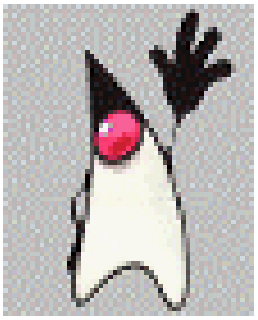
Grafische Oberflächen, Listener, Threads



Prof. Dr. Edgar Jäger

1 Einleitung

In den folgenden Kapiteln werden anhand einer Beispielanwendung mit grafischer Oberfläche wichtige Elemente der objektorientierten Programmierung (***OOP***) eingeführt.

Ein Bestandteil der ersten Java-Version (Java 1.0) war das sog. ***AWT*** (***Abstract Window Toolkit***), welches die ***GUI*** (***Graphical User Interface***) Bibliothek von Java 1.0 enthielt. Dieser erste Zugang wurde in den Folgeversionen Java 1.1 und Java 1.2 durch einen klareren, strenger objektorientierten Zugang ersetzt. In Java 1.2 wurden schließlich die ***JFC*** (***Java Foundation Classes***) eingeführt. Deren GUI-Teil wurde ***Swing*** genannt. Swing enthält all die Elemente, die von einem modernen ***UI*** (***User Interface***) erwartet werden: Buttons, die mit Bildern unterlegt sind, Tooltips, Bäume, Gitter, Tabellen, Registerkarten, Tooltips usw.

Wir nutzen deshalb von Anfang an Swing für Anwendungen mit grafischer Oberfläche.

Unsere Anwendung soll einen **Mähroboter** simulieren.

2 Wichtige GUI-Klassen

Als erste Teilaufgabe soll das zu mähende Grundstück festgelegt werden, indem seine Grenze vom Benutzer in Paintbrush-Manier gezeichnet wird.

2.1 Die Swing-Klassen JFrame und JPanel

JFrame ist Kindklasse der AWT-Klasse Frame**:**

class JFrame extends Frame.

Damit erbt JFrame alle Eigenschaften und Fähigkeiten von Frame. Diese Klasse stellt ein Fenster dar, welches mit Rahmen, Titelleiste und optionalem Menü versehen ist. Die Klassen ***`JFrame`***, ***`JDialog`***, ***`JWindow`*** und ***`JApplet`*** sind die einzigen ***Top-Level-Container***, also die obersten Container, welche andere Komponenten wie Schaltflächen (Buttons), Checkboxes, usw. enthalten können. Top-Level-Container selbst können nicht in anderen Containern enthalten sein.

Wir sehen hier schon die Namenskonvention von Swing: AWT-Klassen, die mit neuer Swing-Funktionalität ergänzt wurden, sind durch ein „J“ am Anfang zu erkennen.

Die „Ahnenreihe“ von JFrame sieht in der ***UML*** (***Unified Modeling Language***) wie folgt aus. Die Pfeilspitze zeigt dabei immer zur Elternklasse. Vererbung drückt die Beziehung „***ist ein***“ aus: Ein JFrame ist ein Frame, ein Frame ist ein Window, ein Window ist ein Container,…

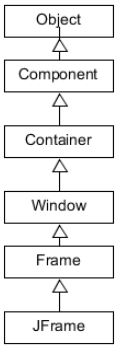


Abbildung 1: Elternklassen von JFrame

Die `JFrame`-Klasse besitzt ein sog. ***Content-Pane***, welches dazu dient, Komponenten wie Buttons, Textfelder usw. aufzunehmen und das Layout zu setzen.

In unserer Anwendung ist das Content-Pane vom Typ ***`JPanel`***. Dabei handelt es sich um einen Container, der andere Komponenten aufnimmt.

Die folgende Klasse ***Rahmen*** belegt das Content-Pane mit einem `JPanel`-Element:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Rahmen {
      public static JFrame inFrame(String title, JPanel jp, int width,
      int height) {
            JFrame frame = new JFrame(title);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.getContentPane().add(jp,BorderLayout.CENTER);
            frame.setSize(width,height);
            frame.setResizable(true);
            frame.setVisible(true);
            return frame;
      }
}
```

Listing 1: Ein Rahmen für eine Anwendung mit grafischer Oberfläche

Die Anweisung

```
JFrame frame = new JFrame(title);
```

erzeugt ein Fenster, dessen Titelleiste mit dem angegeben Text belegt wird. Mit

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

wird die Reaktion auf das Klicken des Schließen-Felds so abgeändert, dass nunmehr die Anwendung beendet wird. Die Default-Aktion wäre, dass das Fenster verschwindet, ohne dass die Anwendung beendet wird. Das Fenster erhält die angegebene Größe und wird mit

```
frame.setResizable(true);
```

so eingestellt, dass die Größe durch den Benutzer nachträglich verändert werden kann.

Die Klasse Rahmen nutzen wir wie folgt, um eine sehr einfache Anwendung mit grafischer Oberfläche zu erzeugen:

```
import javax.swing.*;

public class Grundstueck extends JPanel {

      public static void main(String[] args) {
         Rahmen.inFrame("Mein erstes GUI-Programm", new Grundstueck(),
500, 200);
       }

}
```

Listing 2: Verwendung des GUI-Rahmens

Führen wir diese Anwendung aus, so zeigt sich folgendes Fenster:

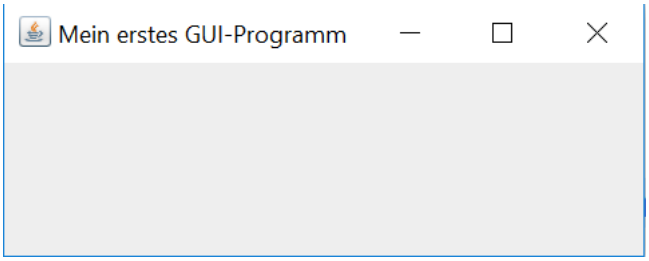


Abbildung 2: Erste GUI-Anwendung

Wichtig ist, dass unsere Klasse (hier: `Grundstueck`) ein **`JPanel`** ist. Das erreichen wir, indem wir `JPanel` beerben: `class Grundstueck extends JPanel`.

In unserem `JPanel`-Objekt können wir nun Bedienelemente anordnen, zeichnen usw.

Im Konstruktor unserer Klasse bestimmen wir, welche Bedienelemente wir benötigen. Beginnen wir mit einer Schaltfläche für die Farbe:

