Tutorial: Post-It–Anwendung

In diesem Tutorial werden wir lernen wie man mit der Programmiersprache Java und der in Java integrierten Bibliothek[1] `javax.swing.*` ein Fenster erstellt mit welchem man Text formatieren und darstellen kann.

1 GUI

Als GUI (Abk. GUI von graphical user interface) wird die Benutzerschnittstelle eines computerprogramms bezeichnet, mit der ein Mensch mittels Eingabegerät-en (z.B. Maus oder Touchscreen) interagiert. Für ein Programm, das eine grafische Oberfläche bekommen soll, macht es Sinn sich zuallererst Gedanken darüber machen, was die Hauptfunktionen der Anwendung sind und wie diese auf einer graphischen Benutzeroberfläche dargestellt werden können. Die dabei entstehenden Skizzen nennt man auch „Mock-Ups“[2]. Aus den Mock-ups leitet man anschließend nicht-funktionalen Anforderungen, die das Aussehen der fertigen Anwendung beschreiben ab.

1.1 Ein erstes Mock-up

Für unser Programm, die Post-It–Anwendung steht ganz klar die Schreibfläche, also das eigentliche Post-It im Mittelpunkt. Damit die Anwender die Schreibfläche auch als Post-It erkennen, sollte sie im klassischen gelb dargestellt werden (siehe Abbildung). Darauf soll man später seine Notizen schreiben können.

Da man den Text später auch formatieren können soll, fehlt uns nun aber noch eine Werkzeugleiste, wie wir sie aus Programmen wie Word (Microsoft) oder Writer (Open-/LibreOffice) kennen. Wie üblich ordnen wir sie im oberen Bereich der Anwendung an, damit sie schnell gefunden werden kann.

Exemplarisch bauen wir auch gleich die drei wichtigsten Buttons „B“ (engl. bold = fett schreiben), „I“ (engl. italic = kursiv schreiben) und „U“ (engl. underlined = unterstrichen schreiben) sowie eine Dropdown-Element zum Auswählen der Schriftart ein, damit wir uns das ganze etwas besser vorstellen können.

Die Werkzeugleiste sieht schon ganz gut aus, was jetzt noch fehlt ist der letzte Schliff am Textbereich. Klar ist, dass wir viel Platz für die Kreativität der Schreibenden lassen müssen, aber evtl. gibt es noch die ein oder andere Information, die der computer besser weiß. Was könnte das sein? Wir könnten zum Beispiel rechts oben ein Datum einfügen. Außerdem wollen wir eine Überschrift haben, die „sticky“ ist, d.h. sie soll immer sichtbar sein, auch wenn der Text evtl. gescrollt werden muss.

Große Firmen wie Apple oder Google haben in den vergangenen Jahren sehr viel Geld investiert, um herauszufinden, wie eine graphische Benutzerschnittstelle gestaltet sein muss, damit Menschen sie *gerne* nutzen. Ein Ergebnis daraus war beispielsweise der Touchscreen des iPhones oder das skeuomorphe[3] Design

[1] Eine Programmbibliothek ist eine Sammlung von bereitgestellten Klassen und Funktionen, die einen bestimmten Zweck erfüllen. Im Falle von `javax.swing.*` das Darstellen einer graphischen Oberfläche für eine Programm.

[2] siehe dazu http://de.wikipedia.org/wiki/Mock-up

[3] Skeuomorphismus beschreibt die Strategie, sich im Software-Design an Gegenständen und Werkzeugen aus der realen Welt zu orientieren, um die Funktion von Programmen oder Systemen zu erklären.

Abbildung 1: Unser erstes Mockup-Element. Das Post-It, eine gelbe Fläche. Wirklich großartig, aber sind wir jetzt schon fertig?

des Betriebssystems iOS (das aber mittlerweile seit iOS 7 durch den sog. „Minimalismus“ abgelöst wurde).

