Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
18. Dezember 2017

Lösungsvorschlag zur 9. Übung zur Vorlesung

Einführung in die Programmierung

A9-1 *Catch & Throw* Gegeben sei folgendes (nicht sehr sinnvolles) Java-Programm:

```
class PayloadA extends Throwable {
  private double x;
  PayloadA(double x)  { this.x = x; }
  public double get() { return x;   }
}

class PayloadB extends Throwable {
  private int x;
  PayloadB(int x)     { this.x = x; }
  public int get()    { return x;   }
}

class ThrowAndCatch {
  public static void main(String[] args) {
    try {
      f();
    } catch (PayloadB p) {
      System.out.println("Got payload B " + p.get());
  }  }

  static void f() throws PayloadB {
    try {
      g();
    } finally {
      System.out.println("Greetings from f's finally");
      throw new PayloadB(42);
  } }


  static void g() throws PayloadA {
    try {
      throw new PayloadA(4711);
    } finally {
      System.out.println("Greetings from g's finally");
} } }
```

a) Erklären Sie genau, wie das Programm abläuft. Machen Sie dies, ohne das Programm auszuführen! Welche Ausgabe wird produziert?

LÖSUNGSVORSCHLAG:

```
> java ThrowAndCatch
Greetings from g's finally
Greetings from f's finally
Got payload B 42
```

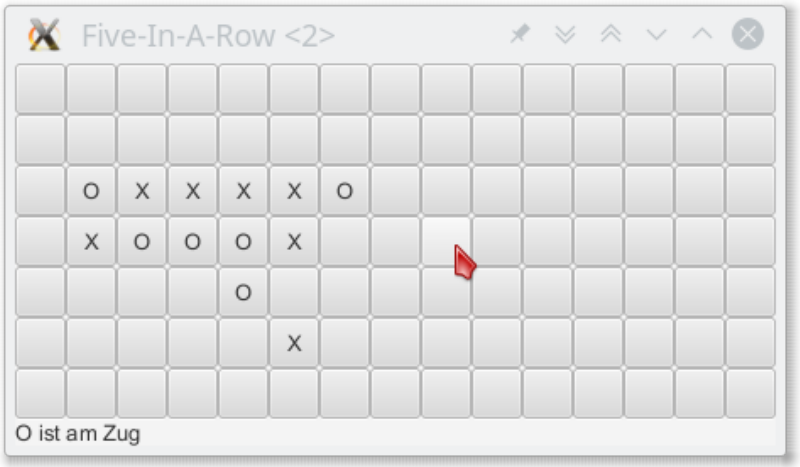
b) Warum muss in der Signatur von Methode `f` kein `throws PayloadA` deklariert werden, obwohl die darin aufgerufene Methode `g` sehr wohl ein `PayloadA`-Objekt werfen kann?

LÖSUNGSVORSCHLAG:

Egal ob `g` ein `PayloadA`-Objekt wirft, betritt `f` den `finally`-Block und wirft dort eine neue Ausnahme mit einem `PayloadB`-Objekt. Funktion `f` wird also nie durch Wurf von `PayloadA` verlassen.

Wenn man im `finally`-Block von Methode `f` die Zeile `throw new PayloadB(42);` auskommentiert, dann beschwert sich der Compiler, über die fehlende Deklaration – schließlich wird mit dem
verb—finally—-Block eine bestehende Ausnahme nicht abgefangen, sondern nur temporär unterbrochen. Im ursprünglichen Fall wird die bestehende Ausnahme `PayloadA` durch das neue Werfen des `PayloadB`-Objekt jedoch übergangen (es gilt immer das letzte `throw`).

A9-2 ***Fünf in einer Reihe*** Implementieren Sie eine kleine JavaFX-Applikation, mit der man das Spiel "Fünf in einer Reihe" spielen kann. Zwei Spieler, X und O, setzen abwechselnd, beginnend mit X, auf ein rechteckiges Spielfeld beliebiger Größe (meist 15x15 oder 19x19, wir wollen hier aber beliebige Rechtecke erlauben, deren Kanten größer als 5 sind). Der Spieler, der zuerst 5 horizontal, vertikal, oder diagonal zusammenhängende Steine gesetzt hat, hat gewonnen.



