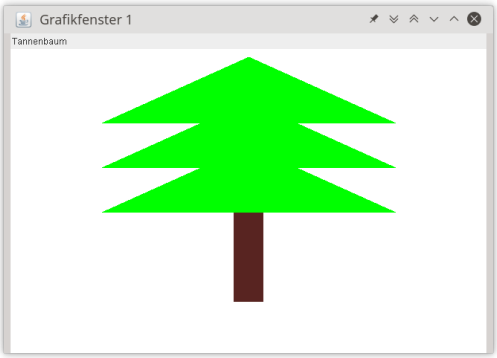
Prof. Martin Hofmann, PhD
Dr. Steffen Jost

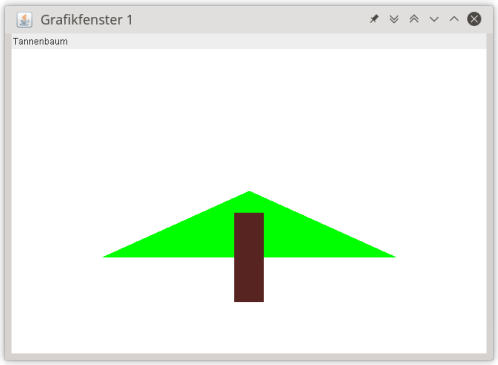
Ludwig-Maximilians-Universität München
Institut für Informatik
4. Dezember 2017

Lösungsvorschlag zur 7. Übung zur Vorlesung
Einführung in die Programmierung

A7-1 ***Dreiecks Rätsel*** EiP-Teilnehmer Thilo Tannenbaum tüftelt tierisch-gern tolle Tangrams. Da der dazu dringend Dreiecke braucht, hat er sich eine Klasse `Dreieck` geschrieben. Sein erstes einfaches Tangram soll ein Tannenbaum sein, ausgegeben von `DreieckDemo`:



Thilo's toller Traum



Thilo's traurige Tatsache

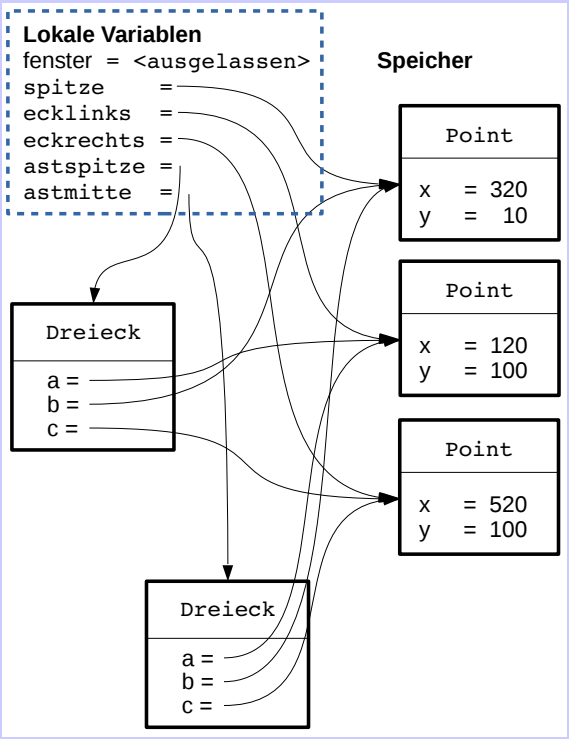
a) Thilo versteht leider nicht, warum er nur ein einzelnes grünes Dreieck bekommt. Finden Sie den Fehler? Thilo's Code sollte diesem Übungsblatt beiliegen.
WICHTIG: Bearbeiten Sie einfach die nächste Teilaufgabe, welche Ihnen eine Hilfestellung gibt, falls Sie den Fehler nach 5 Minuten nicht gefunden haben — was wir für Java-Anfänger auch erwarten würden! (Alte Java-Experten dürfen sich gerne schämen.)