Anforderungen Aus dem eben erstellten Mock-up können wir einen Teil der Anforderungen an die Anwendung ableiten, die wir hier noch einmal zusammenfassen:

- Die Graphische Oberfläche der Anwendung soll aus einem Fenster bestehen, das zwei Bereiche beinhaltet.
- Der Bereich oben repräsentiert die Werkzeugleiste der Anwendung und nimmt etwa 1/6 bis 1/5 des Fensters ein.
- Der größere Bereich darunter rerpäsentiert einen Notizzettel. Er soll eine gelbe Farbe besitzen. Weiterhin soll man Text eingeben können und diesen unterschiedlich(!) formatieren können.

1.2 Implementierung mit javax.swing

Die Programmbibliothek javax.swing (ab jetzt nur noch Swing genannt), ist eine Bibliothek, die im Java Developer Kit enthalten ist, die es relativ einfach ermöglicht Grafische Oberflächen mit Java code zu erzeugen. Neben Swing existieren noch einige weitere Bibliotheken, die Oberflächen erzeugen können, in diesem Beispiel möchten wir uns aber auf Swing beschränken.

Zunächst einmal müssen wir ein Fenster erzeugen, in welchem unsere Anwendung ihre Inhalte bereit stellt. Da wir später noch einige Anpassungen an der Klasse vornehmen wollen, erweitern wir die Klasse `javax.swing.JFrame` und erstellen eine eigene Klasse NotesFrame. Achtung: Immer dort, wo bisher im code Punkte stehen, kommt später noch etwas hinzu:

Abbildung 2: Unser Mockup-Element. Jetzt mit einem seperaten Bereich für die Werkzeugleiste.

```
public class NotesFrame extends JFrame {

//...

}
```

Um den code sauber zu trennen, schreiben wir uns noch eine Hauptklasse, in der die Anwendung (und der soeben erstellte Frame) gestartet wird:

```
public class NotesMain{

        static int x = 420;
        static int y = 320;

        public static void main(String[] args) {
                //...
                NotesFrame frame = new NotesFrame(//...);
                frame.setSize(x, y);
                frame.setVisible(true);
        }
}
```

Nachdem wir jetzt eine Haupt-Klasse und eine Frame-Klasse haben, können wir uns um das Layout kümmern. Da wir im Mock-up schon relativ klar definiert haben, welche GUI-Bereiche es in der Anwendung gibt, müssen wir sie nun noch im code implementieren. Um uns die Arbeit etwas zu erleichtern, gibt es auch für diese Aufgabe Bibliotheken, die uns den code auf ein Minimum reduzieren lassen: Die *Layout–Manager*.

Abbildung 3: Unser Mockup-Element. So soll die Werkzeugleiste am Ende in etwa aussehen.

Layout–Manager Normalerweise müsste man in GUIs pixelgenau festlegen, wo sich welches Element befindet und wie es sich verhält, wenn sich die Fenstergröße ändert. Die Layout–Manager unterstützen beim Entwickeln von GUIs, indem sie alle nötigen Berechnung für die Entwickler übernehmen und dabei bestimmte Strategien verfolgen. Jeder Layout–Manager verfolgt dabei eine eigene Strategie:

- **FlowLayout**: Alle Elemente werden nebeneinander abgelegt. Ist die Zeile voll, wird in der nächsten Zeile weiter gemacht.
- **GridLayout**: Die Elemente werden in einem Gitter mit einer zuvor festgelegten Anzahl von Zeilen und Spalten abgelegt.
- **BorderLayout**: Das BorderLayout ist das klassische Fensterlayout. Es gibt fünf Bereiche für Elemente: oben, unten, links, rechts und in der Mitte.
- **cardLayout**: Arbeitet wie ein Kartenstapel und kann Elemente ein- und ausblenden.
- **GridBagLayout**: Ähnlich dem GridLayout, kann aber noch spezifischer angepasst werden.

An dieser Stelle ist noch festzuhalten, dass sich alle Layout-Manager beliebig schachteln lassen und so beliebig komplexe Layouts erstellt werden können.

