Universität Augsburg

Fakultät für Angewandte Informatik
Lehrprofessur für Informatik
Prof. Dr. Robert Lorenz

SS 2012

23.05.2012

Betreutes Programmieren Vorlesung Informatik II, Blatt 5

Grafischer ToDoManager

In der letzten Woche haben Sie ein Datenmodell für einen ToDoManager erstellt. Heute werden Sie eine grafische Anbindung programmieren, um den ToDoManager auch wirklich einsetzen zu können. Die GUI soll mit AWT erstellt werden, so wie Sie es in der Vorlesung gesehen haben.

Als Erinnerungshilfe erhalten Sie zunächst nocheinmal das UML-Diagramm der letzten Woche.

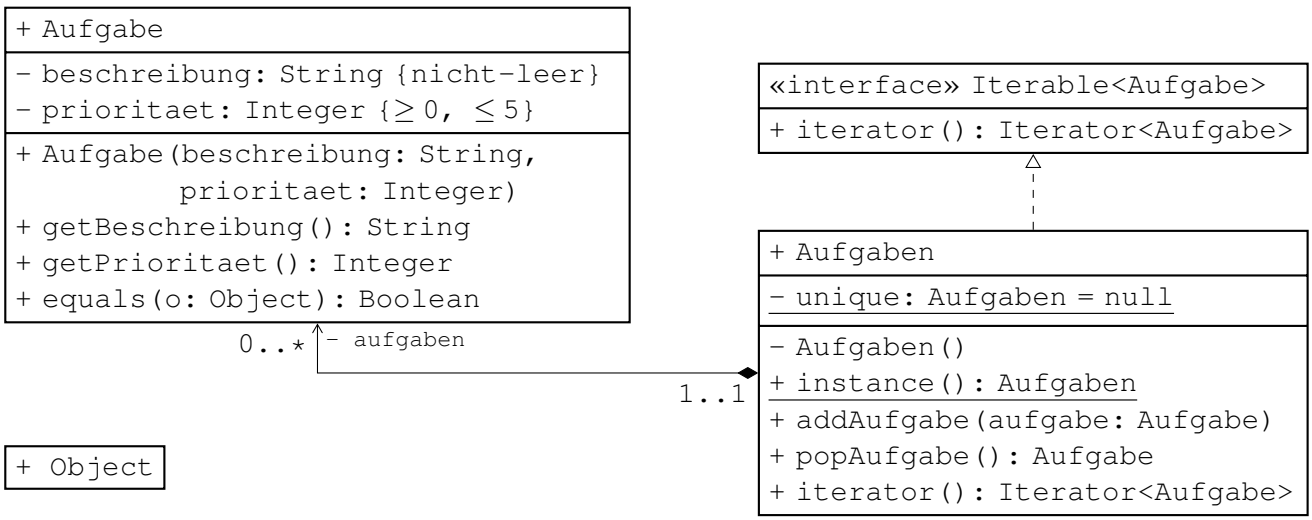

Abbildung 1: UML-Klassendiagramm der Datenkomponenten

Es ist Ihnen freigestellt, Ihren eigenen Code oder den Code auf den nächsten Seiten zu benutzen.

Zunächst die Klasse **`Aufgabe`**: Die Einschränkungen der beiden Attribute werden im Konstruktor überprüft und ggf. werden Standardwerte benutzt.

```
public class Aufgabe
{
  private String beschreibung;
  private int prioritaet;

  public Aufgabe(String beschreibung, int prioritaet)
  {
    this.beschreibung = beschreibung;
    if (beschreibung == null || beschreibung.length() == 0)
      this.beschreibung = "(keine Beschreibung)";

    this.prioritaet = prioritaet;
    if (prioritaet > 5) this.prioritaet = 5;
```

```
    if (prioritaet < 0) this.prioritaet = 0;
  }

  public String getBeschreibung()
  {
    return beschreibung;
  }

  public int getPrioritaet()
  {
    return prioritaet;
  }

  public boolean equals(Object o)
  {
    if (o instanceof Aufgabe && prioritaet == ((Aufgabe) o).prioritaet)
      return beschreibung.equalsIgnoreCase(((Aufgabe) o).beschreibung);

    return false;
  }
}
```