```
Grundstueck() {
            JButton farbe = new JButton("Farbe");
            add(farbe);
}
```

Listing 3: Erzeugen und Hinzufügen einer Schaltfläche

Nun erscheint eine Schaltfläche, die wir anklicken können. Diese verändert sogar beim Anklicken die Farbe, aber ansonsten geschieht nichts.

2.2 Reaktion auf Ereignisse: Event Listener

Wie reagiert man nun auf GUI-Ereignisse wie Auswahl eines Menüpunkts, Klicken eines Buttons, Auswahl eines Elements aus einer Liste usw.?

Eine Komponente kann bestimmte Ereignisse auslösen („***fire an event***"). Solche Ereignisse sind beispielsweise:

- ***ActionEvent:*** Klicken eines Buttons, Beenden der Eingabe in ein Textfeld, ...
- ***KeyEvent:*** Drücken/Loslassen einer Taste
- ***MouseEvent:*** Maus-Klicks und –bewegung
- ***MenuEvent:*** Auswahl Menüpunkt, Verlassen Menüpunkt

Jeder Ereignistyp wird durch eine bestimmte Klasse repräsentiert (**`ActionEvent`**, **`KeyEvent`**, **`MouseEvent`**...).

Ein Ereignis kann von einem sog. ***Event Listener*** „gefangen" und damit bearbeitet werden. Im Event Listener findet die entsprechende Aktion statt. Jeder Event Listener muss registriert werden. Dies geschieht über Methoden mit Namen ***`addXXXListener()`***, wobei XXX für den entsprechenden Ereignistyp steht.

Ein Listener ist als sog. ***Interface*** angelegt. Dabei handelt es sich um Schnittstellen, die ***abstrakte Methoden*** und Konstanten enthalten. Abstrakte Methoden bestehen nur aus einem Kopf ohne Implementierung. Sie können nicht direkt verwendet werden, sondern **müssen überschrieben werden**. Auf diese Weise wird erzwungen, dass man eine oder mehrere Methoden mit vorgegebenem Namen und vorgebener Argumentliste implementiert, in welcher die **Reaktion** auf bestimmte Ereignisse erfolgt. Die so zur Verfügung gestellten Methoden werden nie von uns selber, sondern immer als Reaktion auf bestimmte Ereignisse vom System aufgerufen.

Ein Listener muss das entsprechende ***interface*** implementieren. So hat ein ***`ActionListener`*** die Methode ***`actionPerformed()`*** zu realisieren, ein ***`MenuListener`*** die Methoden ***`menuSelected(), menuDeselected(), menuCanceled()`***.
Es ist zweckmäßig, die Listener als ***innere Klasse*** zu realisieren, d. h. die Klassendeklaration innerhalb der umgebenden Klasse zu verfassen. Man erhält so eine logische Gruppierung von Bedienelementen und entsprechender Reaktion. Ferner ist die betreffende Klasse ausschließlich für das entsprechende Bedienelement zuständig.

Wir beginnen mit einer einfachen Reaktion auf das Klicken von **Farbe**. Zunächst soll einfach eine entsprechende Information ausgegeben werden:

```
import java.awt.event.*;
import javax.swing.*;

public class Grundstueck extends JPanel {
	Grundstueck() {
		JButton farbe = new JButton("Farbe");
		add(farbe);
		farbe.addActionListener(new FarbeAL());
	}

	class FarbeAL implements ActionListener {
		public void actionPerformed(ActionEvent e) {
			System.out.println("Farbe geklickt");
		}
	}

	public static void main(String[] args) {
		Rahmen.inFrame("Mein erstes GUI-Programm", new Grundstueck(), 500,
200);
	}
}
```

Listing 4: ActionListener mit Reaktion auf Klicken der Schaltfläche

Wenn wir einen Button mit Beschriftung „Farbe“ klicken, erwarten wir natürlich mehr als die oben implementierte Textausgabe. Swing bietet eine Reihe von fertigen Standarddialogfenstern z.B. für die Dateiauswahl und die Farbauswahl:

```
class FarbeAL implements ActionListener {
	public void actionPerformed(ActionEvent e) {
		JColorChooser.showDialog(null, "Hintergrundfarbe" , Color.GREEN);
	}
}
```

Listing 5: Aufruf des Dialogs zur Farbauswahl

Was nützt die schönste Farbauswahl, wenn man am Ende nicht an die ausgewählte Farbe kommt? Da der Farbauswahldialog die ausgewählte Farbe als Rückgabewert liefert, ist das sehr einfach. Wir übernehmen die ausgewählte Farbe in der neuen Membervariablen **`hintergrundfarbe`**:

```
import java.awt.Color;
import java.awt.event.*;
import javax.swing.*;

public class Grundstueck extends JPanel {

	Color hintergrundfarbe;

	Grundstueck() {
		hintergrundfarbe = Color.RED;
		JButton farbe = new JButton("Farbe");
		add(farbe);
		farbe.addActionListener(new FarbeAL());
	}
```

```
class FarbeAL implements ActionListener {
	public void actionPerformed(ActionEvent e) {
		Color ret = JColorChooser.showDialog(null, "Hintergrundfarbe" ,
Color.GREEN);
		if (ret != null)
			hintergrundfarbe = ret;
		}
	}

	public static void main(String[] args) {
		Rahmen.inFrame("Mein erstes GUI-Programm", new Grundstueck(), 500,
200);
	}
}
```

Listing 6: Übernahme der ausgewählten Farbe

Natürlich sollten wir die ausgewählte Farbe nur dann übernehmen, wenn der Dialog nicht mit CANCEL oder durch Klicken des Schließen-Felds geschlossen wurde. Wird der Dialog nicht mit OK beendet, ist der Rückgabewert das null-Handle.

2.3 Zeichnen: paintComponent()

Nun soll das Grundstück gezeichnet werden. Dazu ist die Methode **paintComponent()** zu überschreiben. Wir zeichnen zunächst den Hintergrund in der soeben ausgewählten Farbe:

```
	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		Dimension d = getSize();
		g.setColor(hintergrundfarbe);
		g.fillRect(0, 0, d.width, d.height);
	}
```

Listing 7: Zeichnen

Zunächst muss die **paintComponent**-Methode der Oberklasse ausgeführt werden. Danach fragen wir mit **getSize()** die Größe unseres Fensters ab, übernehmen die ausgewählte Farbe in den Grafikkontext und füllen das komplette Fenster mit der gewählten Farbe. In vielen Situationen führen Änderungen dazu, dass neu gezeichnet werden muss. In solchen Fällen wird **paintComponent()** nicht direkt aufgerufen, sondern über **repaint().** Wählen wir z.B. eine neue Farbe, so sorgen mit **repaint()** wir dafür, dass diese auch sofort für den Hintergrund verwendet wird:

```
class FarbeAL implements ActionListener {
	public void actionPerformed(ActionEvent e) {
		Color ret = JColorChooser.showDialog(null,
"Hintergrundfarbe" , Color.GREEN);
		if (ret != null) {
			hintergrundfarbe = ret;
			repaint();
```

```
            }
        }
}
```

Listing 8: Neuzeichnen nach Wahl einer neuen Farbe

Nun soll die Grundstücksgrenze in Paintbrush-Manier gezeichnet werden: Das Drücken einer Maustaste soll den Stift auf dem „Blatt“ aufsetzen, und Mausbewegungen sollen als Kurve gezeichnet werden.

Dazu müssen wir auf die Mausereignisse reagieren. Das wichtigste Ereignis in diesem Zusammenhang ist „Mausbewegung bei gedrückter Maustatste“, für welches die Methode **`mouseDragged()`** zuständig ist. Diese gehört zur Ereignisgruppe ***MouseMotion*** (Mausbewegung). Wir registrieren daher im Konstruktor einen ***`MouseMotionListener`***:

```
addMouseMotionListener(new MML());
```

In der Klasse MML implementieren wir die uns interessierende Methode **`mouseDragged()`**:

```
class MML implements MouseMotionListener {
        public void mouseDragged(MouseEvent m) {
        }
}
```

Listing 9: Erster Ansatz für die Klasse MML

Der Compiler weist darauf hin, dass wir zwar behaupten, ein **`MouseMotionListener`** zu sein („*implements MouseMotionListener*“), diese aber nicht sind, weil mindestens ein zu einem **`MouseMotionListener`** gehöriges Ereignis nicht implementiert wurde. In unserem Fall geht es um das Ereignis „Mausbewegung ohne gedrückte Maustaste“, welches uns zwar nicht interessiert, aber trotzdem implementiert werden muss.
Für dieses Problem gibt es für alle Listener zwei Lösungen:

1. Wir implementieren die noch fehlenden Methoden als leere Funktionen. Dabei werden wir vom Editor unterstützt, der als Quick fix „add unimplemented methods“ anbietet.
2. Wir beerben die sog. ***Adapterklasse***, in unserem Fall die Klasse ***`MouseMotionAdapter`*** und überschreiben nur die uns interessierende Methode. Die Adapterklasse enthält leere Implementierungen für alle erforderlichen Ereignisbehandlungen. Listing 9 müssten wir in diesem Fall wie folgt ersetzen:

```
class MML extends MouseMotionAdapter {
        public void mouseDragged(MouseEvent m) {
        }
}
```

Listing 10: MML als Kindklasse von MouseMotionAdapter

Obwohl der unter 2. beschriebene Ansatz mit weniger Code verbunden ist, entscheiden wir uns für die erste Variante, weil diese als sicherer angesehen werden kann: Angenommen, Sie vertippen sich in Listing 10 (z.B. *mouseDraged()*). In diesem Fall erhalten Sie keinerlei

Fehlermeldung, weil Sie eine neue Methode (*mouseDraged()*) erzeugt haben, die niemanden interessiert. Die für das Ereignis „Mausbewegung bei gedrückter Maustaste“ zuständige Funktion ist durch die Beerbung von **`MouseMotionAdapter`** vorhanden, aber leer. Das bedeutet, dass Ihre Anwendung nicht auf Mausbewegungen reagiert.

Nehmen wir da Quick-Fix-Angebot an, so wird in MML folgende Methode zusätzlich erzeugt:

```
        @Override
        public void mouseMoved(MouseEvent arg0) {
              // TODO Auto-generated method stub
        }
```

Listing 11: Erforderliche Methode mouseMoved als quick fix erzeugt

@Override weist darauf hin, dass wir diese Methode überschreiben sollten. Da uns das Ereignis nicht interessiert, belassen wir es bei dem leeren Rumpf.

Da wir die Grundstücksgrenze später speichern wollen, merken wir uns sämtliche Punkte, die wir mit gedrückter Maustaste passieren. Da wir am Anfang nicht wissen, wie viele Punkte das insgesamt sein werden, verwenden wir hier die Klasse **`Vector`**, und für die einzelnen Elemente die Klasse **`Point`** (Punkt mit zwei ganzzahligen Koordinaten). Die entsprechende Variable legen wir als Membervariable unserer Klasse an:

```
        Vector<Point> grenze;
```

Im Konstruktor erzeugen wir mit

```
              grenze = new Vector<Point>();
```

den Behälter. Nun können wir in **`mouseDragged()`** jeden Punkt, den wir passieren, hinzufügen und dann ein Neuzeichnen veranlassen:

```
    class MML implements MouseMotionListener {
        public void mouseDragged(MouseEvent m) {
              grenze.add(new Point(m.getX(),m.getY()));
              repaint();
        }

        @Override
        public void mouseMoved(MouseEvent arg0) {
              // TODO Auto-generated method stub
        }
    }
```

Listing 12: mouseDragged(): Speichern aller besuchten Punkte, Neuzeichnen

Wir haben hier erstmals das Argument vom Typ **`MouseEvent`** verwendet, um die aktuelle Mausposition zu erhalten. Im Allgemeinen erhällt das Event-Argument nützliche

Informationen, die das Ereignis näher spezifizieren, z.B. im Fall eines Tastendrucks die gedrückte Taste.

Nun ergänzen wir **`paintComponent()`** entsprechend:

```
public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Dimension d = getSize();
      g.setColor(hintergrundfarbe);
      g.fillRect(0, 0, d.width, d.height);

      g.setColor(Color.BLACK);
      for (int i=1;i<grenze.size();i++) {
            Point p1 = grenze.elementAt(i-1);
            Point p2 = grenze.elementAt(i);
            g.drawLine(p1.x, p1.y, p2.x, p2.y);
      }
}
```

Listing 13: paintComponent mit Zeichnen der Grenze

Soll die Grenze dicker gezeichnet werden, müssen wir auf die Klasse ***Graphics2D*** zurückgreifen, welche **Graphics** beerbt und mit weiteren Funktionen ergänzt:

```
      Graphics2D g2 = (Graphics2D) g;
      g2.setStroke(new BasicStroke(10));

