

Kapitel 1

Grundlagen SWT

Das Standard-Widget-Toolkit und das darauf aufbauende JFace-Toolkit entstanden im Rahmen der Entwicklung von Eclipse™ ursprünglich bei IBM. Heute liegt Eclipse (mit allen dazu gehörenden Toolkits) in den Händen der Eclipse Foundation und ist in der Version **2018-09** bei www.eclipse.org verfügbar. Bei uns installiert sind – je nach Labor – die Versionen **Photon** bzw. **Oxygen**. Einige Grundeigenschaften von Eclipse:

- In Java entwickelt.
- Enthält jeweils plattform*abhängigen* Kernel.
- Für viele Plattformen verfügbar (u. a. Linux, Windows 7 bis 10, Mac OS, ...).
- Ist eine IDE.
- Ist ein Software-Framework.
- Hat OSGi als Basis-Architektur (eigene OSGi-Implementierung Equinox)
- Stellt eigene Widget-Sets (SWT und JFace) bereit.

Die Architektur von SWT im Vergleich mit dem Java-eigenen Widget-Set Swing lässt sich folgendermaßen darstellen:



Abbildung 1.1: Vergleich Java Swing mit SWT / JFace (aus [8], S.166)

SWT setzt hierbei im Gegensatz zu Java Swing direkt auf dem nativen Windowing-System auf. Der GUI-Entwickler sieht hiervon jedoch nichts mehr.

1.1 Installation des SWT

Für die Installation zu Hause gilt: Sie müssen nur das „richtige“ Eclipse-IDE-Paket von www.eclipse.org herunterladen. und entpacken.

- Gehen Sie auf die Seite www.eclipse.org und gehen Sie dort auf die URL <http://www.eclipse.org/downloads/>.
- Sie sehen auf dieser Seite verschiedene Eclipse-Pakete (für Java-Entwicklung, für C++-Entwicklung, usw.).
- Laden Sie sich für Ihre Plattform (Windows oder Linux) die IDE **Eclipse for RCP and RAP Developers** herunter. Hierbei handelt es sich um eine recht große .zip-Datei.
- Achten Sie bitte darauf, ob Sie ein 32- oder ein 64-Bit-Betriebssystem haben. Die Eclipse-Pakete haben je nach Betriebssystem durchaus unterschiedliche Kernel-Pakete!
- Entpacken Sie Ihre .zip-Datei in Ihrem gewünschten Ziel-Verzeichnis (z. B. C:\EclipseRCP).

Unter Windows können Sie statt dessen auch den Installer verwenden – dann sind allerdings die Unterverzeichnisse, die Sie für's SWT benötigen, etwas schwerer zu finden. Nach dem Entpacken sollten Sie im Zielverzeichnis folgenden Dateibaum sehen:

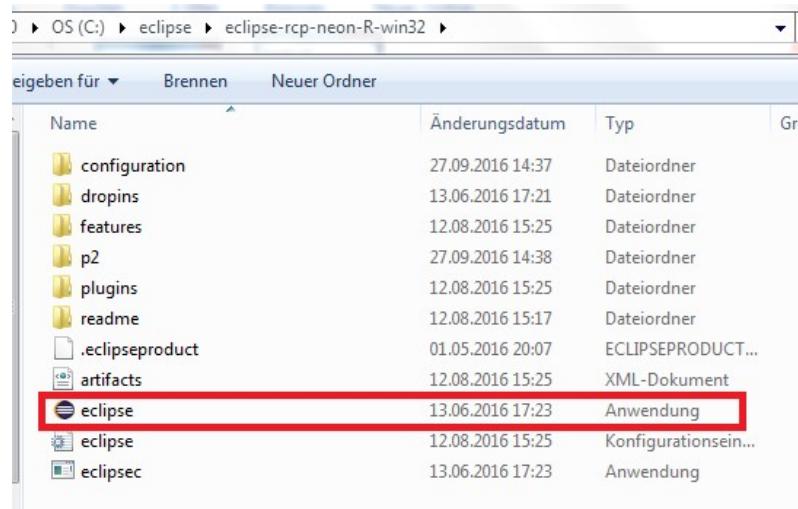


Abbildung 1.2: Dateibaum von Eclipse

Durch Doppelklick auf das Eclipse-Symbol starten Sie die IDE.