Die Klasse **AufgComp** wird benötigt, um **Aufgabe**n in eine Ordnung zu bringen. Auf diese Weise können die Container von Java intern alle Objekte sortiert vorhalten, wobei die Reihenfolge der Sortierung vom Programmierer abgeändert werden kann. Damit **AufgComp** dazu eingesetzt werden kann, muss die Schnittstelle **Comparator** – besser **Comparator<Aufgabe>** – implementiert werden. Die Methode **compare()** bekommt dann zwei **Aufgabe**n übergeben und soll entscheiden, ob beide Objekte gleich sind (Rückgabewert 0), das erste Objekt vor das zweite einsortiert werden soll (negativer Rückgabewert) oder andersherum (positiver Rückgabewert).

```
import java.util.Comparator;

public class AufgComp implements Comparator<Aufgabe>
{
  public int compare(Aufgabe a1, Aufgabe a2)
  {
    if (a1.getPrioritaet() == a2.getPrioritaet())
      return a1.getBeschreibung().compareToIgnoreCase(a2.getBeschreibung());

    return a2.getPrioritaet() - a1.getPrioritaet();
  }
}
```

In der Klasse **Aufgaben** können nun die Früchte der Vorarbeit geerntet werden, indem als Container die Klasse **PriorityQueue** benutzt wird. So eine Warteschlange sortiert Objekte nach einer Priorität und stellt den Zugriff auf das dringlichste Objekt bereit. Um den Mechanismus nun benutzen zu können, wird ein Objekt der Klasse **AufgComp** dem Konstruktor von **PriorityQueue** übergeben. Der Rest läuft von selbst.

```
import java.util.Iterator;
import java.util.PriorityQueue;

public class Aufgaben implements Iterable<Aufgabe>
{
  private static Aufgaben unique = null;

  private Aufgaben()
  {
    aufgaben = new PriorityQueue<Aufgabe>(10, new AufgComp());
  }

  public static Aufgaben instance()
  {
    if (unique == null) unique = new Aufgaben();
```

```
    return unique;
  }

  PriorityQueue<Aufgabe> aufgaben;

  public void addAufgabe(Aufgabe aufgabe)
  {
    if (!aufgaben.contains(aufgabe))
      aufgaben.offer(aufgabe);
  }

  public Aufgabe popAufgabe()
  {
    return aufgaben.poll();
  }

  public Iterator<Aufgabe> iterator()
  {
    return aufgaben.iterator();
  }
}
```

Aufgabenstellung

Setzen Sie das UML-Klassendiagramm aus Abbildung 2 in Java-Code um und berücksichtigen Sie dabei die weiteren Beschreibungen nach dem Diagramm. Auch wenn Ihnen noch nicht alle Elemente des Diagramms klar sein sollten, sollten Sie in der Lage sein, den Code gewissermaßen als Grundgerüst aufzuschreiben.