b) Thilo findet den Fehler einfach nicht und bittet daher seine Kommilitonin Trixi Triangel um Hilfe. Trixi versteht den Fehler zwar auch nicht, kann aber trotzdem weiterhelfen: in der EiP-Vorlesung am 28.11.17 hat Sie gehört, dass Immutable-Objekte einfacher zu verstehen sind. Sie schreibt schnell eine Klasse `Position`, so wie diese in der Game of Life Fallstudie vorgestellt wurde. Dann ersetzt Sie einfach in den Klassen `Dreieck` und `DreieckDemo` überall `Point` durch `Position`. Damit dies funktioniert, muss Sie Position noch um die benötigten Methoden `translate` und `distance` erweitern. (Trixi's Code liegt ebenfalls bei.)

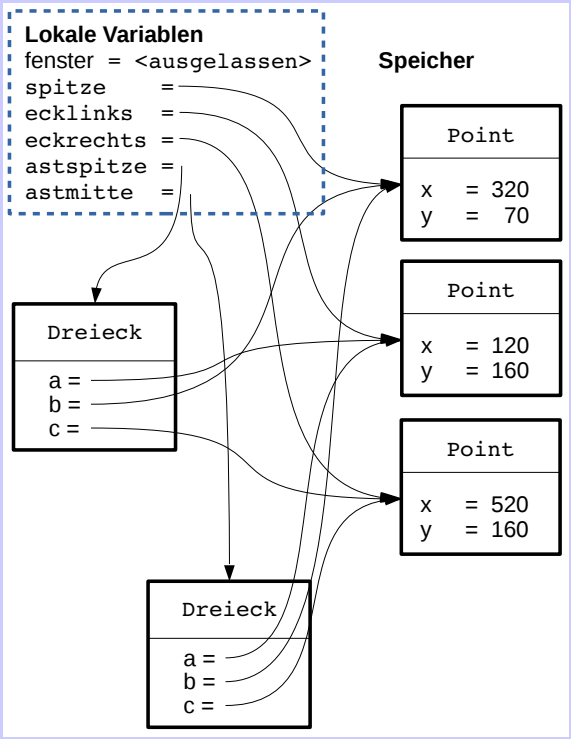
   Nun funktioniert es wie erwartet! Aber warum?

c) Egal ob Sie den Fehler bereits verstanden haben oder nicht: Zeichnen Sie jeweils ein Speicherbild (analog zu A2-2b) für die beiden in `DreickDemo` mit dem Kommentar `\\Speicherbild` markierten Stellen!

LÖSUNGSVORSCHLAG:



an Position 1



an Position 2

Die Speicherbilder verraten uns das Problem: Der Konstruktor von Thilo's `Dreieck`-Klasse kopiert die `Point`-Objekte nicht, sondern speichert lediglich einen Verweis darauf. Wird eines dieser `Point`-Objekte später aus irgendeinem Grund geändert, dann verändert sich natürlich auch sofort das Dreieck — es speichert seine Ecke ja im gleichen Punkt! Meist ist es deshalb sinnvoll, wenn ein Konstruktor übergebene Objekte kopiert/klont („Defensives Kopieren").

Bei unveränderlichen Objekten (Immutable) wie bei der Klasse `Position` ist das natürlich nicht notwendig, da wir ja wissen, dass sich eine Position nie verändert. Daher ist dieser häufig vorkommende Fehler in Trixi's Version von vornherein ausgeschlossen!

Der Nachteil ist natürlich, dass die `translate`-Methode ein neues Objekt zurückgeben muss, wodurch Trixi's Dreick-Klasse die gespeicherten Positionen explizit ersetzen muss. Ein besseres Design erhalten wir, wenn Trixi's Dreieck-Klasse ebenfalls unveränderlich wäre, und deren `translate`-methode dann ein neues Dreieck zurückliefert — was auch gleich noch den Code für Thilo's Weihnachtsbaum vereinfachen würde!

