Dr. Steffen Jost
Stephan Barth

Ludwig-Maximilians-Universität München
Institut für Informatik
WS 2017/18

Softwareentwicklungspraktikum Nebenfach

Blatt 2

Abnahme erfolgt durch den Gruppenbetreuer in der 46. Kalenderwoche, d.h. 13.–17.11.17

Aufgabe 1 Implementieren Sie das Spiel „Minesweeper" mit Hilfe von JavaFX.

Minesweeper (= Minenräumer) ist ein einfaches Computerspiel. Für das Spielfeld an sich gilt

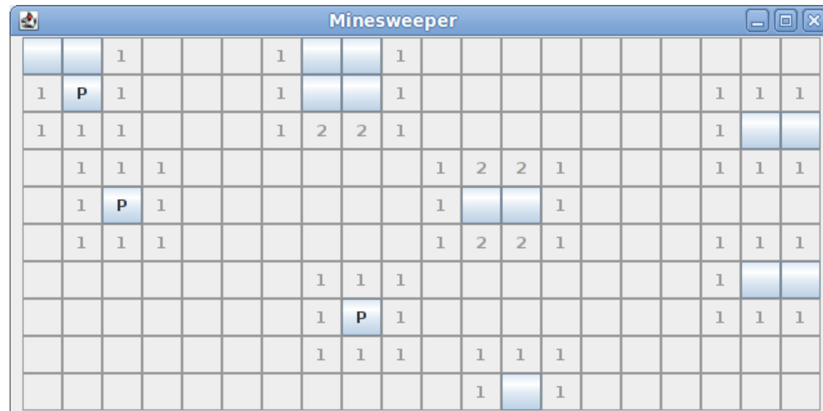
- Das Spielfeld ist ein rechteckiges Gitter aus quadratischen Zellen.
- In jeder Zelle kann eine Mine stecken.
- Jede nicht verminte Zelle enthält eine Ziffer, nämlich die Anzahl der Minen innerhalb ihrer acht Nachbarn (bzw. am Rand innerhalb der fünf oder drei Nachbarn).

Beim Starten des Spiels wählt der Computer die Position der Minen zufällig. Das Spielfeld wird komplett verdeckt angezeigt.

Im Spiel gelten dann folgende Regeln

- 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.
- Deckt man ein Feld ohne Mine auf, sieht man wie viele Minen auf den Nachbarfeldern liegen.
- Mit der rechten Maustaste kann man eine Flagge setzen, welche angibt, dass man unter diesem Feld eine Mine vermutet. Links-Klicks auf Flaggen werden ignoriert, Rechts-Klicks auf Flaggen entfernen diese wieder.
- Der Spieler gewinnt das Spiel, falls alle Zellen ohne Mine aufgedeckt sind.
- Sobald der Spieler gewonnen oder verloren hat, soll das Programm eine entsprechende Meldung ausgeben. Falls der Spieler verloren hat, sollen auch alle Minen angezeigt werden, die nicht mit Flaggen markiert sind, sowie alle Flaggen hervorgehoben werden, unter denen keine Mine liegt.

Das Fenster könnte wie folgt aussehen:



In diesem Anzeigebeispiel wurde das Feld ganz links unten mit der linken Maustaste angeklickt. Da das Spielfeld mit sehr wenigen Minen initialisiert wurde, wurden dadurch große Bereiche des Spielfelds aufgedeckt. Die mit „P“ (Flaggen) markierten Felder wurden danach mit der rechten Maustaste angeklickt.

Wie das Spielfeld genau realisiert werden soll, ist Ihnen freigestellt, wir empfehlen folgende Variante:

Das Spielfeld ist eine `GridPane`, in der wir jede einzelne Zelle direkt durch Objekte der Klasse `javafx.scene.control.Button` darstellen.

- Die Methode `void setText(String text)` erlaubt es uns, einen Text auf dem Knopf darzustellen (Beispielsweise eine Zahl, oder „X“ für eine Mine).

- 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)` ermöglicht, auf Drücken des Knopfes zu reagieren. Am einfachsten geht das mit einem Lambda-Ausdruck:

  ```
  int x; int y; GridPane pane;
  ... x, y, pane initialisieren ...
  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 dem MVC-Pattern (wird später im Plenum behandelt). Allerdings ist es bei eine 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) Vervollständigen Sie nun Ihre Implementierung in folgender Reihenfolge:

   i) Implementierung des Modells, Initalisierung eines Spielfeldes mit fester Größe (einstellbar über eine Konstante im Quelltext).
   ii) Anzeige des Spielfelds.
   iii) Drücken eines Knopfes zeigt die hier liegende Zahl an oder deckt die Mine auf.
   iv) Das Drücken von bereits aufgedeckten Spielfeldern wird ignoriert.
   v) Nach jedem Zug wird die Gewinnbedingung geprüft und ggf. angezeigt.
   vi) Wenn ein Feld ohne Mine in der Nachbarschaft aufgedeckt wird, so sollen die Nachbarfelder auch aufgedeckt werden.
   vii) Anzeigen der Flaggen implementieren.
   viii) Breite und Höhe des Spielfelds vor Spielbeginn eingeben und anstelle der Konstanten verwenden.

Hinweise:

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

  `setOnAction` reagiert nur auf Links-Klicks; für Rechts-Klicks 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");
  }
});
```