Unser Fenster besteht aus einer `GridPane` (wer möchte auch `FlowPlane` oder `BorderPane`), welche zwei Element übereinander anordnet. Das obere stellt das Spielfeld dar, während das untere Element ein einfaches `Label` ist, welches den Spielstatus anzeigen soll.

Das Spielfeld ist erneut eine `GridPane`, in der wir jede einzelne Zelle direkt durch Objekte der Klasse `javafx.scene.control.Button` darstellen. Dies ist zwar nicht wirklich hübsch, aber für unseren derzeitigen Kenntnisstand besonders einfach:

- Die Methode `void setText(String text)` erlaubt es uns, einen Text wie X oder `O` auf dem Knopf darzustellen.
- Die Methoden `void setMinWidth(double value)`, `void setMinHeight(double value)`, `void setMaxWidth(double value)`, `void setMaxHeight(double value)` erlauben es uns, die Größe der Knöpfe zu beeinflussen. Wir setzen diese einfach auf einen hinreichend großen Wert, damit alle Knöpfe gleich groß dargestellt werden.
- Die Methode `void setOnAction(EventHandler<ActionEvent> value)` erlaubt es uns, auf einen Druck des Knopfes zu reagieren. Am einfachsten machen wir dies mit einem Lambda-Ausdruck:

```
int x; int y; GridPane pane;
...
Button button = new Button();
pane.add(button,x,y);
button.setOnAction(event -> { System.out.println("Button("+x+","+y")"); });
```

  Die Text Ausgabe in diesem Beispiel ist natürlich durch sinnvollen Code zu ersetzen. Das Beispiel zeigt aber, dass der Behandler sein Argument `event` hier gar nicht beachten braucht, wenn wir die Position gleich beim Erstellen des Knopfes fest in den Behandler hineinschreiben.

Bei der Implementierungen folgen wir erneut dem MVC-Pattern. Wir können auch viele Teile aus dem zuvor behandelten Beispiel „Game of Life“ wiederverwenden, da wir erneut ein 2D-Spielfeld mit quadratischen Zellen haben. Allerdings ist es bei einer grafischen Benutzeroberfläche oft etwas schwierig, Controller und View ordentlich zu trennen.

Wir empfehlen hier, zuerst mit dem Modell anzufangen. Dieses sollte von beiden unabhängig sein und alle benötigten Methoden zur Verfügung stellen. Einige Klassen des Modells sollten vermutlich Unterklassen von `Observable` sein, und an geeigneten Stellen die geerbten Methoden `setChanged()` und `notifyObservers()` aufzurufen, damit sich die View bei Bedarf selbst aktualisiert.

Der Konstruktor der View erstellt die Elemente der Szene, setzt Ereignis-Behandler auf entsprechenden Aufrufe im Modell und hängt umgekehrt die Beobachter des Modells ein.

a) Überlegen Sie sich zuerst, welche Klassen sie erstellen sollten und welche Funktionalität diese bieten sollten.

b) Vergleichen Sie Ihren Entwurf mit der beiliegenden Dateivorlage. Unsere Dateivorlage ist nur ein Vorschlag! Trauen Sie sich ruhig, eigene Wege zu gehen!

c) Vervollständigen Sie nun Ihre Implementierung in folgenden Schritten:

   i) Initalisierung eines Spielfeldes mit fester Größe (einstellbar über eine Konstante, wie im Game of Life).

   ii) Ein Drücken eines Knopfes ändert des Beschriftung zu X.

   iii) Abwechselnd werden X und `O` gesetzt. Die Statuszeile zeigt an, wer dran ist.

iv) Das Drücken von bereits besetzten Spielfeldern wird ignoriert.

v) Nach jedem Zug wird die Gewinnbedingung geprüft und ggf. in der Statuszeile angezeigt.

vi) Ein Klick auf einem beliebigen Feld startet das Spiel neu, falls dieses beendet war. Das Spielfeld soll nun wieder leer sein, und $\mathcal{X}$ ist am Zug.

vii) Eingabe von Breite und Höhe des Spielfelds über JavaFX vor Spielbeginn (z.B. über `javafx.scene.control.TextInputDialog` oder über Elemente im Hauptfenster).