Für die Post-It–Anwendung entscheiden wir uns, da wir einen Bereich oben und einen Hauptbereich in der Mitte haben, für das BorderLayout.

Dafür erweitern wir unsere Klasse `NotesFrame` um ein Feld `mainPanel` und einen Konstruktor, der diesem den Borderlayout–LayoutManager zuweist. Das ganze funktioniert wie folgt:

Abbildung 4: Das BorderLayout und seine Positionsangaben

```
public class NotesFrame extends JFrame {

        JPanel mainPanel;

        public NotesFrame(){

                this.mainPanel = new JPanel();
                this.mainPanel.setLayout(new BorderLayout());
                this.getcontentPane().add(mainPanel);
                pack();

                this.setDefaultcloseOperation(EXIT_ON_CLOSE);
        }
```

Im Konstruktor der Klasse erstellen wir zunächst ein neues Objekt vom Typ `JPanel`. Mit der Methode `setLayout(new BorderLayout())` wird dem soeben erstellen JPanel ein BorderLayout–Manager zugewiesen. Zuletzt wird das `mainPanel` mit der Methode `add(mainPanel)` dem `contentPane`[4] zugewiesen.

Damit haben wir nun ein Fenster mit einem JPanel und einem LayoutManager. Damit sind die Grundlagen für die GUI geschaffen. Was nun noch fehlt, ist das Einbinden und Konfigurieren unserer Oberflächenelemente.

[4]contentPane ist ein container, dem alle Panel-Elemente hinzugefügt werden müssen, die angezeigt werden sollen.

1.3 Toolbar

JToolBar ist eine GUI-Komponente, mit welcher man Steuerelemente, wie z.B. Buttons o.Ä. anzeigen und bedienbar machen kann. Wir erstellen uns als in einem ersten Schritt ein JPanel, das selbst wiederum eine besitzt JToolbar, in der wir unsere Buttons (bold, italic, underlined) einfügen und Anpassungen vornehmen können. Passenderweise benennen wir das JPanel ToolbarPanel:

```
public class ToolbarPanel extends JPanel {

        JToolBar toolbar;

        public ToolbarPanel(){

                toolbar = new JToolBar();
                this.add(toolbar);

        }
}
```

Damit haben wir schon das wichtigste geschafft. Unsere `JToolBar` muss nun mit Leben gefüllt werden, um den Sinn Ihres Daseins zu erfüllen. Fangen wir damit an, vier Buttons hinzuzufügen:

```
 public class ToolbarPanel extends JPanel {

        JToolBar toolbar;
        private JToggleButton boldButton = new JToggleButton();
        private JToggleButton italicButton = new JToggleButton();
        private JToggleButton underlineButton = new JToggleButton();

        public ToolbarPanel(){

                toolbar = new JToolBar();
                this.setButtons();
                this.add(toolbar);

        }

        private void setButtons(){

                toolbar.add(boldButton);
                toolbar.add(italicButton);
                toolbar.add(underlineButton);

                boldButton.setIcon(new ImageIcon("src/img/font_bold_icon&32.png
                italicButton.setIcon(new ImageIcon("src/img/font_italic_icon&3
                underlineButton.setIcon(new ImageIcon("src/img/font_underline_i
        }

 }
```

Das Hinzufügen von Buttons funktioniert genauso, wie das Hinzufügen der `JToolBar` zum `JPanel`: Mit der Methode `add(...)` wird es zu einer „Kindklassen“ der `JToolBar`.

Im Beispiel nutzen wir übrigens drei `JToggleButtons`. Im Gegensatz zu `JButtons` blieben diese gedrückt, nachdem man sie einmal angeklickt hat, wohingegen eine `JButtons` einen einmaligen *klick* auslösen und anschließend wieder den alten Zustand annehmen. Für unsere Anwendungsfälle ist der `JToggleButton` daher für die Textformatierung besser geeignet. Wir erweitern die Methode `setButtons` außerdem noch um weitere Methoden, welche Grafiken auf die Buttons setzen.