      g.setColor(Color.BLACK);
      for (int i=1;i<grenze.size();i++) {
            Point p1 = grenze.elementAt(i-1);
            Point p2 = grenze.elementAt(i);
            g.drawLine(p1.x, p1.y, p2.x, p2.y);
      }
```

Listing 14: Zeichnen mit dickerem Stift

***Hinweis**: Unsere Methode zum Festlegen der Grundstücksgrenze funktioniert nur, wenn die Grenze in einem Zug durchgezeichnet wird. Muss man den Stift absetzen, um z.B. eine Insel im Grundstück zu zeichnen, so hat man das Problem, dass nun der Punkt, bei dem der Stiift abgesetzt wurde, mit dem Punkt, wo er wieder aufesetzt wurde, verbunden wird. Das könnte man vermeiden, indem man mit einem `MouseListener` das Absetzen und das Aufsetzen des Stifts abfängt und in den Vector `grenze` Markierungen (z.B. ein Punkt mit den Koordinaten `(-1,-1)`) einfügt, um das korrekte Zeichnen zu ermöglichen.*

2.4 Trennung zwischen Bedienelementen und Zeichenfläche

Der Farbe-Button befindet sich aktuell im Zeichenfeld und stört dort etwas. Außerdem ist das unüblich. Schließlich ist z.B. bei einem WORD-Dokument auch der Bereich, den Sie selbst gestalten können, von der Menü-und Symbolleiste getrennt. Wir ordnen deshalb die Elemente unserer Anwendung so an, dass Bedienelemente und Zeichenfläche voneinander getrennt sind. Dazu gibt es in Swing die sog. ***Layout Manager***, mit denen gesteuert werden kann, wie sich die einzelnen Elemente verhalten, wenn man das umgebende Fenster vergrößert bzw. verkleinert. Im Fall eines ***`GridLayout`*** werden beispielsweise alle Elemente bei Vergrößerung und Verkleinerung des Fensters in gleichem Maße vergrößert.
Wir wählen das sog. ***`BorderLayout`***, welches über fünf Regionen verfügt: NORTH, SOUTH, EAST, WEST, CENTER. Bei Vergrößerung wird nur die Region CENTER in beiden Richtungen gedehnt. Die anderen Regionen wachsen jeweils nur in einer Richtung.

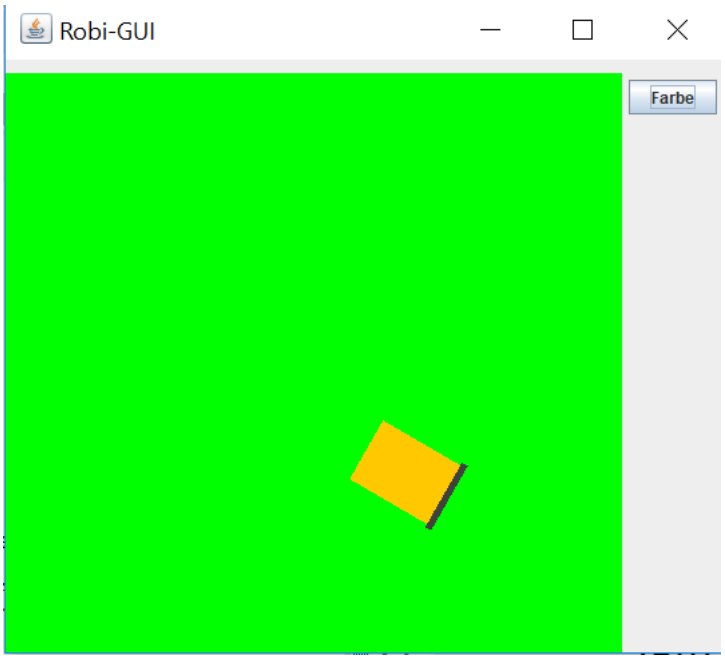


Abbildung 3: BorderLayout mit Grundstück-Klasse im Zentrum

Dies wird mit folgender **umgebenden Klasse** `GUI` erreicht. Beachten Sie, dass wir hier die Eigenschaft nutzen, dass ein `JPanel` ein Container ist, der wiederum andere JPanel-Elemente enthalten kann:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GUI extends JPanel {

      Grundstueck grund;
```

```
      GUI() {
            setLayout(new BorderLayout());
            JPanel oben = new JPanel();
            JPanel rechts = new JPanel();
            grund = new Grundstueck();
            add(grund, BorderLayout.CENTER);
            add(oben, BorderLayout.NORTH);
            add(rechts, BorderLayout.EAST);

            JButton farbe = new JButton("Farbe");
            rechts.add(farbe);
            farbe.addActionListener(new FarbeAL());
      }

      class FarbeAL implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                  Color ret = JColorChooser.showDialog(null,
"Hintergrundfarbe" , Color.GREEN);
                  if (ret != null) {
                        grund.hintergrundfarbe = ret;
                        repaint();
                  }
            }
      }

      public static void main(String[] args) {
            Rahmen.inFrame("Robi-GUI", new GUI(), 1500, 2000);
      }

}
```

Listing 15: BroderLayout gemäß Abbildung erzeugen

Diese Technik wenden wir auch künftig an, um neue Bedienelemente zu ergänzen: Das Bedienelement und seine Ereignisbehandlung platzieren wir in der Klasse GUI.

Um auf die Datenelemente (hier: **`hintergrundfarbe`**) zugreifen zu können, hat die Klasse GUI eine Membervariable vom Typ Grundstueck.

2.5 Speichern und Laden

Zum Speichern und Laden des Grundstücks führen ergänzen wir unsere Anwendung mit einem Menü. Die verwendeten Klassen sind:

- **`JMenuBar`** – Menüleiste
- **`JMenu`** – Menü, welches Menüpunkte enthält
- **`JMenuItem`** – Menüpunkt

Diese sind ein schönes Beispiel für eine „***Teil-Ganzes-Beziehungen***“: Ein Menüpunkt ist Teil eines Menüs, ein Menü ist Teil einer Menüleiste. Für derartige Beziehungen gibt es in der OOP die Begriffe ***Komposition*** und ***Aggregation***:

Komposition und Aggregation

Beide Begriffe bezeichnen „Teil-von“ –bzw. „besteht-aus“-Beziehungen. Die ***Komposition*** bezeichnet dabei eine stärkere Bezeihung dieser Art: Löscht man hier den „Behälter“, so werden auch die darin enthaltenen Elemente gelöscht. Das muss bei der schwächeren Form der „Teil-von“-Beziehung, der ***Aggregation***, nicht notwendig der Fall sein.

In unserem Beispiel werden mit Löschen eines Menüs alle darin enthaltenen Menüpunkte gelöscht, und mit Löschen der Menüleiste werden alle darin enthaltenen Menüs gelöscht. Daher können wir hier von einer **Komposition** sprechen. Das gleiche gilt für die Beziehung eines Ordners auf Ihrem Computer zu den enthaltenen Dateien. Wird der Ordner gelöscht, so verschwinden auch alle enthaltenen Dateien.

Beispiele für eine Aggregation:

1. Im Fall eines Hundebesitzers, der mehrere Hunde besitzt, würde man von einer Aggregation sprechen: Auch wenn der Hundebesitzer stirbt, leben die Hunde weiter.
2. In unserem obigen Beispiel hat die Klasse GUI ein Element der Klasse Grundstueck. Da letzteres auch ohne Bedienelemente existieren könnte, haben wir hier eine Aggregation.

In der UML werden Komposition und Aggregation durch Rauten dargestellt:

- **Ausgefüllte Raute**: Komposition
- **Nicht ausgefüllte Raute**: Aggregation

Im Fall der drei obigen Klassen sieht das entsprechende UML-Diagramm wie folgt aus:

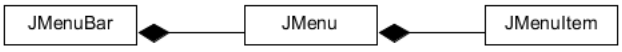


Abbildung 4: Kompositionsdiagramm

2.5.1 Implementierung: Laden und Speichern

Im Konstruktor von **GUI** legen wir die Menüleiste an und fügen **`ActionListener`** hinzu, welche bei Wahl der entsprechenden Menüpunkte aktiviert werden. Die Farbauswahl verlagern wir nun ebenfalls in ein Menü, so dass wir den Button entfernen können:

```
    GUI() {
          setLayout(new BorderLayout());
          JPanel oben = new JPanel();
          JPanel rechts = new JPanel();
          grund = new Grundstueck();
          add(grund, BorderLayout.CENTER);
          add(oben, BorderLayout.NORTH);
```

```
        add(rechts, BorderLayout.EAST);

        JMenuBar mb = new JMenuBar();
        JMenu datei = new JMenu("Datei");
        JMenu col = new JMenu("Farbe");
        JMenuItem neu = new JMenuItem("Neu");
        JMenuItem speichern = new JMenuItem("Speichern");
        JMenuItem laden = new JMenuItem("Laden");
        JMenuItem farbe = new JMenuItem("Hintergrundfarbe");

        datei.add(neu);
        datei.add(speichern);
        datei.add(laden);
        col.add(farbe);
        mb.add(datei);
        mb.add(col);
        oben.add(mb);

        speichern.addActionListener(new SpeichernAL());
        laden.addActionListener(new LadenAL());
        neu.addActionListener(new NeuAL());
        farbe.addActionListener(new FarbeAL());
    }
```

Abbildung 5: Menüleiste erzeugen

Wie Sie sehen, wird die „Teil-von“-Beziehung hier durch die Methode **add()** realisiert: Den Menüs werden die Menüpunkte mit **add()** hinzugefügt, die Menüs werden mit **add()** zur Menüleiste hinzugefügt, und schließlich wird die Menüleiste mit **add()** dem **JPanel** hinzugefügt.