Hinweis: Bitte dazu nicht den Eingabe-Dialog von Folie 4.57ff verwenden, da dieser ein anderes GUI Framework verwendete und es sich nicht empfiehlt, mehrere Frameworks zu mischen.

LÖSUNGSVORSCHLAG:
Wir folgen in diesem Lösungsvorschlag der Vorlage mit einem minimalen Controller; andere akzeptable Aufteilungen von View und Controller sind durchaus möglich, insbesondere bei dem Einsatz von FXML für die View (via SceneBuilder).

Der Präsenzlösung zu dieser Übung sollten die entsprechenden Dateien beiliegen.

Wichtig ist vor allem die Abtrennung des Modells, hier die Klassen `FiveCell`, `FiveBoard` und `FiveGame`. Die Klasse `Position` könnt man auch noch dazu zählen oder auch nur als allgemeine Basis-Klasse auffassen.

Faustregel: Die Klassen des Modells sollten keine `javafx` Importe aufweisen! Diese Faustregel haben wir hier befolgt. Dies hat den Vorteil, dass das Modell leicht wiederverwendbar ist, z.B. wenn man eine andere Darstellung (z.B. Swing, Konsole, Browser) hinzufügen möchte. Weiterhin kann man das Modell und die enthaltene "Geschäftslogik" (Logik des Spiels, d.h. wie ein Spielzug ausgewertet wird, Gewinnbedingungen, etc.) unabhängig vom Rest des Projektes betrachten und testen, was die Entwicklung und Wartung erheblich vereinfacht.

`FiveInARow.java` Hauptklasse zum Starten des Spiels mit einer voreingestellten Größe, ohne Besonderheiten.

`FiveInARowFXML.java` Alternative Hauptklasse zum Starten des Spiels mit Abfrage der gewünschten Spielfeldgröße zur Lösung der optionalen Teilaufgabe c)vii.

Den Eingabe-Dialog für die Spielfeldgröße haben wir hier zur Demonstration mit dem SceneBuilder erstellt.

`FiveCell.java` Die Methode `setOwner` überprüft hier selbst, ob hier ein Zug gültig ist. Falls der Zug ausgeführt werden konnte, so wird dies mit `true` signalisiert. Im Falle eine Änderung müssen hier eventuelle Beobachter benachrichtigt werden.

```
public boolean setOwner (int owner) {
  if (isEmpty()){
      this.owner = owner;
      setChanged();
      notifyObservers();
      return true;
  } else {
      return false;
}  }

public void resetOwner () { // setOwner cannot be used for resetting.
  this.owner = EMPTY;
  setChanged();
  notifyObservers();
}
```

`FiveBoard.java` Die Methode `getCells` ähnlich zu `Spielfeld.alleZellen` im Game of Life. Die Methode `hasEmptyCells` geht einfach nur alle Zellen durch und bricht ab, sobald eine leere Zelle gefunden wurde.

```
public ArrayList<FiveCell> getCells() {
  ArrayList<FiveCell> result = new ArrayList<>(max_x*max_y);
  for (int x=0; x< max_x; x++) {
      for (int y = 0; y < max_y; y++) {
          result.add(board[x][y]);
  }   }
  return  result;
}
public boolean hasEmptyCells() {
  for (int x=0; x < max_x; x++) {
      for (int y = 0; y < max_y; y++) {
          if (board[x][y].isEmpty()) return true;
      }
  }
  return false;
}
```