Als weitere Funktion wollen wir es den Anwendern ermöglichen die Schriftart zu wechseln. Dafür müssen wir noch das DropDown-Menü aus dem Mock-up Implementieren. Das entsprechende Element in Swing heißt: `JcomboBox`.

```
 public class ToolbarPanel extends JPanel {

//...

        private JComboBox<String> fontscomboBox;


        public ToolbarPanel(){

                toolbar = new JToolBar();
                this.setFontBox();
                this.setButtons();
                this.add(toolbar);

        }

        private void setFontBox(){
                fontsComboBox = new JComboBox<String>();
                toolbar.add(fontscomboBox);
        }

//...

 }
```

1.4 Post-It

In den vorherigen Schritten haben wir zunächst eine Hauptklasse erstellt. Diese erstellt ein Fenster (`JFrame`) mit der von uns abgeleiteten Klasse `NotesFrame`. Im dritten Schritt haben wir die Toolbar (`JToolBar`) erstellt und darin die Buttons (`JToggleButtons`) und ein Dropdown-Menü (`JcomboBox`) erstellt. Was nun noch fehlt, ist der Eingabebereich unserer Anwendung, sozusagen das Post-It.

Um die zuvor aufgestellten Anforderungen bezüglich des Post-Its umzusetzen, müssen wir nun ein Swing-Objekt finden, dass die Eingabe von Text ermöglicht. Zur Auswahl stehen uns hier verschiedene Eingabefelder: `JTextField`,

`JTextArea` oder das `JTextPane`. Nach genauerer Betrachtung wird man aber feststellen, dass sich eigentlich für unseren Zweck nur eines eignet. `JTextfield` ist lediglich ein einzeiliges Textfeld und fliegt damit raus. `JTextArea` ist mehrzeilig, lässt jedoch nur die Formatierung des gesamten Textes auf einmal zu. `JTextPane` erfüllt schließlich die Anforderung, ist aber etwas komplizierter anzupassen.

Als letzten Schritt für die Entwicklung unserer GUI, müssen wir alles noch in unserer Fensterklasse NotesFrame zusammenfügen. Dies geschieht im Konstruktor der Klasse `NotesFrame`.

```
public class NotesFrame extends JFrame {

        JPanel mainPanel;
        ToolbarPanel toolbar;
        JTextPane textpane;

        public NotesFrame(){

                this.mainPanel = new JPanel();
                toolbar = new ToolbarPanel();
                textpane = new JtextPane();

                mainPanel.add(toolbar ,BorderLayout.PAGE_START);
                mainPanel.add(textpane,BorderLayout.CENTER);
                this.getcontentPane().add(mainPanel);

                pack();
                this.setDefaultcloseOperation(EXIT_ON_CLOSE);
        }
```

Versuchen wir das ganze Programm mal zu bauen und schauen wir uns das vorläufige Ergebnis doch mal genauer an:

Das sieht unserem Mock–up schon sehr ähnlich. An dieser Stelle kann man jetzt noch versuchen, die Anordnung schöner zu gestalten oder vielleicht ein paar Anpassungen vorzunehmen, aber für unsere Zwecke in diesem Tutorial sollte das ausreichen.

Wenn man probiert auf die Buttons zu klicken passiert allerdings nichts. Das liegt natürlich daran, dass wir bisher noch gar keine Anwendungslogik implementiert haben. Darum geht es im nächsten Kapitel