A7-2 *Sudoku-Assistent* Beim Zahlenrätsel *Sudoku* soll eine 9×9 Tabelle, die in 3×3 Quadrate unterteilt ist, so mit den Ziffern 1 bis 9 gefüllt werden, so dass in jeder Zeile, jeder Spalte, und jedem der Quadrate jede Ziffer genau einmal vorkommt. Dabei sind schon einige Kästchen ausgefüllt, z.B. so:

		9	3					
1		4			6	2	5	
6		2				7		1
		7	4		3		2	
2		·						3
	6		8		2	5		
9		1				8		2
	4	5	2			9		6
					4			

Manche Kästchen sind sehr einfach zu füllen, z.B. das mit einem Pünktchen markierte (5te Zeile, 3te Spalte). Man sieht leicht, dass hier nur die 8 passt. Alle anderen Ziffern kommen entweder in der Spalte, der Zeile, oder dem Quadrat schon vor.

Sie möchten sich einen elektronischen Assistenten programmieren, der alle Kästchen, in die nur noch eine Ziffer passt, automatisch für Sie ausfüllt. Die Sudoku-Tabelle repräsentieren Sie als zweidimensionales Array, das mit den Werten 0 bis 9 beschrieben wird. Dabei steht 0 für ein freies Kästchen. Sie haben schon einiges programmiert, z.B. das Ansteuern aller Felder und Einsetzen der allein passenden Ziffer, solange möglich. Außer dem Einlesen des Sudoku-Rätsels aus den Kommandozeilen-Argumenten und dem Ausdrucken einer Sudoku-Tabelle fehlt ihnen noch die Funktion, ob eine Ziffer in ein Kästchen passt.

Auf der Vorlesungshomepage finden Sie den Code zum Beispiel „Magisches Quadrat“, der Ihnen bereits viele nützliche Inspirationen liefern könnte. Weiterhin liegt diesem Übungsblatt bereits eine Vorlage `Sudoku.java` bei, in der Sie nur noch die fehlendne Methoden ausfüllen müssen.

Testen Sie die Funktionalität, z.B. mit obigem Rätsel (kein Zeilenumbruch)!

```
java Sudoku  ..93.....  1.4..625.  6.2...7.1  ..74.3.2.  2.......3  .6.8.25..
             9.1...8.2  .452..9.6  .....4...
```

Entwickeln Sie den Sudoku-Assistenten weiter! Hier einige Ideen:

- Lassen Sie Ihren Sudoku-Assistenten prüfen, ob das eingelesene Rätsel überhaupt lösbar sein kann.
- Erweitern Sie den Assistenten so, dass man interaktiv Kästchen ausfüllen kann. Der Assistent soll prüfen, ob die Setzung legal ist, und dann weiter automatisch ausfüllen.
- Überlegen Sie sich ein weiteres Kriterium, wann in ein Kästchen nur noch eine Ziffer passt, und erweitern Sie `putUniqueAt` entsprechend.
- Verpassen Sie dem Assistenten eine grafische Oberfläche mit `GraphicsWindow`.
- Animieren Sie das automatische Ausfüllen, so dass man es mitverfolgen kann.
- Lassen Sie den Assistenten Vorschläge machen, die nicht in einer Sackgasse enden.