Schwieriger ist die Überprüfung der Siegesbedingung in `isWinningMove`. Hier könnte man vier gewöhnliche Schleifen einsetzen. Damit es etwas spannender ist und wir Redundanz im Code vermeiden, setzen wir hier das funktionale Interface `Direction` ein: die Methode `dominion` klettert solange in die gegebene Richtung weiter, wie die Zellen vom gegebenen Spieler besetzt sind. Wir klettern zwei Mal in entgegen gesetzte Richtungen von der gegebenen Position und berechnen anschließend die Entfernung:

```
public boolean isWinningMove(Position pos) {
    FiveCell actcell = getCell(pos);
    int player       = actcell.getOwner();

    FiveCell check;
    // Check Row
    Position minPos = dominion(player,pos, p -> p.left());
    Position maxPos = dominion(player,pos, p -> p.right());
    if (maxPos.getX()-minPos.getX() > 5) return true;
    // Check Column
    minPos = dominion(player,pos, p -> p.up());
    maxPos = dominion(player,pos, p -> p.down());
    if (maxPos.getY()-minPos.getY() > 5) return true;
    // Check Diag1
    minPos = dominion(player,pos, p -> p.up().left());
    maxPos = dominion(player,pos, p -> p.down().right());
    if (maxPos.getX()-minPos.getX() > 5) return true;
    // Check Diag2
    minPos = dominion(player,pos, p -> p.down().left());
    maxPos = dominion(player,pos, p -> p.up().right());
    if (maxPos.getX()-minPos.getX() > 5) return true;
    // No more possibilites
    return  false;
}
public Position dominion(int player, Position pos, Direction dir) {
    Position curPos = pos;
    FiveCell check;
    do {
        curPos = dir.goFrom(curPos);
        check  = getCell(curPos);
    } while (check != null && check.getOwner() == player);
    return curPos;
}
```

`FiveGame.java` Einen Spielzug reichen wir einfach an die entsprechende Zelle weiter. Da sich der Spielstatus ggf. ändert, müssen wir noch Beobachter informieren.

```
import java.util.Observable;
/**  Modell */
public class FiveGame extends Observable {

    public int height;
    public int width;

    private FiveBoard board;
    private int activePlayer = FiveCell.EMPTY;
    private boolean gameOngoing = false;

    public FiveGame() {
        this(15,7);
    }

    public FiveGame(int width, int height) {
        this.height = height;
        this.width  = width;
        this.board  = new FiveBoard(width,height);
        reset();
    }

    public void reset() {
        for (FiveCell cell : this.board.getCells()){
            cell.resetOwner();
        }
        this.gameOngoing = true;
        this.activePlayer = FiveCell.PLAYER_X;
        setChanged();
        notifyObservers();
    }


    public void moveAt(Position pos) {
        if (gameOngoing) {
            if (board.getCell(pos).setOwner(activePlayer)) {
                if (board.isWinningMove(pos)) {
                    gameOngoing = false;
                } else {
                    if (board.hasEmptyCells()) {
                        nextPlayer();
                    } else {
                        activePlayer = FiveCell.EMPTY;
                        gameOngoing = false;
                    }
                }
                setChanged();
                notifyObservers();
            }
        } else {
            reset();
        }
    }

    public void nextPlayer() {
        if (activePlayer == FiveCell.PLAYER_O) {
            activePlayer = FiveCell.PLAYER_X;
        } else {
            activePlayer = FiveCell.PLAYER_O;
        }
    }

    public FiveBoard getBoard() {
        return board;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public boolean isGameOngoing() {
        return gameOngoing;
    }

    public int getActivePlayer() {
        return activePlayer;
    }
}
```

