (isSpotEmpty(new Position(currentPos.getX(), i)))
            {
                emptySpots.add(new Position(currentPos.getX(), i));
            }
            else if (Movement.isFieldOccupiedByWhite(new Position(currentPos.getX(), i),
                rows))
```

```
            {
                break;
            }
        }

        for (int i = currentPos.getY() - 1; i > -1; i--)
        {
            if (isSpotEmpty(new Position(currentPos.getX(), i)))
            {
                emptySpots.add(new Position(currentPos.getX(), i));
            }
            else if (Movement.isFieldOccupiedByWhite(new Position(currentPos.getX(), i),
                rows))
            {
                break;
            }
        }

        return (emptySpots.size() == 0)
                ? getPiece(BoardUtil.getBlackFigure(occupiedFields).getY(),
                    BoardUtil.getBlackFigure(occupiedFields).getX())
                        .setOptimalMovement(getWhiteFigure(occupiedFields),
                            BoardUtil.getBlackFigure(occupiedFields), rows)
                        .move(BoardUtil.getBlackFigure(occupiedFields).getY(),
                            BoardUtil.getBlackFigure(occupiedFields).getX())
                : emptySpots.get(new Random().nextInt(emptySpots.size()));
    }
```

```
protected boolean isSpotEmpty(Position target)
{
    boolean empty = true;

    for (int i = target.getX(); i < 8; i++)
    {
        if (Movement.isFieldOccupiedByWhite(new Position(i, target.getY()), rows))
        {
            System.out.println(getPiece(target.getY(), i));
            empty = false;
        }
    }

    for (int i = target.getY() + 1; i < 8; i++)
    {
        if (Movement.isFieldOccupiedByWhite(new Position(target.getX(), i), rows))
        {
            empty = false;
        }

    }

    for (int i = target.getY() - 1; i > -1; i--)
    {
        if (Movement.isFieldOccupiedByWhite(new Position(target.getX(), i), rows))
        {
            empty = false;
        }

    }

    return empty;
}
```

5.2 Nächster Zug

```
public void nextMove()
   {

       // BLACK PLAYER
       if (currentPlayer == COLOR.BLACK)
       {
           FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Black",
               "Black player is thinking."));
           Position oldPosition = BoardUtil.getBlackFigure(occupiedFields);
           if (oldPosition != null)
           {
               Piece piece = getPiece(oldPosition.getY(), oldPosition.getX());
               Position newPosition = findEmptySpot(oldPosition);
               int i = 0;
               while (Movement.isFieldOccupied(newPosition, rows) ||
                   Movement.fieldNextToWhite(newPosition, rows))
               {
                   i++;
                   newPosition = getSquare(oldPosition.getY(),
                       oldPosition.getX()).getPiece().move(oldPosition.getY(),
                       oldPosition.getX());
                   if(i==20)
                   {
                       restart();
                       autoplay = false;
                   }
               }
               removePiece(oldPosition.getY(), oldPosition.getX());
               setNewPiecePosition(newPosition.getY(), newPosition.getX(), piece);
               occupiedFields.add(getSquare(newPosition.getY(), newPosition.getX()));
               currentPlayer = COLOR.WHITE;
               logger.info("BLACK moved from Row:" + oldPosition.getY() + " Square:" +
                   oldPosition.getX() + "to Row:" + newPosition.getY() + " Square:"
                       + newPosition.getX());
           }
       }
       // WHITE PLAYER
       else
       {
           FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("White",
               "White player is thinking."));
           int i = 0;
           while(i < amountOfRounds)
           {

           Position oldPosition = getWhiteFigure(occupiedFields);
           if (oldPosition != null)
           {
               Piece piece = getPiece(oldPosition.getY(), oldPosition.getX());
               Position newPosition = getSquare(oldPosition.getY(),
                   oldPosition.getX()).getPiece()
```