LÖSUNGSVORSCHLAG:

```
public class Sudoku {

    private static final int MAX    = 9; // numbers 1..9, array dim 0..8
    private static final int SQUARE = 3; // size of sub-boards
    private static final int EMPTY  = 0; // special for empty field

    private int[][] board;  // entries: 0 = empty, 1..9

    // construct empty Sudoku board of size MAX
    public Sudoku() { board = new int[MAX][MAX]; }

    // parse initial set-up of Sudoku board
    // expects an array of rows (as Strings)
    public Sudoku (String[] arr) {
        this();
        for (int r = 0; r < MAX; r++)
            for (int c = 0; c < MAX; c++) {
                char x = arr[r].charAt(c);
                if (x >= '1' && x <= '9') board[r][c] = x - '0';
                  // or:  Integer.parseInt(String.valueOf(x));
                else board[r][c] = EMPTY;
            }
    }

    // print board in human-readable form
    public String toString() {
        StringBuffer s = new StringBuffer();
        for (int r = 0; r < MAX; r++) {
            for (int c = 0; c < MAX; c++) {
                s.append(' ');
                int entry = board[r][c];
                if (entry == EMPTY) s.append('.');
                else s.append(entry);
            }
            s.append('\n');
        }
        return s.toString();
    }

    public boolean legalInRow (int row, int number) {
        for (int col = 0; col < MAX; col++)
            if (board[row][col] == number) return false;
        return true;
    }

    public boolean legalInCol (int col, int number) {
        for (int row = 0; row < MAX; row++)
            if (board[row][col] == number) return false;
        return true;
    }

    public boolean legalInSquare (int row, int col, int number) {
        // compute left-upper corner
        int r0 = (row / SQUARE) * SQUARE;
        int c0 = (col / SQUARE) * SQUARE;
        // scan square
        for (int rd = 0; rd < SQUARE; rd++)
            for (int cd = 0; cd < SQUARE; cd++)
                if (board[r0 + rd][c0 + cd] == number) return false;
        return true;
    }

    // checks if it is legal to put number in field (row,col)
```

```
    public boolean legalAt (int row, int col, int number) {
        return legalInRow (row, number)
            && legalInCol (col, number)
            && legalInSquare (row, col, number);
    }

    // tries to fill position (row,col) with the unique matching number
    // returns true if successful
    public boolean putUniqueAt (int row, int col) {
        int number = EMPTY;
        for (int n = 1; n <= MAX; n++)
            if (legalAt(row,col,n))
                if (number == EMPTY) number = n; // first match
                else return false;               // second match
        board[row][col] = number;
        return true;
    }

    // tries to fill with unique match at all positions
    // returns true if successful
    public boolean putUniques () {
        boolean result = false;
        for (int r = 0; r < MAX; r++)
            for (int c = 0; c < MAX; c++)
                if (board[r][c] == EMPTY && putUniqueAt(r,c))
                    result = true;
        return result;
    }

    // repeats filling all uniquely determined positions
    // as long as it finds one
    // returns true if it could fill at least one field
    public boolean repeatPutUniques() {
        boolean result = putUniques();
        if (result) while (putUniques()) {}
        return result;
    }

    // read initial board configuration from the first MAX command line
    // arguments and fill all uniquely determined positions
    public static void main(String[] args) {
        System.out.println("Sudoku Assistant");
        Sudoku s = new Sudoku(args);
        System.out.println("Initial board");
        System.out.println(s.toString());
        if (s.repeatPutUniques()) {
            System.out.println("Filled board");
            System.out.println(s.toString());
        }
    }
}
```

In den vergangenen Vorlesung wurde eine Implementierung des “Game of Life” von Conway ausführlich vorgestellt. Die Implementierung finden Sie auf der Vorlesungshomepage. Verwenden Sie diesen Code jeweils als Vorlage für die drei hier folgenden Hausaufgaben!

H7-1 ***Game of Life I*** (3 Punkte; Abgabe: `H7-1.txt` oder `H7-1.pdf`)

Betrachten Sie die Methode `rundeBerechnen` in der Klasse `Spielfeld` unserer Implementierung des „Game of Life“: Auf den ersten Blick erscheint es dort unsinnig bzw. ineffizient, in Zeilen 97-101 noch eine zweite Schleife zu durchlaufen, nur um alle geänderten Zellen noch ein zweites Mal zu bearbeiten. Man könnte meinen, dass ein einzelne Schleife doch ausreichen könnte, etwa wie hier rechts abgebildet.

Erklären Sie in höchstens 5 Sätzen, warum der rechts abgebildete alternative Code fehlerhaft ist!