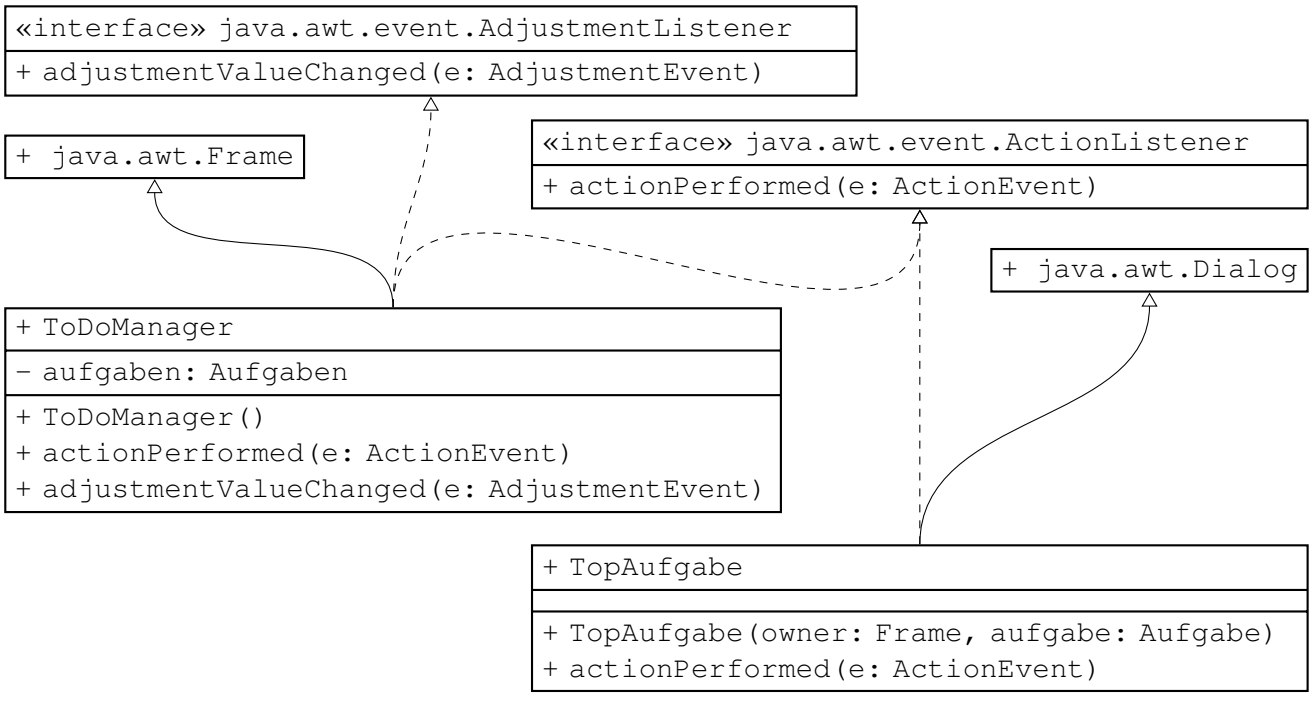

Abbildung 2: UML-Klassendiagramm der GUI-Komponenten.

Im Hauptfenster der GUI-Anbindung soll es möglich sein, Aufgaben hinzuzufügen. Dazu soll zum Einen eine Beschreibung in ein Eingabefeld eingegeben werden können und zum Anderen soll mittels eines Schiebereglers eine Priorität festgelegt werden können. Danach soll ein Klick auf eine Schaltfläche mit der Aufschrift `"eintragen"` die neue Aufgabe in das System einfügen. Neben dem Eingabefeld soll der Schriftzug `"Beschreibung: "` stehen, neben dem Schieberegler der Text `"Priorität: #"`, wobei statt des Gartenzauns die eingestellte Priorität eingesetzt sein soll. Eine weitere Schaltfläche mit der Aufschrift `"Aufgabe!!!"`

soll ein zweites Fenster öffnen, in dem die Beschreibung der nächsten Aufgabe mit der höchsten Priorität angezeigt werden soll. Eine Schaltfläche mit der Aufschrift `"Erledigt!"` soll das zweite Fenster wieder schließen. Falls keine Aufgabe im System vorhanden sein sollte, soll im Fenster der Text `"Nichts zu tun!"` erscheinen sowie die Schaltfläche die Aufschrift `"Juhu!"` tragen. Die erledigte Aufgabe soll aus dem System gelöscht werden.

Hilfestellungen

Allgemeine Erklärungen sind bereits auf den letzten beiden Blättern enthalten. Hier erhalten Sie nur Beispiel-Code für die Programmierung von grafischen Oberflächen mit Java AWT.

Zunächst wird das Hauptfenster der Anwendung erzeugt. Das Grundgerüst der Klasse kann als allgemeine Schablone für GUI-Fenster dienen.

Da es sich um das Hauptfenster für eine Anwendung handelt, wird die Klasse **`ToDoManager`** von der AWT-Klasse **`Frame`** abgeleitet. Um dem Fenster einen Titel zu verpassen, wird der Konstruktor der Klasse **`Frame`**, der eine Zeichenkette als Parameter erwartet, benutzt.