`FiveView.java` Hier können wir es uns einfach machen und brauchen keine Instanzvariable, um uns die Referenz auf das Spiel zu merken. Stattdessen "verdrahten" wir gleich alles zu Beginn: Beim Erstellen eines Knopfes kennen wir dessen Position und können damit direkt einen Ereignis-Behandler erstellen, der die `moveAt`-Methode aufruft. Ein Beobachter für die korrespondierende Zelle des Spiels aktualisiert jeweils eigenständig den Text des Knopfes.

```
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;

/**
 * View
 */
public class FiveView extends GridPane {

    public static final int scale = 30;

    public FiveView(FiveGame game) {
        super();
        // Board
        GridPane boardView = new GridPane();
        for (FiveCell cell : game.getBoard().getCells()) {
            Position pos = cell.getPosition();
            Button btn = new Button(showPlayer(cell.getOwner()));
            btn.setMinWidth(scale);
            btn.setMinHeight(scale);
            btn.setMaxWidth(Double.MAX_VALUE);
            btn.setMaxHeight(Double.MAX_VALUE);
            btn.setOnAction(event -> {
                game.moveAt(pos);
            });
            cell.addObserver((o,arg) -> {
                btn.setText(showPlayer(cell.getOwner()));
            });
            boardView.add(btn, pos.getX(), pos.getY());
        }
        this.add(boardView,0,0);
        // Statuszeile
        Label status = new Label(showStatus(game.isGameOngoing(),game.getActivePlayer()));
        game.addObserver((o,arg) -> {
            status.setText(showStatus(game.isGameOngoing(),game.getActivePlayer()));
        });
        this.add(status,0,1);
    }

    public static String showPlayer(int zustand) {
        if (zustand == FiveCell.PLAYER_O) {
            return "O";
        } else if (zustand == FiveCell.PLAYER_X) {
            return "X";
        } else return "";
    }

    public static String showWinner(int zustand) {
        if (zustand == FiveCell.EMPTY)
            return "Kein Spieler";
        else
            return showPlayer(zustand);
    }

    public static String showStatus(boolean ongoing, int zustand) {
        if (ongoing) {
            return showPlayer(zustand) + " ist am Zug";
        } else {
            return "Spiel beendet. " + showPlayer(zustand) + " hat gewonnen!";
        }
    }

}
```

H9-1 ***Ausnahmen*** (4 Punkte; Alle `.java`-Dateien Ihrer Lösung abgeben)

a) Schreiben Sie eine Klasse `Neighbor` mit Prozedur `boolean neighbor(Object[] arr)`, die prüft, ob in einem `Object[]`-Array zwei benachbarte Elemente gleich sind. Die Prozedur soll also genau dann `true` zurückgeben, wenn `arr[i].equals(arr[i+1])` für wenigstens ein `i` gilt. Die `main`-Methode testet `neighbor` an den Kommandozeilenargumenten.

```
public class Neighbor {

  public static boolean neighbor(Object[] arr) {
  }

  public static void main (String[] args) {
    if (neighbor(args))
         System.out.println("Neighbor(s) in arguments!");
    else System.out.println("No neighbors in arguments!");
  }
}
```

LÖSUNGSVORSCHLAG:

```
public class Neighbor {

  public static boolean neighbor(Object[] arr) {
    for (int i = 0; i < arr.length - 1; i++)
      if (arr[i].equals(arr[i + 1]))
        return true;
    return false;
  }

  public static void main (String[] args) {
    if (neighbor(args))
         System.out.println("Neighbor(s) in arguments!");
    else System.out.println("No neighbors in arguments!");
  }
}
```

b) Kopieren Sie `Neighbor` nach `NeighborWithException`, welches Sie wie folgt modifizieren: Schreiben Sie eine Ausnahme `ArrayTooShortException`, die ausgelöst werden soll, wenn das Array weniger als zwei Elemente enthält und insofern eine sinnvolle Prüfung auf benachbarte Elemente nicht möglich ist. Modifizieren Sie die Prozedur `neighbor` so, dass die Ausnahme ausgelöst wird, wenn das Eingabearray nicht lang genug ist. Ändern Sie `main`, so dass die Nachricht

```
Too few arguments to find neighbors!
```

ausgegeben wird, falls die Ausnahme ausgelöst wurde.