```
                .setOptimalMovement(BoardUtil.getBlackFigure(occupiedFields),
                    oldPosition, rows).move(oldPosition.getY(), oldPosition.getX());
            while (Movement.isFieldOccupiedByWhite(newPosition, rows) &&
                Movement.cannotCapture(newPosition, rows))
            {
                newPosition = getSquare(oldPosition.getY(),
                    oldPosition.getX()).getPiece().move(oldPosition.getY(),
                    oldPosition.getX());
            }
            removePiece(oldPosition.getY(), oldPosition.getX());
            setNewPiecePosition(newPosition.getY(), newPosition.getX(), piece);
            occupiedFields.add(getSquare(newPosition.getY(), newPosition.getX()));
            currentPlayer = COLOR.BLACK;
            logger.info("WHITE moved from Row:" + oldPosition.getY() + " Square:" +
                oldPosition.getX() + "to Row:" + newPosition.getY() + " Square:"
                   + newPosition.getX());
            if (lastMovedWhite.size() == 7)
            {
                lastMovedWhite = new ArrayList<>();
            }
            Pawn p = (Pawn) piece;
            lastMovedWhite.add(p.getIndex());
        }
        i++;
    }
    }

    if (BoardUtil.isGameOver(occupiedFields))
    {
        autoplay = false;
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Game end",
            "Game came to an End."));
    }

}
```

Der Algorithmus für das Ziehen unterscheidet sich für Schwarz und Weiß nur gering. Nachdem die Position der Figur ermittelt wurde wird diese in der Variable oldPosition zwischen gespeichert. Als nächstes wird die move() Methode der Figur auf dieser Position ausgeführt, und die neue Position in der Variable newPosition gespeichert.
Als nächstes wird die alte Position gelöscht und die neue gesetzt. Zusätzlich wird das Feld mit der neuen Position in der Liste occupiedFields gespeichert.

5.3 Figuren auf dem Brett platzieren

```
protected void placeWhitePawns()
   {
       logger.info("Placing white pawns");
       int row = 4;
       for (int i = 0; i < amountOfPawns; i++)
       {

           if (i < 8)
           {
               Pawn pawn = new Pawn(PIECE_ID.PAWN, COLOR.WHITE, i);
               Position pos = new Position(i, row);
               rows.get(pos.getY()).getSquares().get(pos.getX()).setPiece(pawn);
               occupiedFields.add(rows.get(pos.getY()).getSquares().get(pos.getX()));
               logger.info("Placed pawn at: X:" + pos.getX() + " Y:" + pos.getY());
           }
           else
           {
               Pawn pawn = new Pawn(PIECE_ID.PAWN, COLOR.WHITE, i);
               Position pos = new Position(i%8, new Random().nextInt(8));
               rows.get(pos.getY()).getSquares().get(pos.getX()).setPiece(pawn);
               occupiedFields.add(rows.get(pos.getY()).getSquares().get(pos.getX()));
               logger.info("Placed pawn at: X:" + pos.getX() + " Y:" + pos.getY());
           }
       }
   }

   /**
    * Placing all black figures on the board, on a random position
    */
   protected void placeBlackRook()
   {
       logger.info("Placing Black rook");
       Rook rook = new Rook(PIECE_ID.ROOK, COLOR.BLACK);

       Position newPosition = new Position(new Random().nextInt(8), new Random().nextInt(8));
       while (Movement.isFieldOccupied(newPosition, rows) ||
           Movement.fieldNextToWhite(newPosition, rows))
       {
           newPosition = new Position(new Random().nextInt(8), new Random().nextInt(8));
       }

       rows.get(newPosition.getY()).getSquares().get(newPosition.getX()).setPiece(rook);
       occupiedFields.add(rows.get(newPosition.getY()).getSquares().get(newPosition.getX()));
       logger.info("Placed rook at: X:" + newPosition.getX() + " Y:" + newPosition.getY());
   }
```