Sind eigene Initialisierungen nötig, können diese nun erledigt werden.

Danach wird ein passendes Layout gewählt und alle GUI-Elemente werden erzeugt und im Fenster platziert. Wenn das Fenster über Ereignisbehandlung verfügen soll, muss es dazu auch angemeldet sein. Dazu dienen die verschiedenen **`add...Listener()`**-Methoden.

Als nächstes wird die Größe des Fensters eingestellt. Eine passende Größe kann von AWT automatisch ermittelt werden, wozu die Methode **`pack()`** zum Einsatz kommt. Schließlich wird das Fenster schließbar gemacht, sodass die Applikation auch beendet werden kann. Abschließend wird das Fenster sichtbar gemacht, womit es auf dem Bildschirm angezeigt wird und auch die Ereignisbehandlung beginnt. Das Sichtbarmachen übernimmt die Methode **`setVisible()`**.

Um auf auftretende Ereignisse reagieren zu können, werden sog. Listener eingesetzt. In der Regel wird das Fenster selbst dazu bestimmte Schnittstellen – z. B. **`ActionListener`** – implementieren, sodass neben den eigenen Methoden auch Methoden zur Ereignisbehandlung in die Klasse aufgenommen werden müssen.

Um alle Klassen und Schnittstellen zur Verfügung zu haben, müssen sie am Anfang der Datei importiert werden, sodass die Schablone nun folgendes Aussehen hat:

⟨*notwendige Importe, v. a. der verwendeten AWT-Klassen und Schnittstellen*⟩

```
public class ToDoManager extends Frame ⟨benötigte Schnittstellen⟩
{
  ⟨benötigte Attribute, insbesondere AWT-Elemente⟩

  public ToDoManager()
  {
    super("ToDoManager");

    ⟨eigene Initialisierungen⟩

    ⟨Layout wählen⟩
    ⟨GUI-Elemente erzeugen und hinzufügen⟩
    ⟨Fenster zur Ereignisbehandlung anmelden⟩
    pack();

    ⟨Fenster schließbar machen⟩

    setVisible(true);
  }
```

```
    ⟨benötigte Methoden, insbesondere Ereignisbehandlungsmethoden⟩
}
```

Um ein Fenster schließbar zu machen, gibt es mehrere Möglichkeiten. Will man ein Fenster schließen, z. B. durch einen Klick auf eine entsprechende Schaltfläche, so wird ein Ereignis vom Typ **`WindowEvent`** ausgelöst, auf das das Fenster dann reagieren kann. Um über das Ereignis informiert zu werden, muss die Schnittstelle **`WindowListener`** implementiert werden. Da diese Schnittstelle jedoch mehrere Methoden verlangt, wird hier ein einfacherer Weg gewählt: Über den **`WindowAdapter`**.

Es wird ein anonymes Objekt[1] erzeugt, das von der Klasse **`WindowAdapter`** abgeleitet ist und bei dem die Methode **`windowClosing()`** geeignet überschrieben wird. Nachdem das Objekt über **`addWindowListener()`** als **`WindowListener`** bekannt gemacht wurde, kümmert es sich fortan um das Schließen des Fensters mittels **`dispose()`** und das Beenden der Anwendung durch **`System.exit()`**.

```
addWindowListener(new WindowAdapter()
{
  public void windowClosing(WindowEvent e)
  {
    dispose();
    System.exit(0);
  }
});
```

Für die Wahl des Layout stellt AWT eine ganze Reihe an Klassen zur Verfügung. Es lohnt sich, die Dokumentation ein wenig zu durchstöbern. Für den ToDoManager empfiehlt sich der Einfachheit halber ein **`GridLayout`** mit drei Zeilen und zwei Spalten. Zusätzlich gibt es jeweils fünf Pixel Zwischenraum in horizontaler und vertikaler Richtung.

```
setLayout(new GridLayout(3, 2, 5, 5));
```