Von professionellen Anwendungen erwarten wir für die Dateiauswahl einen entsprechenden Dialog. Diesen bietet Swing mit der Klasse **JFileChooser**:

```
    class SpeichernAL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser chooser = new JFileChooser(".");
            FileNameExtensionFilter filter =
              new FileNameExtensionFilter("Grundstueck-Dateien",
                    "gnd");
            chooser.setFileFilter(filter);
            int returnVal = chooser.showSaveDialog(null);
            if(returnVal == JFileChooser.APPROVE_OPTION) {
                String dateiname =
                      chooser.getSelectedFile().getName();
                try {
                    RandomAccessFile datei = new
                      RandomAccessFile(dateiname,"rw");
```

```
                        for (int i=0;i<grund.grenze.size();i++) {
                             Point p = grund.grenze.elementAt(i);
                             datei.writeInt(p.x);
                             datei.writeInt(p.y);
                        }
                        datei.close();
                   } catch (FileNotFoundException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                   } catch (IOException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                   }
              }

         }
    }
```

Listing 16: Speichern der Grundstücksgrenze

Mit

```
new JFileChooser(".");
```

erzeugen wir ein Objekt der Klasse `JFileChooser`, welches als **Startverzeichnis** das aktuelle Verzeichnis (`"."`) wählt. Da unsere Dateien einen Aufbau haben, den nur unser Porgramm kennt, versehen wir diese mit einer eigenen Endung („gnd“) und verwenden diese Endung als ***Filter***, so dass im folgenden Dialog nur Dateien mit dieser Endung und Unterverzeichnisse angezeigt werden. Mit **`showSaveDialog()`** wird die Dateiauswahl angezeigt. Der Rückgabewert dieser Methode gibt an, wie der Dialog beendet wurde. Der Wert `APPROVE_OPTION` zeigt an, dass eine Datei ausgewählt wurde. Andere Rückgabewerte zeigen ein Beenden ohne Auswahl an (z.B. Klicken von `CANCEL` oder Schließen-Feld).

Mit

```
chooser.getSelectedFile().getName();
```

holen wir den Namen der ausgewählten Datei. Unter diesem Namen speichern wir unsere Datei in einem sehr einfachen Format: Es werden jeweils die x- und y-Koordinate jedes Punktes der Grenze als **`int`** in die Datei geschrieben.

Das Laden funktioniert analog:

```
class LadenAL implements ActionListener {
	public void actionPerformed(ActionEvent arg0) {
		JFileChooser chooser = new JFileChooser(".");

	FileNameExtensionFilter filter = new FileNameExtensionFilter(
				"Grundstueck-Dateien", "gnd");
		chooser.setFileFilter(filter);
		int returnVal = chooser.showOpenDialog(null);
		if(returnVal == JFileChooser.APPROVE_OPTION) {
			String dateiname =
			chooser.getSelectedFile().getName();
			RandomAccessFile datei=null;
			try {
				datei = new RandomAccessFile(dateiname,"r");
				grund.grenze.removeAllElements();
				while(true) {
					Point p = new Point();
					p.x=datei.readInt();
					p.y=datei.readInt();
					grund.grenze.addElement(p);
				}
			}
			catch (EOFException e0) {
					try {
						datei.close();
					} catch (IOException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
					}
					repaint();
			}
			catch (IOException e) {
			// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}
}
```

Listing 17: Laden der Grundstückgrenze

Schließlich haben wir noch einen Menüpunkt **Datei→Neu**, mit welchem wir das Löschen einer evtl. vorhandenen Grundstücksgrenze veranlassen können:

```
	class NeuAL implements ActionListener {
		public void actionPerformed(ActionEvent e) {
			grund.grenze.removeAllElements();
			repaint();
```

```
        }
    }
```

Listing 18: Löschen einer Zeichnung

3 Mähroboter bauen

Die erste Version unseres Mähroboters soll zunächst folgende Zustände und Methoden besitzen:

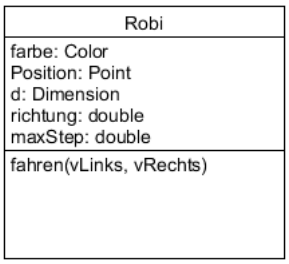


Abbildung 6: Klassendiagramm für den Roboter

Unsere Roboter soll als Rechteck dargestellt werden. Die Position des Roboters enthält die Koordinaten seines Mittelpunkts. Das Element **d** vom Typ **Dimension** enthält die Elemente **width** und **height** für Breite und Höhe.
Der Roboter soll ferner über zwei Motoren verfügen, welche zwei Räder (links und rechts) antreiben. Die Methode **fahren()** benötigt als Parameter die Geschwindigkeiten der beiden Motoren, welche zwischen 0 und 100 liegen sollen. 0 bedeutet dabei „Motor aus“.

```
    void fahren(double speedL, double speedR) {
        double spd = Math.sqrt(speedL*speedL+speedR*speedR);
        if ( spd < 1e-2)
            return;
      richtung = richtung-0.1*(Math.atan2(speedR,speedL)-Math.PI/4);
      position.x += maxStep*spd*Math.cos(richtung);
      position.y += maxStep*spd*Math.sin(richtung);
    }
```

Listing 19: Methode fahren

Wählt man für beide Motoren die gleiche Geschwindigkeit, so ändert sich die Richtung des Roboters nicht. Ansonsten macht man eine Linkskurve, wenn `speedL > speedR` und eine Rechtskurve, wenn `speedL < speedR`.

In der neuen Richtung bewegt sich der Roboter bei einem Aufruf der Methode **fahren()** um den Betrag $maxStep \cdot \sqrt{speedL^2 + speedR^2}$ weiter.

Um den Roboter grafisch darstellen zu können, müssen wir ihn so ausrichten, wie seine aktuelle Fahrrichtung angibt. Dazu ist ein wenig Mathematik erforderlich.

Um das rote Rechteck um einen bestimmten Winkel (hier: 30°) gegen den Uhrzeigersinn zu drehen, verschieben wir es zunächst so, dass sein Mittelpunkt bei (0,0) liegt.