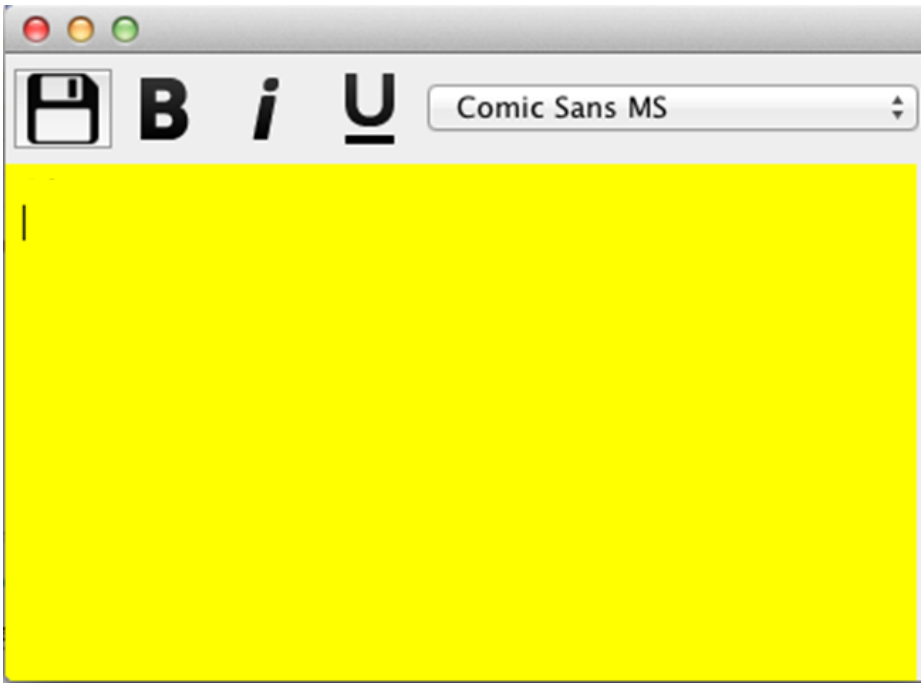

Abbildung 5: Das BorderLayout und seine Positionsangaben

2 Anwendungs–Logik

Der „Rahmen“ unserer Anwendung steht bereits und sieht auch schon ganz gut aus. Was aber noch fehlt, sind die Methoden, mit welchen wir unsere Toolbar funktionsfähig machen können.

2.1 System–Schriftarten laden

Der erste Schritt soll sein, in unserem Dropdown–Menu alle dem System bekannten Schriftarten anzuzeigen und für den Anwender auswählbar zu machen. Dafür wechseln wir zurück in die Klasse `ToolbarPanel`. In der Methode `setFontBox()` nutzen wir schon einen Konstruktor für unsere `JComboBox`, allerdings ohne Parameter. Übergibt man allerdings ein Array von Objekten, dann werden diese in der Liste angezeigt. Wir erstellen uns daher ein String–Array mit den Namen `fonts`.

```
 public class ToolbarPanel extends JPanel {

//...

        private void setFontBox(){
                fontsComboBox = new JComboBox<String>(fonts);
                toolbar.add(fontscomboBox);
        }
```

```
//...

 }
```

Da unser `fonts`–Array noch leer ist, müssen wir eine Methode finden, mit welcher wir die System-Schriftarten laden können. Die Lösung des Problems bietet uns die Klasse `java.awt.GraphicsEnvironment`. Diese Klasse stellt unter Anderem Methoden für den Zugriff auf die Schriftarten des Systems bereit. Dafür müssen wir uns zuerst mit der Factory–Methode `getLocalGraphicsEnvironment()` ein GraphicsEnviroment–Objekt erstellen und können anschließend mit der Methode `getAvailableFontFamilyNames()` auf alle dem System bekannten Schriftarten zugreifen. Wir nutzen hier speziell diese Methode, um Duplikate auszuschließen.

```
public class ToolbarPanel extends JPanel {

//...

        private void setFontBox(){
                GraphicsEnvironment enviroment = GraphicsEnvironment.getLocalGr
                fonts = enviroment.getAvailableFontFamilyNames();
                fontsComboBox = new JComboBox<String>(fonts);
                toolbar.add(fontscomboBox);
        }

//...

 }
```

Starten wir nun die Anwendung erneut, sehen wir eine Liste aller verfügbaren Schriftarten in unserem DropDown-Menü. Sehr gut! Was jetzt noch fehlt, sind Aktionen, die beim Klicken auf einen Button oder beim Auswählen einer Schriftart ausgeführt werden. Die Lösung dafür sind „Listener“.