Hinweis: Wir fragen hier nach einem semantischen Fehler; der Code hier rechts sollte problemlos kompilieren und ausführen. Der Code berechnet jedoch nicht korrekt das „Game of Life“ von Conway, wie in der Vorlesung erklärt!

```
public ArrayList<Zelle> rundeBerechnen() {
  ArrayList<Zelle> geändert= new ArrayList<>();
  for (Zelle z : this.alleZellen()) {
    int lebendeNachbarn = 0;
    for (Zelle nachbar : getNachbarn(z)){
      if (nachbar.istLebendig())
        { lebendeNachbarn++; }
    }
    if (z.istLebendig()) {
      if (lebendeNachbarn <= 1 ||
          lebendeNachbarn >= 4 ) {
        Zelle neu = z.sterben();
        geändert.add(neu);
        this.setZelle(neu);
    } } else { // z.istTot()
      if (lebendeNachbarn == 3) {
        Zelle neu = z.auferstehen();
        geändert.add(neu);
        this.setZelle(neu);
  } } }
  return geändert;
}
```

LÖSUNGSVORSCHLAG:

Das Problem liegt darin, dass alle Änderungen eine Runde gleichzeitig ausgeführt werden müssen. Eine lebende Zelle, welche in der nächsten Runde stirbt, muss in der aktuellen Runde für alle 8 Nachbarn als lebend gewertet werden. Deshalb berechnen wir erst alle notwendigen Änderungen und führen diese dann erst später aus, so dass wir immer mit dem ursprünglichen Zustand zu Beginn der Runde rechnen. [3 Sätze]

Kommentar Eine alternative Implementierung könnte auch zuerst die Instanzvariable `spielfeld` kopieren und dann die lebenden Zellen immer in dieser unveränderten Kopie nachschlagen. Dann kann man jedoch nicht mit den Methoden `setZelle` und `getZelle` arbeiten, was aufgrund der verschachtelten `ArrayList` schnell umständlich und fehleranfällig wird. Hinzu kommt noch, dass das Kopieren des Spielfeldes meist mehr Zeit kostet, als noch einmal nur über die Änderungen zu iterieren.

H7-2 ***Game of Life II*** (4 Punkte; `SpielfeldWrapped.java`)

Implementieren Sie Lösung 2 „Wrap-Around“ von Folie 7.11 für das „Game of Life“. Sie dürfen dazu ausschließlich die Klasse `Spielfeld` verändern! Es reicht, wenn Sie darin die Methode `getZelle` und/oder `getNachbarn` ändern; Sie dürfen auch neue Methoden hinzufügen.

Benennen Sie Ihre geänderte Klasse um zu `SpielfeldWrapped`, so dass Sie beide Varianten vergleichen können. Natürlich muss dann auch noch in der Klasse `Spieler` entsprechend `SpielfeldWrapped` statt `Spielfeld` verwendet werden.

Testen Sie Ihre Lösung, in dem Sie einen Glider (Muster von Folie 7.5) über das Spielfeld laufen lassen. Dieser sollte Ränder und Ecken nahtlos überqueren. Wenn der Glider die Ecke unten rechts erreicht, sollte er oben links wieder ins Spielfeld hineinkommen!

Geben Sie ausschließlich eine kompilierbare Datei `SpielfeldWrapped.java` ab!

LÖSUNGSVORSCHLAG:

Wir ändern lediglich `getZelle()` wie folgt ab:

```
public Zelle getZelle(Position pos){        // WRAP-AROUND
  int x = pos.getX() % spielfeld.size();
  while (x < 0) x+= spielfeld.size();
  int y = pos.getY() % spielfeld.get(0).size();
  while (y < 0) y+= spielfeld.get(0).size();
  return spielfeld.get(x).get(y);
}
```

Die `while`-Schleifen benötigen wir, weil das Ergebnis des Modulo-Operators `%` in Java entgegen der mathematisch korrekten Definition auch negative Zahlen zurückliefern kann.