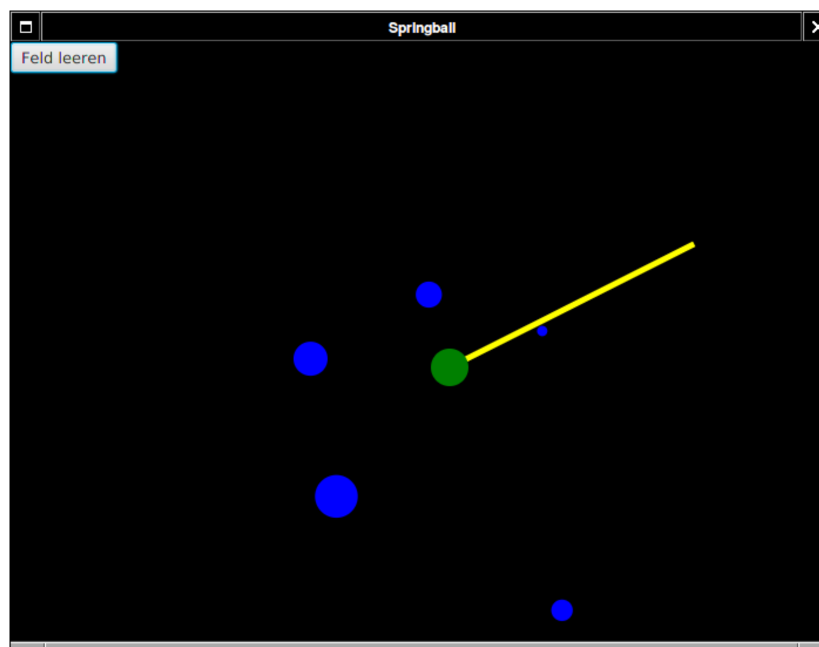
- Zufallszahlen können mit der Klasse `Random` bzw. mit deren Methode `nextInt` erzeugen.

- 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.

Aufgabe 2 Implementieren Sie „Springball“ mit Hilfe von JavaFX.

Dieses Programm zeigt eine physikalische Simulation von springenden Bällen an.

Eine Situation in der Simulation mit einigen Bällen könnte beispielsweise wie folgt aussehen:



- Die Simulation läuft „in Echtzeit“ ab. Das bedeutet, dass sich die Position von allen Bällen automatisch ändert, ohne dass explizit auf einen Knopf für die nächste Runde gedrückt werden muss.
- Mithilfe der Maus kann man neue Bälle in gewünschter Geschwindigkeit erstellen. Klickt man eine Stelle an, so entsteht hier ein Ball. Hält man die Maustaste gedrückt und bewegt die Maus weiter, bevor man sie loslässt, so gibt man an, in welcher Geschwindigkeit sich die Bälle bewegen sollen.
- Bälle werden durch Gravitation nach unten gezogen.

- Trifft ein Ball eine Seitenwand oder den Boden, so wird er reflektiert. Durch die Reflexion nimmt die Geschwindigkeit jedoch um 10% ab.
- Treffen sich zwei Bälle, so zerteilen sie sich in kleinere Bälle. Die Fläche der dabei entstehenden Bälle soll gleich der Fläche der ursprünglichen Bälle sein. Sie sollen dann in verschiedene Richtungen auseinander fliegen. So entstehende Ballteile, die kleiner als eine gewisse Grundgröße sind, sollen dabei allerdings nicht erstellt, sondern einfach gelöscht werden.
- Es soll eine Möglichkeit geben, wie man das Feld wieder komplett leeren kann.

Für jeden Ball sollten folgende Werte gespeichert werden

- `double x; double y;`

  Dies gibt die Position des Balls an.
- `double dx; double dy;`

  Dies gibt die Änderung des Position des Balls pro Sekunde an.
- `double size;`

  Dies ist die Größe des Balls.

Ein verbreiteted Verfahren, welches wir auch hier verwenden wollen, unterteilt die Zeit kleine, gleich große Zeiteinheiten, so genannte „Ticks“.

Schreitet die Simulation um einen Tick, also einen Teil `p` einer Sekunde voran, so wird

- `x` dabei um `dx*p` erhöht.
- `y` dabei um `dy*p` erhöht.
- `dx` unverändert gelassen, es sei denn, der Ball stößt an eine Wand.
- `dy` dabei um `g*p` erhöht, wobei `g` ein zu überlegender Wert für die Gravitation ist.