2.2 Listener

Da wir mit einer interaktiven graphischen Oberfläche arbeiten, die in Abhängigkeit von einer Nutzerinteraktion (z.B. Klick auf den Button „Bold“) ihren Zustand verändert, müssen wir dafür sorgen, dass das Programm auf die Eingaben des Benutzer *reagiert*. Um dies zu ermöglichen, benötigen wir sog. Listener–Klassen, die bei einer Interaktion eine bestimmte Methode aufrufen, in der wir festlegen, was passieren soll.

Jedes unserer interaktiven Toolbar Elemente (die Buttons und das DropDown–Menü) bekommen eine eigene Listener–Klasse, die jeweils eine Aktion ausführt, sobald ein Element angeklickt wurde.

Zur Implementierung der Listener gibt es verschiedene Möglichkeiten: Man könnte a) für jeden Button eine eigene Klasse anlegen, b) eine gemeinsame Listener–Klasse, die je nachdem welcher Button sie aufruft unterschiedliche Aktionen durchführt, oder c) jeweils anonyme innere Klasse als Listener erstellen.

In diesem Tutorial entscheiden wir uns für die letzte Möglichkeit, da diese uns ein paar Vorteile bietet. Welche das sind, sehen wir gleich. Schauen wir uns

zuerst mal die Implementierung an:

```
public class ToolbarPanel extends JPanel {

//...

        private void setButtons(){

                toolbar.add(boldButton);
                toolbar.add(italicButton);
                toolbar.add(underlineButton);

                boldButton.setIcon(new ImageIcon("src/img/font_bold_icon&32.png
                italicButton.setIcon(new ImageIcon("src/img/font_italic_icon&32
                underlineButton.setIcon(new ImageIcon("src/img/font_underline_i

                boldButton.addItemListener(new ItemListener() {...});
                italicButton.addItemListener(new ItemListener() {...});
                underlineButton.addItemListener(new ItemListener() {...});
        }

}
```

Mithilfe der Methode `addItemListener(...)` wird einem Button ein neuer ItemListener hinzugefügt. Für die anonyme innere Klasse ist nichts weiter notwendig, als diese innerhalb der Methode zu definieren. Bisher tut unsere Klasse noch nichts, außer zu existieren. Indem wir aber eine odere mehrere ihrer Methoden überschreiben, können wir (Re-)Aktionen der Buttons definieren.

```
public class ToolbarPanel extends JPanel {

//...

        private void setButtons(){

                //...

                boldButton.addItemListener(new ItemListener() {

                        @Override
                        public void itemStateChanged(ItemEvent e) {
                                //..
                        }

                });
                //...
        }

}
```

Wie im oben stehenden Code–Listing zu sehen, wird die Methode `itemStateChanged(ItemEvent e)`

überschrieben. Die Methode wird immer dann aufgerufen, wenn ein Button geklickt und wieder losgelassen wurde. Über `ItemEvent e` können wir nun feststellen, wie der aktuelle Status des „zugehörigen“ Buttons ist.

```
public class ToolbarPanel extends JPanel {

        boolean bold;

//...

        private void setButtons(){

                //...

                boldButton.addItemListener(new ItemListener() {

                        @Override
                        public void itemStateChanged(ItemEvent e) {
                                if(e.getStateChange()==e.SELECTED){
                                bold = true;
                                } else {
                                bold = false;
                                }
                        }

                });
                //...
        }

}
```

Möglich wird dies mit der Instanzmethode `getStateChange()`, welche die Konstante `SELECTED` zurückgibt, falls ein Button angeklickt ist. Dies kann für die anderen Buttons analog umgesetzt werden. In unserem Post-It speichern wir den aktuellen Zustand der Buttons in boolschen Variablen, um später darauf zugreifen zu können.