LÖSUNGSVORSCHLAG:

```
public class NeighborWithException {

  public static boolean neighbor(Object[] arr) throws ArrayTooShortException {
    if (arr.length < 2)
      throw new ArrayTooShortException();
    for (int i = 0; i < arr.length - 1; i++)
      if (arr[i].equals(arr[i + 1]))
        return true;
    return false;
  }

  public static void main (String[] args) {
    try {
      if (neighbor(args))
           System.out.println("Neighbor(s) in arguments!");
      else System.out.println("No neighbors in arguments!");
    } catch (ArrayTooShortException e) {
      System.out.println("Too few arguments to find neighbors!");
    }
  }
}

class ArrayTooShortException extends Exception {}
```

H9-2 ***Mine Sweeper*** (8 Punkte; Abgabe: `Minesweeper.java` als Hauptklasse, plus ggf. weitere Klassen)

In dieser Aufgabe sollen Sie eine (vereinfachte) Variante des Spiel "Minesweeper" mit Hilfe von JavaFX implementieren.

Das Spielfeld ist erneut ein rechteckiges Gitter aus quadratischen Zellen. In jeder Zelle kann eine Mine stecken. Jede nicht verminte Zelle enthält eine Ziffer, nämlich die Anzahl der Minen innerhalb ihrer acht Nachbarn (am Rand sind etwas natürlich weniger). Am Anfang wählt der Computer die Position der Minen zufällig und zeigt das Spielfeld verdeckt. Der Spieler kann nun durch Linksklicks eine Zelle aufdecken. Liegt darunter eine Mine, hat er verloren. Anderfalls wird die Zelle mit der entsprechenden Zahl angezeigt — die Anzeige der Ziffer Null wird aber unterdrückt. Ist die Zahl Null, deckt der Computer auch alle Nachbarn auf. Wird dabei eine weitere Null aufgedeckt, werden auch deren Nachbarn vom Computer wieder aufdeckt. Diese letzter Vorgang wird so lange wie möglich wiederholt. Danach ist

der Spieler wieder am Zug. Der Spieler gewinnt das Spiel, falls alle verdeckten Zellen nur noch Minen enthalten. Sobald der Spieler gewonnen oder verloren hat, soll das Programm eine entsprechende Meldung ausgeben und terminieren. Als Hilfe kann der Spieler verdeckte Zellen mit einem rechten Mausklick markieren und diese Markierung auf gleichem Wege auch wieder aufheben. Die Markierung kann der Spieler benutzen, um sich zu merken, dass eine Zelle vermint ist — die Markierung hat sonst keinen Einfluss auf den Spielverlauf. Beachten Sie das *Model-View-Control*-Entwurfsprinzip!

Das Fenster könnte wie folgt aussehen:

Dazu wurde die Zelle links unten mit der linken Maustaste angeklickt und die Zellen, die nun mit `P` markiert sind, mit der rechten Maustaste angeklickt.

Wichtig: Jede Klasse muss am Anfang einen Javadoc-Kommentar enthalten, welcher die Funktionsweise der Klasse in wenigen und einfachen Sätzen beschreibt!

Hinweise:

- Sie können große Teile des Lösungsvorschlags zu Aufgabe A9-2 wiederverwenden. Die Zellen können erneut durch `Button` dargestellt werden.

  Eine "aufgdeckte" Zelle kann einfach durch Deaktivierung des Buttons mit dem Aufruf `button.setDisabled(true);` realisiert werden.