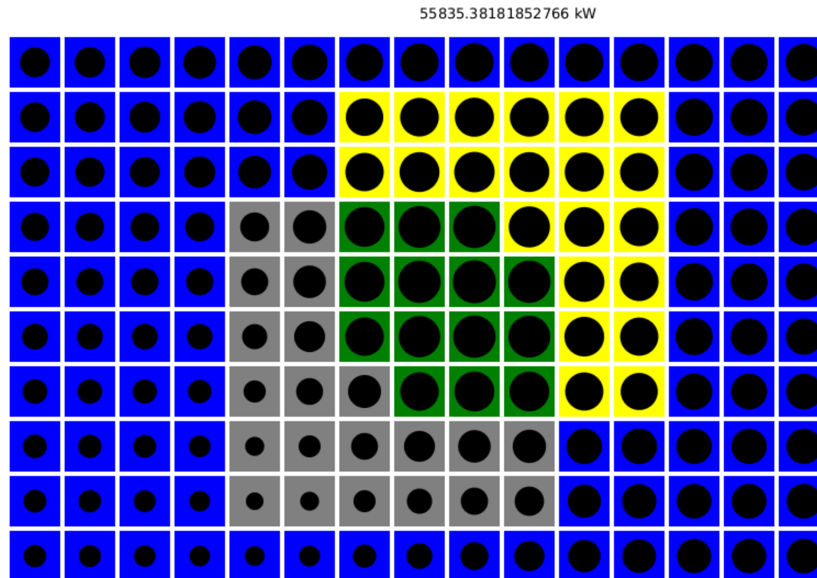
Für die Reflexion an der linken Wand ist zu prüfen, ob gleichzeitig `x` kleiner als 0 ist, wie auch `dx` kleiner als 0 ist. In dem Fall soll `x` auf 0 und `dx` auf `-dx` gesetzt werden.

Hinweis: In der Literatur findet man oft die Angabe, „Wenn `x` kleiner als 0 ist, soll `dx` auf `-dx` gesetzt werden“. Dieses Vorgehen ist wesentlich fehleranfälliger und der Grund, warum man in vielen Spielen in der Wand steckbleiben kann; ist man einmal tief genug in der Wand und langsam genug, so ändert sich die Geschwindigkeit immer wieder so, dass man tiefer in die Wand hineingezogen wird.

Für den Boden und die rechte Wand ist ein analoger Mechanismus mit teilweise anderen Variablen und Werten zu implementieren.

Aufgabe 3 Implementieren Sie „Kernreaktor“ mit Hilfe von JavaFX.

Dieses Programm stellt eine sehr stark vereinfachte Simulation eines Kernreaktors dar.



- Das Simulationsfeld ist hier in ein rechteckiges Gitter eingeteilt. Jedes Feld besteht aus einem Material und einem Strahlungswert.
- Material ist hierbei Luft, Uran, Blei oder Regelblock.

  Es bietet sich an dafür ein Enum zu verwenden (Enum wird erst viel später im Plenum behandelt, wer diese jetzt noch nicht kennt, kann einen `int` verwenden).

  Mit der Maus soll das Material auf jedem Feld wählbar sein.

  Bei Programmstart sollen alle Felder Luft enthalten.
- Jedes Material hat zwei Grundeigenschaften: Aktivität, Rückstrahlung

  Diese Werte sind

  - Luft: Aktivität:0 Rückstrahlung:0.99
  - Uran: Aktivität:1 Rückstrahlung:1.12
  - Blei: Aktivität:0 Rückstrahlung:0.7
  - Regelblock, falls Strahlungswert < 1000: Aktivität:0 Rückstrahlung:0.99
  - Regelblock, falls Strahlungswert ≥ 1000: Aktivität:0 Rückstrahlung:0.7
- Die Simulation ist in kleine Zeiteinheiten, so genannte Ticks, unterteilt. In jedem Tick wird auf jedem Feld der Strahlungswert neu berechnet. Um dies auf einem Feld zu berechnen wird die Aktivität A, sowie die Rückstrahlung R auf diesem Feld benötigt. Zudem wird der Durchschnittswert d der alten Strahlungswerte (also Strahlungswerte vor diesem Tick) auf allen senkrecht oder waagrecht benachbarten Felder benötigt (nicht die diagonal benachbarten Felder).

  Der neue Strahlungswert beträgt dann $A + R * d$.

  Lassen Sie etwa 50 Ticks in der Sekunde laufen.

- Status: Die Stromproduktion entspricht der Summe der Strahlungswerte aller Felder. Wenn an einem *einzelnen* Feld der Strahlungswert über 10000 steigt, dann explodiert der Reaktor und die Simulation ist beendet.

  Zeigen Sie beides geeignet an.

- Deklarieren Sie notwendige Konstanten wie die Größe des Spielfeldes explizit mit static final. Ausnahme: Die Werte werden über ein Eingabefenster, Konsole oder Kommandozeilenargumente eingelesen.

Für gesteigerten Realismus können Sie noch optional einbauen, dass die Regelblöcke nicht sofort reagieren, sondern etwa 5 Sekunden nach Über- oder Unterschreitung des Strahlungswertes 1000 die neuen Werte annehmen.