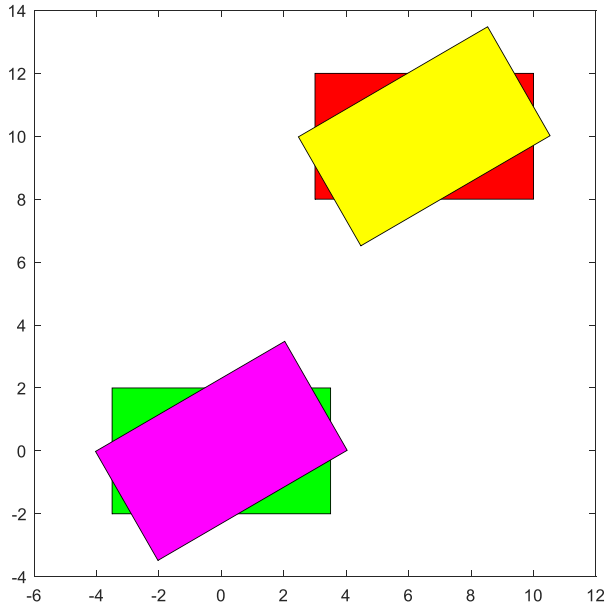


Abbildung 7: Drehung eines Rechtecks

In MATLAB geht das wie folgt:

```
p = [3,8;10,8;10,12;3,12];
 center = [sum(p(:,1))/4,sum(p(:,2))/4];
 fill(p(:,1),p(:,2),'r');
 q = zeros(length(p),2);
% Verschieben
 for i=1:length(p)
     q(:,1) = p(:,1)-center(1);
     q(:,2) = p(:,2)-center(2);
 end
hold on
fill(q(:,1),q(:,2),'g');
```

Listing 20: MATLAB-Code: Verschieben eines Rechtecks

Das rote Rechteck ist damit auf das grüne Rechteck verschoben. Nun drehen wir das grüne Rechteck um den angegebenen Winkel. Die Ortsvektoren der Ecken sind nun die Richtungsvektoren, die wir drehen. Die Drehung geschieht durch Multiplikation dieser Richtungsvektoren mit der Matrix

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

In MATLAB:

```
% Drehung um 30°:
 A = [cos(pi/6) -sin(pi/6);sin(pi/6) cos(pi/6)];
```

```
 for i=1:length(p)
     pd = (A*p')';
     qd = (A*q')';
 end
 fill(qd(:,1),qd(:,2),'m');
```

Listing 21: Drehung durch Multiplikation mit der Rotationsmatrix

Es entsteht das Magenta-Viereck. Dieses wird nun wieder an seinen ursprünglichen Ort verschoben, um das gelbe Viereck zu erhalten:

```
for i=1:length(p)
     qdv(:,1) = qd(:,1)+center(1);
     qdv(:,2) = qd(:,2)+center(2);
 end
 fill(qdv(:,1),qdv(:,2),'y');
```

Listing 22: Gedrehtes Viereck zurückverschieben

Dieses bauen wir in der Methode **`ausrichten()`** nach. Deren Ergebnis ist ein **Polygon** (Vieleck) mit den Ecken des gedrehten Vierecks:

```
Polygon ausrichten() {
            // Drehen um den Winkel richtung
            // ursprüngliche Ecken: (0,0), (b,0), (b,h), (0,h)
            // Mittelpunkt: (b/2, h/2)
            // Vektoren von der Mitte zu den Eckpunkten:
            double b = d.width;
            double h = d.height;
            double [][] ecken = {{-b/2,-h/2},{ b/2,-h/2}, { b/2, h/2},
                                 {-b/2, h/2}};

            double [][] eckenRot = new double[4][2];

            for (int i=0;i<4;i++) {
                  eckenRot[i][0]=Math.cos(richtung)*ecken[i][0]-
                                 Math.sin(richtung)*ecken[i][1];
                  eckenRot[i][1]=Math.sin(richtung)*ecken[i][0]+
                                 Math.cos(richtung)*ecken[i][1];
            }
            for (int i=0;i<4;i++) {
                  eckenRot[i][0]+=position.x;
                  eckenRot[i][1]+=position.y;
            }

            Polygon p = new Polygon();
            for (int i=0;i<4;i++)
                  p.addPoint((int)eckenRot[i][0], (int)eckenRot[i][1]);
            return p;
      }
```

Listing 23: Methode ausrichten()

Das so erhaltene Polygon zeichnen wir wie folgt:

```
void zeichnen(Graphics g, Polygon p) {
      g.setColor(farbe);
      g.fillPolygon(p);

      Graphics2D g2 = (Graphics2D)g;
      g2.setStroke(new BasicStroke(5));
      g2.setColor(Color.DARK_GRAY);
      // Vorderseite durch dicke Linie kennzeichnen:
      g2.drawLine(p.xpoints[1],p.ypoints[1],
                  p.xpoints[2],p.ypoints[2]);
}
```

Listing 24: Roboter zeichnen

Die dicke Linie, welche die Punkte mit Index 1 und 2 verbindet, kennzeichnet die Vorderseite des Roboters.
Um den Roboter auf unserem Grundstück darzustellen, fügen wir der Grundstück-Klasse ein Member-Objekt `robi` vom Typ `Robi` hinzu, erzeugen dieses im Konstruktor der Grundstück-Klasse:

```
            robi = new Robi(0.1, Math.PI/6, 300, 300, Color.ORANGE,70,50);
```

Dabei verwenden wir folgenden Robi-Konstuktor:

```
      Robi(double maxV, double winkel, int posX, int posY, Color f,
            int b, int h) {
            maxStep = maxV;
            richtung = winkel;
            farbe = f;
            position = new Point(posX,posY);
            d = new Dimension(b,h);
      }
```

Listing 25: Robi-Konstruktor

Nun ergänzen wir die **`paintComponent()`**-Methode durch folgende Zeilen:

```
            Polygon p = robi.ausrichten();
            robi.zeichnen(g, p);
```

Listing 26: Zeichnen des Roboters

Um das Fahren zu testen, fügen wir in der GUI-Klasse einen Button „Fahren“ hinzu. Klickt man diesen an, so soll Folgendes passieren:

```
class FahrenAL implements ActionListener {
          public void actionPerformed(ActionEvent e) {
                grund.robi.fahren(30,20);
                repaint();
          }
     }
```

Listing 27: Bewegen des Roboters

Da es sich hier um einen Mähroboter handeln soll, soll der bisher gemähte Rasen sichtbar werden. Dazu zeichnen wir alle Polygone, die der Roboter bisher besucht hat. Um dies zu ermöglichen, legen wir in der Grundstückklasse eine neue Membervariable an, in der wir alle Polygone sammeln:

```
Vector<Polygon> gemaeht;
```

Dieses Element wird im Konstruktor initialisiert. In der **`paintComponent`**-Methode sammeln wir nun alle Polygone, die vom Roboter bisher „gemäht“ wurden:

```
          Polygon p = robi.ausrichten();
           gemaeht.add(p);

          g.setColor(new Color(0,200,100));
          for (int i=0;i<gemaeht.size();i++)
                g.fillPolygon(gemaeht.elementAt(i));

          robi.zeichnen(g, p);
```

Listing 28: Speichern der bisher "gemähten" Polygone

4 Autonomes Fahren: Die Klasse Thread

Um den Roboter nun für eine bestimmte Zeit **selbstständig** fahren zu lassen, könnte man auf die naheliegende Idee kommen, einfach eine Schleife in den **`ActionListener`** für Button „Fahren“ einzubauen. Der Effekt einer solchen Schleife ist jedoch, dass wir nur das Endergebnis von z.B. 100 Schritten, aber keine Zwischenschritte sehen. Der Button und die gesamte Anwendung sind so lange blockiert, bis die Schleife abgearbeitet ist.