Die Elemente, aus denen die Oberfläche aufgebaut ist, werden von links nach rechts und von oben nach unten im Fenster angeordnet. Die beiden Texte werden als **`Label`** realisiert. Auf den Text `"Beschreibung"` muss nach der Erstellung nicht mehr zugegriffen werden, sodass das betreffende Objekt nicht als Attribut gespeichert wird. Die Zeichenkette `"Priorität: #"` muss immer wieder geändert werden, sodass das **`Label`**-Objekt als Attribut gespeichert wird. Auf das Eingabefeld und den Schieberegler muss ebenfalls wieder zugegriffen werden, weswegen zwei weitere Attribute hinzukommen. Die beiden Schaltflächen werden lediglich benötigt, um das Fenster bei ihnen zur Ereignisbehandlung anzumelden; also keine weiteren Attribute; außer **`aufgaben`**, das als Abkürzung dienen soll, um nicht immer über die statische **`instance()`**- Methode auf das Singleton-Objekt zugreifen zu müssen. So kommt also diese Liste an Attributen zustande.

```
private Aufgaben aufgaben;
private TextField beschreibung;
private Label prioritaet;
private Scrollbar schieber;
```

Das Attribut **`aufgaben`** wird im Konstruktor einmalig initialisert.

```
aufgaben = Aufgaben.instance();
```

Mit den oben erwähnten Vorgaben gestaltet sich nun die Erstellung der grafischen Oberfläche wie folgt:

```
add(new Label("Beschreibung:"));
beschreibung = new TextField(40);
add(beschreibung);
```

[1] Ein Objekt, das nicht über eine Variable ansprechbar ist und gewissermaßen keinen Namen besitzt.

```
prioritaet = new Label("Priorität: 0");
add(prioritaet);
schieber = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 6);
add(schieber);

Button eintragen = new Button("eintragen");
add(eintragen);

Button aufgabe = new Button("Aufgabe!!!");
add(aufgabe);
```

Was genau die einzelnen Parameter in den verschiedenen Konstruktoren bedeuten und welche anderen Konstruktoren die verwendeten AWT-Klassen haben, können Sie in der Dokumentation nachschlagen.

Wie schon angekündigt, wird das Fenster selbst als Ereignisbehandler bei den Schaltflächen und dem Schieberegler angemeldet.

```
eintragen.addActionListener(this);
aufgabe.addActionListener(this);
schieber.addAdjustmentListener(this);
```

Damit dies möglich wird, muss **ToDoManager** die passenden Schnittstellen implementieren, die – durch Komma getrennt – zusammen mit dem Schlüsselwort **implements** an die Signaturzeile der Klasse angehängt werden.

```
implements ActionListener,
           AdjustmentListener
```

Die Methoden, die sich um die Ereignisbehandlung kümmern, werden der Klasse ebenfalls hinzugefügt. Vom Schieberegler erhält das Fenster ein Ereignis gesandt, das über seine **getValue()**-Methode Zugriff auf den aktuellen Wert des Reglers gestattet. Dieser Wert wird zusammen mit der Zeichenkette "Priorität: " als neuer Text von **prioritaet** gesetzt.

```
public void adjustmentValueChanged(AdjustmentEvent e)
{
  prioritaet.setText("Priorität: " + e.getValue());
}
```

Bei den Schaltflächen muss zuerst geklärt werden, von welcher der beiden das Ereignis gesandt wurde. Abhängig davon wird eine neue Aufgabe erzeugt und dem Container hinzugefügt oder das zweite Fenster geöffnet.

```
public void actionPerformed(ActionEvent e)
{
  if (e.getActionCommand().equals("eintragen"))
  {
    aufgaben.addAufgabe(new Aufgabe(beschreibung.getText(),
                                    schieber.getValue()));

    beschreibung.setText("");
    schieber.setValue(0);
    prioritaet.setText("Priorität: 0");
  }
  else new TopAufgabe(this, aufgaben.popAufgabe());
}
```

Alles, was jetzt noch fehlt, sind die Importe aller verwendeten Klassen und Schnittstellen am Anfang der Datei.

```
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.TextField;
import java.awt.Button;
import java.awt.Scrollbar;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.AdjustmentListener;
import java.awt.event.AdjustmentEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

Um den grafischen ToDoManager später auch starten zu können, wird eine **main()**-Funktion in der Klasse **ToDoManager** untergebracht. Ihre einzige Aufgabe ist es, ein neues Fenster zu erzeugen. Der Konstruktoraufruf kehrt erst zurück, wenn das Fenster geschlossen wird.