H7-3 ***Wireworld*** (5 Punkte; Verzeichnis `Wireworld`)
Ändern Sie den Code von „Game of Life“ so wenig wie möglich ab, um damit „Wireworld“ zu implementieren! In „Wireworld“ hat jede Zelle vier mögliche Zustände:

Zustand	Farbe	Setup-Clicks	Zustand	Farbe	Setup-Clicks
ISOLATOR	Schwarz	0	ELEKTRONKOPF	Blau	2
KUPFER	Gelb	1	ELEKTRONENDE	Rot	3

Bitte beachten Sie die angegebenen Farben und die Anzahl der Mausklicks um Zellen zu Initialisieren, da diese Vereinheitlichung unsere Korrektur enorm vereinfacht!

Die Spielregel ist nun wie folgt:

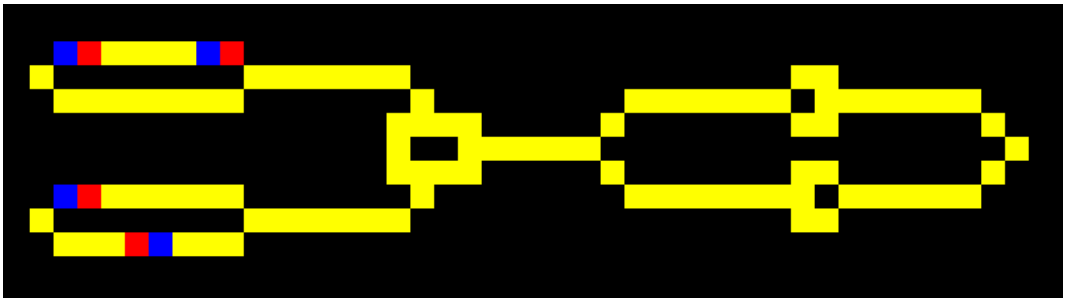
a) Eine Zelle mit Zustand ELEKTRONKOPF hat in der nächsten Runde immer den Zustand ELEKTRONENDE.

b) Eine Zelle mit Zustand ELEKTRONENDE hat in der nächsten Runde immer den Zustand KUPFER.

c) Eine Zelle mit Zustand ISOLATOR hat in der nächsten Runde immer den Zustand ISOLATOR; sie verbleibt also immer unverändert.

d) Eine Zelle mit Zustand KUPFER hat in der nächsten Runde den Zustand ELEKTRONKOPF, falls von Ihren 8 Nachbarzellen genau 1 oder 2 den Zustand ELEKTRONKOPF besitzen. Ansonsten verbleibt die Zelle im Zustand KUPFER.

Testen Sie Ihre Implementierung z.B. mit folgendem Start-Muster:



Wenn Sie es richtig gemacht haben, dann sollten die 4 Elektronen links immer weiter im Kreis herumgehen und jeweils einen neuen Elektronenimpuls in die nach rechts führenden Drähte entsenden. Wenn genau 1 Elektron den Kreisverkehr in der Mitte erreicht, wird es nach rechts weitergeleitet; falls 2 gleichzeitig ankommen, löschen diese sich gegenseitig aus (ein XOR-Gatter). Im rechten Teil haben wir zwei Dioden, d.h. ein Elektron kann die obere nur von rechts-nach-links durchqueren und die untere nur von links-nach-rechts.

Weitere spannende Beispiele für Wireworld finden Sie problemlos im Internet, z.B. auf Wikipedia oder dem Wireworld Computer.