Der Grund für dieses Verhalten liegt in der Aufteilung der Rechenzeit innerhalb unserer Anwendung. Um zwischendurch zu zeichnen, muss die Ereignisbearbeitung von Swing Rechenzeit erhalten. Das geschieht aber nicht, solange wir uns in der Funktion **`actionPerformed()`** befinden.

Hier kommt das sog. ***Multithreading*** ins Spiel, welches bereits in der ersten Java-Version unterstützt wurde.

4.1 Multithreading

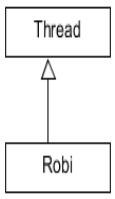
Sie kennen sicher den Begriff ***Multitasking***, mit dem die „parallele“ Ausführung mehrerer Programme gemeint ist. Dieses Leistungsmerkmal wird insbesondere benötigt, wenn mehrere Benutzer ein System gleichzeitig nutzen. Dabei werden mehrere Programme quasi gleichzeitig ausgeführt, können sich aber nicht gegenseitig beeinflussen, da die Datenbereiche der ausgeführten Programme vollständig getrennt sind.

Ein ***Mutlithreading***-System bietet dagegen die Möglichkeit, verschiedene Teile **eines Programms** parallel ablaufen zu lassen, wobei alle Teile auf die gemeinsamen Daten des Programms zugreifen können.

Die Klasse ***Thread*** macht genau dieses möglich. Jeder Thread verfügt über die Methode **`run()`**, die typischerweise in einer Endlosschleife die erforderlichen Aktivitäten ausführt. Mit **`sleep()`** können wir Verzögerungen erzeugen und die Ausführung des aktuellen Thread anhalten. Damit geben wir anderen Threads Gelegenheit zur Ausführung.

Die Methode **`run()`** wird nicht direkt aufgerufen, sondern am Anfang (meist im Konstruktor) mit der Methode **`start()`** gestartet.

Um nun unserem Roboter das selbstständige Fahren zu ermöglichen, beerben wir einfach die Klasse **`Thread`:**



Listing 29: Robi als Kindklasse von Thread

Das erreichen wir, indem wir bei der Klassendekalration „**`extends Thread`**“ ergänzen:

```
public class Robi extends Thread
```

Nun überschreiben wir die Methode **`run()`**. Um das Fahren aktivieren und deaktivieren zu können, führen wir eine Membervariable **`doMove`** vom Typ **`boolean`** ein, die wir im **`ActionListener`** unseres „*Fahren*“-Buttons aus-und einschalten. Hat die Variable den Wert **`false`**, so bleibt der Roboter stehen. Ansonsten bewegt er sich mit den angegebenen Geschwindigkeiten. Um ferner aus dem Thread heraus zu ermöglichen, den Roboter auf dem Grundstück zu zeichnen, ergänzen wir eine weitere **Membervariable vom Typ `Grundstueck`**, in der wir das Handle auf unser Grundstück übergeben. Ferner starten wir im Konstruktor den Thread:

```
boolean doMove;
Grundstueck grund;
```

```
Robi(Grundstueck grd, double maxV, double winkel, int posX,
        int posY, Color f, int b, int h) {
      maxStep = maxV;
      richtung = winkel;
      farbe = f;
      position = new Point(posX,posY);
      d = new Dimension(b,h);
      doMove = false;
      start();
      grund = grd;
}
```

Listing 30: neue Membervariablen und deren Initialisierung

Diesen Konstruktor rufen wir nun in der Grundstück-Klasse wie folgt auf:

```
robi = new Robi(this, 0.1, Math.PI/6, 300, 300,
                  Color.ORANGE,70,50);
```

Die run-Methode ist zunächst wie folgt implementiert:

```
public void run() {
      while (true) {
            try {
                  sleep(20);
                  if (!doMove)
                        continue;
                  fahren(20, 10);
                  grund.repaint();

            } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
}
```

Listing 31: run()-Methode

Den ActionListender für unseren Fahren-Button bauen wir wie folgt um:

```
public void actionPerformed(ActionEvent e) {
       grund.robi.doMove=!robi.doMove;
}
```

Damit halten wir den Roboter an, wenn er fährt. Andernfalls veranlassen wir, dass er weiterfährt.

4.2 Fernsteuern

Nun führen wir in der Klasse Robi zwei Membervariable **vLinks** und **vRechts** vom Typ **int** ein, in denen wir die aktuelle Motorgeschwindigkeit speichern und initialisieren die beiden Variablen im Konstruktor von Robi.
Den **fahren()**-Aufruf in Listing 31 ersetzen wir durch

```
fahren(vLinks, vRechts);
```

In der GUI-Klasse ergänzen wir zwei Membervariable vom Typ Schiebregler (**JSlider**):

```
JSlider spdLinks, spdRechts;
```

Im Konstruktor der Grundstück-Klasse erzeugen wir diese Schieberegler und fügen **ChangeListener** hinzu, in denen wir bei Betätigung des entsprechenden Reglers die jeweilige Geschwindigkeit ändern:

```
spdLinks = new JSlider();
spdLinks.setBorder(new TitledBorder("links"));
rechts.add(spdLinks);
spdRechts = new JSlider();
spdRechts.setBorder(new TitledBorder("rechts"));
rechts.add(spdRechts);

spdLinks.addChangeListener(new LCL());
spdRechts.addChangeListener(new RCL());
```

Listing 32: Schiebregler erzeugen und anordnen

Die ***ChangeListener*** übernehmen einfach den aktuellen Sliderwert (dazu mussten die Membervariablen angelegt werden) und setzen die Motorgeschwindigkeit entsprechend:

```
class LCL implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
        grund.robi.vLinks = spdLinks.getValue();
    }
}

class RCL implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
        grund.robi.vRechts = spdRechts.getValue();
    }
}
```

Listing 33: ChangeListener

Nun lässt sich der Roboter interaktiv mit den beiden Schiebereglern steuern.

4.3 Roboter mit Sensor, autonomes Fahren

Unser Robotor soll nun mit einem Sensor ausgestattet werden, welcher die Entfernung zum Grundstücksrand misst und so in der Lage sein, selbstständig das gesamte Grundstück zu mähen, indem er eigene Entscheidungen trifft, wenn er in die Nähe der Grundstücksgrenze kommt. Das ist ein erster Schritt in Richtung autonomes Fahren, denn nun muss unser Roboter eingene Entscheidungen treffen.

Der Sensor soll mittig vorne angebracht sein.

Da der Roboter alles können soll, was der bisherige Roboter kann, legen wir die neue Klasse einfach als Kindklasse von Robi an:

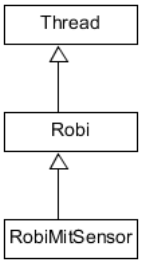


Abbildung 8: Klassenhierarachie für den Roboter

Die aktuelle Position des Sensors ergänzen wir in der neuen Klasse als Membervariable **sensor** vom Typ **`Point`** an.