```
public static void main(String[] args)
{
  new ToDoManager();
}
```

Die Erstellung des zweiten Fensters folgt einer ähnlichen Schablone. Diesmal handelt es sich nicht um das Hauptfenster einer Anwendung, sondern um ein zusätzliches Fenster. Solche Fenster werden von der AWT-Klasse **Dialog** abgeleitet. Dialogfenster können in zwei verschiedenen Arten auftreten: *modal* und *nicht-modal*. Ein modales Dialogfenster blockiert den Rest des Programms, das erst fortgesetzt wird, wenn dieses Fenster wieder geschlossen ist. Ein nicht-modales Dialogfenster blockiert die Anwendung nicht, und es kann zwischen den Fenstern gewechselt werden. Die Klasse **Dialog** besitzt einen Konstruktor, der als dritten Parameter die Modalität als boolesche Variable kodiert erwartet. Der zweite Parameter ist eine Zeichenkette, die als Titel benutzt wird und als ersten Parameter wird der Eigentümer übergeben. Der Eigentümer ist das Fenster, zu dem gewechselt werden soll, wenn der Dialog geschlossen wird.

In dieser Schablone wurde auf Attribute, eigene Initialisierungen und das Schließbarmachen des Fensters verzichtet, da diese Elemente in der Implementierung der Klasse keine Rolle spielen. Für eigene Klassen wird das sicher anders sein.

⟨*benötigte Importe, v. a. der AWT-Klassen*⟩

```
public class TopAufgabe extends Dialog ⟨notwendige Schnittstellen⟩
{
  public TopAufgabe(Frame owner, Aufgabe aufgabe)
  {
    super(owner, "Aufgabe!!!", true);

    ⟨Layout setzen⟩
    ⟨GUI-Elemente erzeugen und platzieren⟩
    ⟨Fenster als Ereignisbehandler anmelden⟩
    pack();

    setVisible(true);
  }

  ⟨benötigte Methoden, insbesondere zur Ereignisbehandlung⟩
}
```

Das Layout für dieses Fenster ist einspaltig und besteht aus zwei Zeilen, wobei wieder fünf Pixel Abstand eingefügt werden.

```
setLayout(new GridLayout(2, 1, 5, 5));
```

Die Oberfläche selbst besteht in der oberen Zeile aus der Beschreibung der übergebenen Aufgabe und in der unteren Zeile aus einer Schaltfläche, mit der das Fenster wieder geschlossen werden kann. Falls die übergebene Aufgabe `null` sein sollte, ist nichts im Container vorhanden und also auch nichts zu tun.

```
if (aufgabe != null) add(new Label(aufgabe.getBeschreibung()));
else add(new Label("Nichts zu tun!"));

Button erledigt;
if (aufgabe != null) erledigt = new Button("Erledigt!");
else erledigt = new Button("Juhu!");
add(erledigt);
```

Wenn die Schaltfläche gedrückt wird, soll sich das Fenster wieder schließen. Also muss das Fenster bei ihr als Ereignisbehandler angemeldet werden.

```
erledigt.addActionListener(this);
```

Dazu muss auch die richtige Schnittstelle implementiert werden.

```
implements ActionListener
```

Und auch die entsprechende Methode geschrieben werden, die in diesem Fall die einzige Aufgabe hat, das Dialogfenster zu schließen.

```
public void actionPerformed(ActionEvent e)
{
  dispose();
}
```

Die vielen verwendeten Klassen und Schnittstellen werden wieder am Anfang der Datei importiert.

```
import java.awt.Dialog;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.Button;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

Nun hat man einen kompletten ToDoManager mit grafischer Oberfläche. Juhu!!