  `setOnAction` reagiert nur auf Links-Klicks; stattdessen können Sie z.B. ähnlich wie hier vorgehen:

```
button.setOnMousePressed(event -> {
  if (event.isPrimaryButtonDown()) {
    System.out.println("Links-Klick");
  }
  if (event.isSecondaryButtonDown()) {
    System.out.println("Rechts-Klick");
```

```
    }
});
```

- Zufallszahlen können mit der Klasse `Random` bzw. mit deren Methode `nextInt` erzeugen.
- Im Gegensatz zu anderen Implementierungen müssen die Zahlen nicht farbig dargestellt werden, und die Flaggen und Minen können durch einen Text dargestellt werden.
- Für das Modell (im Sinne des MVC-Prinzips) kann es sinnvoll sein, die Zahlen der benachbarten Minen im Vorfeld zu berechnen und für jede Zelle einen Eintrag mit den relevanten Daten zu halten – inklusive der Information, ob sie verdeckt, markiert oder aufdeckt ist.
- Deklarieren Sie notwendige Konstanten wie die Größe des Spielfeldes (im Beispiel 20 auf 10) und die Anzahl der Minen (im Beispiel 10) explizit mit `static final`. *Ausnahme:* Die Werte werden über ein Eingabefenster, Konsole oder Kommandozeilenargumente eingelesen. (Hat keinen Einfluss auf Punktwertung.)

LÖSUNGSVORSCHLAG:
Ein Lösungsvorschlag sollte diesem PDF beiliegen. Wir haben der Einfachheit halber unseren Lösungsvorschlag zur Aufgabe A9-2 hergenommen und minimal angepasst.

`MineCell` Eine Zelle in Minesweeper merkt sich deutlich mehr Zustände. Offensichtlich sind `mine`, `flagged` und `position`.

Die Instanzvariable `neighbor_mines` merkt sich die Anzahl der verminten Nachbarn — das könnte man durchaus jedes Mal dynamisch neu berechnen, doch dazu müsste man Zugriff auf das Spielbrett haben, so dass es hier einfacher ist, den Wert beim Platzieren einmal auszurechnen und abzuspeichern.

Die Instanzvariablen `hidden` und `pressed` unterscheiden, ob das Feld angezeigt werden soll, und ob es durch den Spieler mit Links angeklickt wurde. Diese Unterscheidung braucht man nur, wenn man bei Spielende dem Spieler das gesamte Spielfeld offenbaren möchte und der Spieler dennoch erkennen können soll, was während des Spiels angeklickt worden war und was nicht. Diese Anzeige war aber nicht gefordert gewesen.

`Minefield` Am Spielbrett sind nur unwesentliche Vereinfachungen durchzuführen.

`MineGame` bietet zwei Methoden `explore` zum Aufdecken eines Feldes und `toggleMark` zum Markieren eines Feldes mit einer Flagge.

Die Methode `getStatus` liefert den Status des Spiels zurück. Da dies mehr als ein einzelner Wert ist, wurde eine neue Klasse `MinefieldStatus` erstellt, welche lediglich als Ergebnistyp dieser Methode dient, um mehr als einen Wert zurückzugeben.

Zusätzlich wurde eine Methode `reset` hinzugefügt, um ein Spiel zurückzusetzen und die Minen neu zu verteilen. Dies war jedoch nicht gefordert gewesen.

`MineView` Hier gibt es nur wenige Änderungen im Vergleich zu "Fünf-in-einer-Reihe". Wir erstellen wieder eine `GridPane` mit Knöpfen, wobei wir nun jedoch wie in der Aufgabenstellung angegeben zwischen Rechts- und Links-Klicks unterscheiden und jeweils eine andere Funktion von `MineGame` aufrufen. In beiden Fällen wird lediglich die Position übergeben. Die Abfrage von Keyboard-Eingaben war nicht gefragt gewesen und wurde hier nur der Vollständigkeit wegen hinzugefügt.

Die Berechnung des Inhaltes eines Knopfes wurde in eine eigene Methode `fieldContent` ausgelagert. Da bei Spielende alle Knöpfe durch `MineGame.explore` offenbart werden, wird hier entsprechend unterschieden, ob eine gesetzte Flagge korrekt war oder nicht. Dies war jedoch nicht gefordert gewesen.

Abgabe: Lösungen zu den Hausaufgaben können bis Sonntag, den **7.1.18**, mit UniWorX nur als `.zip` abgegeben werden. Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ws-2017-18/eip/`).