Im Konstruktor rufen wir den Konstruktor der Elternklasse auf und initialisieren die neue Variable sowie die beiden Motorgeschwindigkeiten:

```
RobiMitSensor(Grundstueck gs, double maxV, double winkel, int posX,
    int posY, Color f, int b, int h) {
            super(gs, maxV, winkel, posX, posY, f, b, h);
            vLinks = 25;
            vRechts = 25;
            sensor = new Point(posX,posY);
}
```

Listing 34: Konstruktor

Wir überschreiben die **zeichnen()**-Methode wie folgt:

```
void zeichnen(Graphics g, Polygon p) {
            super.zeichnen(g, p);
            sensor.x = (p.xpoints[1]+p.xpoints[2])/2;
            sensor.y =  (p.ypoints[1]+p.ypoints[2])/2;
            g.setColor(Color.RED);
            g.fillOval(sensor.x-d.height/10, sensor.y-d.height/10,
                d.height/5,d.height/5);
     }
```

Listing 35: Neue zeichnen()-Methode

Zunächst rufen wir die `zeichnen()`-Methode der Elternklasse auf. Dann ergänzen wie einen kleinen ausgefüllten Kreis um die Sensorposition.

Damit der Sensor die Entfernung zur Grundstückgrenze bestimmen kann, ergänzen wir eine Methode **`getDistance()`**:

```
double getDistance() {
      double dist = 0;
      for (int i=0;i<grund.grenze.size();i++) {
            Point p = grund.grenze.elementAt(i);
            Double d = Math.sqrt((p.x-sensor.x)*(p.x-sensor.x)+
                        (p.y-sensor.y)*(p.y-sensor.y));
            if (i==0)
                  dist = d;
            else
                  if (d < dist)
                        dist = d;
      }
      return dist;
}
```

Listing 36: Bestimmung der Entfernung zur Grenze

Unser Ziel ist, dass der Roboter die gesamte Rasenfläche mäht. Dafür gibt es verschiedene Strategien. Wir implementieren eine zufallsbasierte Strategie:

Der Roboter fährt, bis er in die Nähe der Grenze kommt. Dann fährt er in fast entgegengesetzter Richtung weiter. Die Abweichung von der entgegengesetzten Richtung wird zufällig bestimmt.

Eine neue Richtung wird für mindestens 500 ms beibehalten. Diese Strategie setzen wir um, indem wir die **`run()`**-Methode überschreiben:

```
public void run() {
      long zeit1=-1;
      while (true) {
            try {
                  sleep(20);
                  if (!doMove)
                        continue;
                   double dist = getDistance();
                  if (dist < 10) {
                        if (zeit1 < 0) {
                              double richtungNeu=0, cos1=0, cos2=0;
                              double z = Math.random();
                              richtung += Math.PI+(z-0.5)*Math.PI/10;
                              zeit1= System.currentTimeMillis();
                        }
```

```
                    else
                      if (System.currentTimeMillis()-zeit1 > 500)
                         zeit1 = -1;
                }
                fahren(vLinks,vRechts);
                grund.repaint();

               } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
               }
          }
     }
```

Listing 37: run()-Methode mit Random-Strategie

Natürlich lassen sich in der `run()`-Methode auch andere Strategien implementieren.

In manchen Situationen überfährt der Roboter die Grenze und verlässt das Grundstück.

4.3.1 Erweiterungsmöglichkeiten

Man könnte sich nun andere Mähstrategien überlegen oder den Roboter mit weiterer Sensorik ausstatten. Hat man z.B. vorne links und vorne rechts einen Sensor zur Entfernungsmessung, so kann man bestimmen, ob die linke oder die rechte Seite näher an der Grenze ist und entsprechend reagieren. In der Klassenhierarchie würde man so einen Roboter nicht als Kindklasse der eben angelegten Klasse **`RobiMitSensor`** anlegen, sondern ebenfalls als Kind der **`Robi`**-Klasse, da es sich beim **`RobiMitZweiSensoren`** nicht um eine Spezialisierung von **`RoboterMitSensor`** handelt.

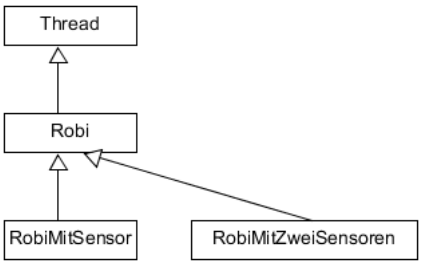


Abbildung 9: Klassenhierarchie für verschiedene Robotervarianten

Eine weitere Erweiterungsmöglichkeit wäre die Berücksichtigung eines Akkus und einer Ladestation: Je nach Energieverbrauch, welcher von der Mähdauer und den Motorgeschwindigkeiten abhängt, wird der Akku entladen. Erreicht die Ladung eine kritische Untergrenze, so muss der Roboter die Ladestation aufsuchen. Deren Position könnte beispielsweise durch eine besonders helle Lichtquelle lokalisiert werden. Ist der Roboter mit einem Helligkeitssensor ausgestattet, so könnte er die Richtung ausfindig machen, in welcher die Helligkeit am größten ist und sich so der Ladestation annähern.

Das folgende Diagramm zeigt die Beziehungen der wichtigsten Klassen und Objekte unserer Anwendung.

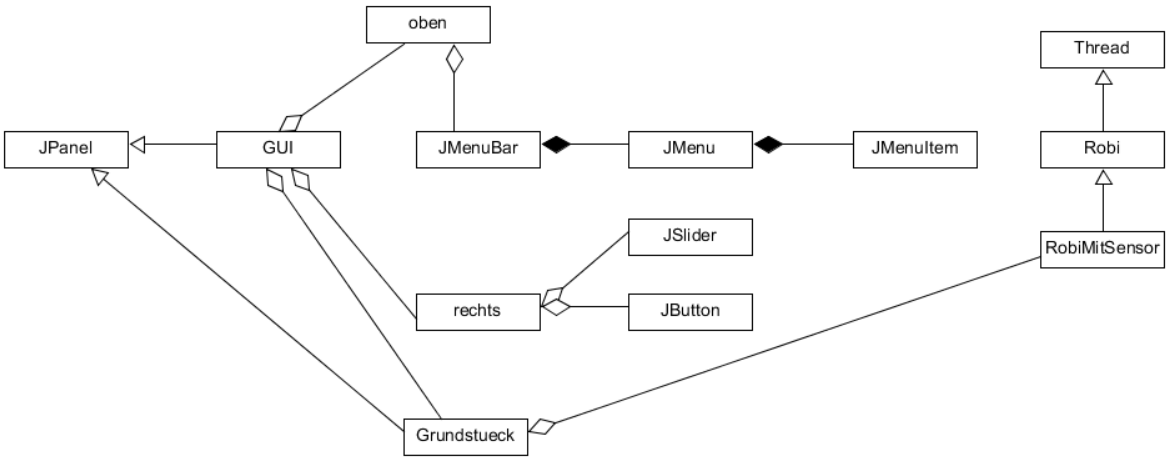


Abbildung 10: Beziehungen unter den verwendeten Klassen