Die Implementierung für das Dropdown–Menü funktioniert analog: Zunächst wird ein Listener hinzugefügt und anschließend eine (bzw. mehrere) Methoden überschrieben. Der Unterschied besteht darin, dass diesmal ein Listener von Typ `PopupMenuListener` verwendet werden muss, da ein Listener für JComboBox anders auf Eingaben reagieren muss. Dafür müssen die Abstrakten Methoden `popupMenuWillBecomeVisible(PopupMenuEvent e)`, `popupMenuWillBecomeInvisible(PopupMenuEvent e)` und `popupMenuCanceled(PopupMenuEvent e)` zwingend überschrieben werden. Für unseren Anwendungsfall interessiert aber nur `popupMenuWillBecomeInvisible(PopupMenuEvent e)`. Sobald das DropDown-Menü geschlossen wurde, wollen wir nämlich die zuletzt ausgewählte Schriftart im String `font` speichern.

```
public class ToolbarPanel extends JPanel {

        String[] fonts;
        String font;
```

```
//...

private void setFontBox() {
                GraphicsEnvironment enviroment = GraphicsEnvironment.getLocalGr
                fonts = enviroment.getAvailableFontFamilyNames();
                fontsComboBox = new JComboBox<String>(fonts);
                fontsComboBox.addPopupMenuListener(new PopupMenuListener() {

                        @Override
                        public void popupMenuWillBecomeVisible(PopupMenuEvent e
                        }

                        @Override
                        public void popupMenuWillBecomeInvisible(PopupMenuEvent
                                font =fontsComboBox.getSelectedItem().toString(
                        }

                        @Override
                        public void popupMenuCanceled(PopupMenuEvent e) {
                        }
                });

                fontsComboBox.setSelectedItem(note.getFontFamily());
        }


 }
```

2.3 Verknüpfung von Toolbar und Textpane

Im letzten Schritt müssen wir dafür sorgen, dass unser Textpane die aktuellen Einstellungen, die wir auf der Toolbar getätigt haben und die unsere Listener in den lokalen Variablen gespeichert haben, vom Textpane verwendet werden.

Dafür erstellen wir uns zunächst Getter-Methoden für unsere drei Button Boolean-Felder und das Schriftart String-Feld. In den meisten IDE's geht das mit ein paar Menü-klicks (einfach mal Googlen. In Eclipse geht das recht einfach mit einem Klick auf *Source -¿ Generate Getters and Setters*). Das Ergebnis sollten folgende vier Getter-Methoden sein:

```
 public class ToolbarPanel extends JPanel {

                String font;
                boolean bold;
                boolean italic;
                boolean underlined;

//...

                public String getFont() {
```

```
                return font;
        }

        public boolean isBold() {
                return bold;
        }

        public boolean isItalic() {
                return italic;
        }

        public boolean isUnderlined() {
                return underlined;
        }

}
```

Damit haben wir die Schnittstellen auf der Seite der Klasse `ToolbarPanel` gebaut. Nun wechseln wir in die Klasse `NotesTextpane`.

Da die Klasse noch leer ist, müssen wir uns nun über zwei Dinge Gedanken machen: Einerseits müssen wir eine Möglichkeit finden, wie wir die TextSettings in der Klasse setzen und andererseits ist es notwendig an einem geeigneten Zeitpunkt die Settings zu laden.

2.4 TextSetting aktualisieren

Mit einem Blick in die API von `JTextPane` (die NotesTextPane erweitert), findet man die Methode setCharacterAttributes(...), die wie folgt beschrieben ist:

public void setCharacterAttributes(AttributeSet attr, boolean replace) Applies the given attributes to character content. If there is a selection, the attributes are applied to the selection range. If there is no selection, the attributes are applied to the input attribute set which defines the attributes for any new text that gets inserted.

Parameters:

- attr - the attributes
- replace - if true, then replace the existing attributes first

Mit dieser Methode kann man die Attribute der Zeichen verändern, die man eintippt. Zusätzlich werden markierte Bereiche überschrieben. Sehr gut, denn das ist genau das was wir wollen. Was wir jetzt noch brauchen ist ein AttributeSet, das wir der Methode übergeben.