Bitte beachten, damit wir Ihre Abgabe akzeptieren und korrigieren: Geben Sie alle Dateien in einem Unterordner `Wireworld` ab. Der Ordner muss weiterhin alle Klassen der ursprünglichen „Game of Life“ Vorlage enthalten, welche Sie angepasst haben. Sie dürfen neue Methoden und neue Klassen hinzufügen. Die Klassen `GraphicsWindow` und `Main` dürfen Sie jedoch nicht verändern (wir werden zur Kontrolle `Main` ausführen). In der Klasse `Param` dürfen Sie lediglich neue Parameter hinzufügen; vorhandenen Parameter zur Geometrie müssen weiter beachtet werden. Es dürfen keine `package`-Deklarationen enthalten sein!

LÖSUNGSVORSCHLAG:

Zuerst passen wir die Klasse `Zelle` an, damit diese 4 Zustände repräsentieren kann. Die Änderungen sind ganz trivial:

```
public class Zelle {
  // Hierfür wäre ein ENUM praktisch;
  // doch das Lernen wir erst am Ende der Vorlesung
  public static final int INSULATOR = 0;
  public static final int COPPER    = 1;
  public static final int EHEAD     = 2;
  public static final int ETAIL     = 3;

  private final Position position;
  private final int zustand;

  public Zelle(Position position, int zustand) {
    this.position = position;
    this.zustand  = zustand;
  }

  public Position getPosition() { return position; }

  public boolean isInsulator(){ return this.zustand == INSULATOR; }
  public boolean isCopper()   { return this.zustand == COPPER;    }
  public boolean isEHead()    { return this.zustand == EHEAD;     }
  public boolean isETail()    { return this.zustand == ETAIL;     }

  public Zelle umschalten() {
    if (zustand == INSULATOR) {
      return new Zelle(this.position, COPPER);
    } else if (zustand == COPPER ) {
      return new Zelle(this.position, EHEAD) ;
    } else if (zustand == EHEAD ) {
      return new Zelle(this.position, ETAIL) ;
    } else
      return new Zelle(this.position, INSULATOR);
} }
```

Ganz analog dazu muss in der Klasse `Ansicht` die Methode `zeichenZelle` an die vier Zustände angepasst werden, etwa so:

```
public void zeichneZelle(Zelle zelle) {
  Position position = zelle.getPosition();
  int x = position.getX() * GRÖßE;
  int y = position.getY() * GRÖßE;
  Rectangle rechteck = new Rectangle(x,y,Parameter.ZELLENGRÖßE,Parameter.ZELLENGRÖßE);
  if (zelle.isEHead()) {
    fenster.setColor(Parameter.FARBE_EHEAD);
  } else if (zelle.isETail()) {
    fenster.setColor(Parameter.FARBE_ETAIL);
  } else if (zelle.isCopper()) {
    fenster.setColor(Parameter.FARBE_KUPFER);
  } else if (zelle.isInsulator()) {
    fenster.setColor(Parameter.FARBE_INSULATOR);
  }
  fenster.fill(rechteck);
}
```

In `Parameter` sind natürlich die entsprechenden vier Konstanten für die Farben anzulegen.

Das Kernstück ist die Änderung in Runde berechnen:

```
public ArrayList<Zelle> rundeBerechnen() {
  ArrayList<Zelle> geänderte = new ArrayList<>();
  for (Zelle z : this.alleZellen()) {
    if (z.isEHead()) {
      geänderte.add(new Zelle(z.getPosition(),Zelle.ETAIL));
    } else if (z.isETail()) {
      geänderte.add(new Zelle(z.getPosition(),Zelle.COPPER));
    } else if (z.isCopper()) {
      int electronHeads = 0;
      for (Zelle nachbar : getNachbarn(z)) {
        if (nachbar.isEHead()) {
          electronHeads++;
        }
      }
      if (electronHeads >= 1 && electronHeads <= 2) {
        geänderte.add(new Zelle(z.getPosition(), Zelle.EHEAD));
      }
    }
  }
  for (Zelle z : geänderte) {
    this.setZelle(z);
  }
  return geänderte;
}
```

Alles andere kann unverändert übernommen werden!

Abgabe: Lösungen zu den Hausaufgaben können bis Sonntag, den 10.12.17, 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/`).