1.2 Aufbau von SWT

Bevor wir ein erstes HelloWorld-Projekt für unser SWT erstellen, zeigen wir kurz die wichtigsten Bestandteile des Widget-Sets: Das SWT selbst ist unter Windows (32-Bit-Variante) als Bibliothek in Form der 2 .jar-Dateien (Java-Archive)

- org.eclipse.swt_<...versionsnummer...>.jar
 - org.eclipse.swt.win32.win32.x86_<...versionsnummer...>.jar

in Eclipse integriert. Die Versionsnummern können sich – je nach Eclipse-Version – ändern. Das folgende Bild zeigt einen Überblick über die Klassenhierarchie des SWT:

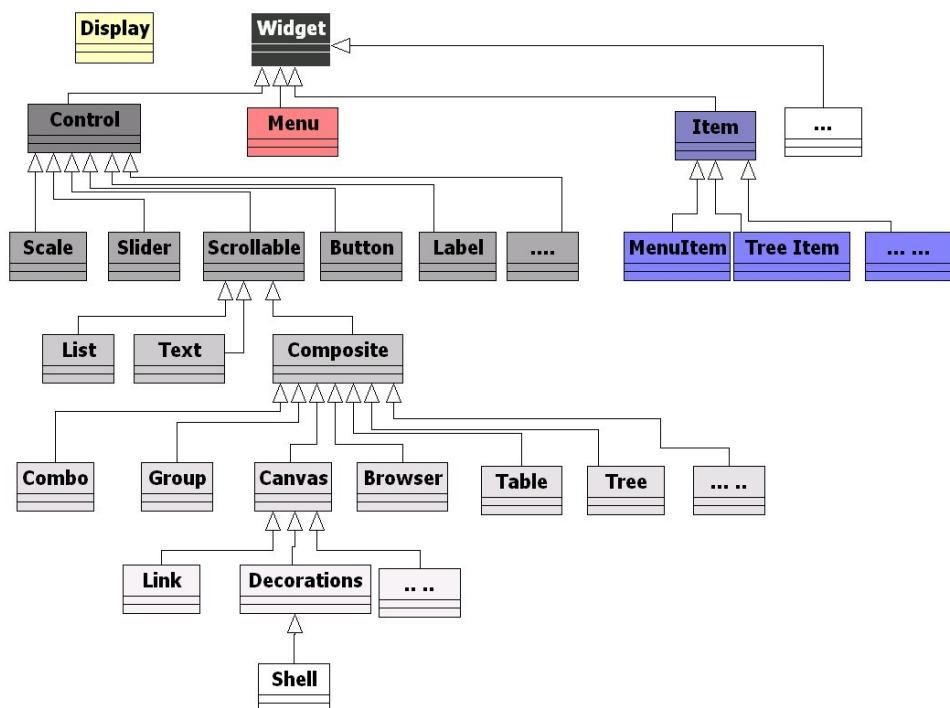


Abbildung 1.3: Klassenhierarchie von SWT

Für uns werden hier zunächst die Klassen *Shell*, *Display* und *Label* interessant sein.

1.3 Ein erstes Hello-World-Projekt

Wir erstellen unser erstes SWT-GUI-Projekt, welches ein einfaches „Hello-World“-Textlabel anzeigt:



Abbildung 1.4: Hello World in Eclipse

Schritt 1: Starten der Eclipse-IDE durch Doppelklick auf das Eclipse-Symbol im Eclipse-Ordner:

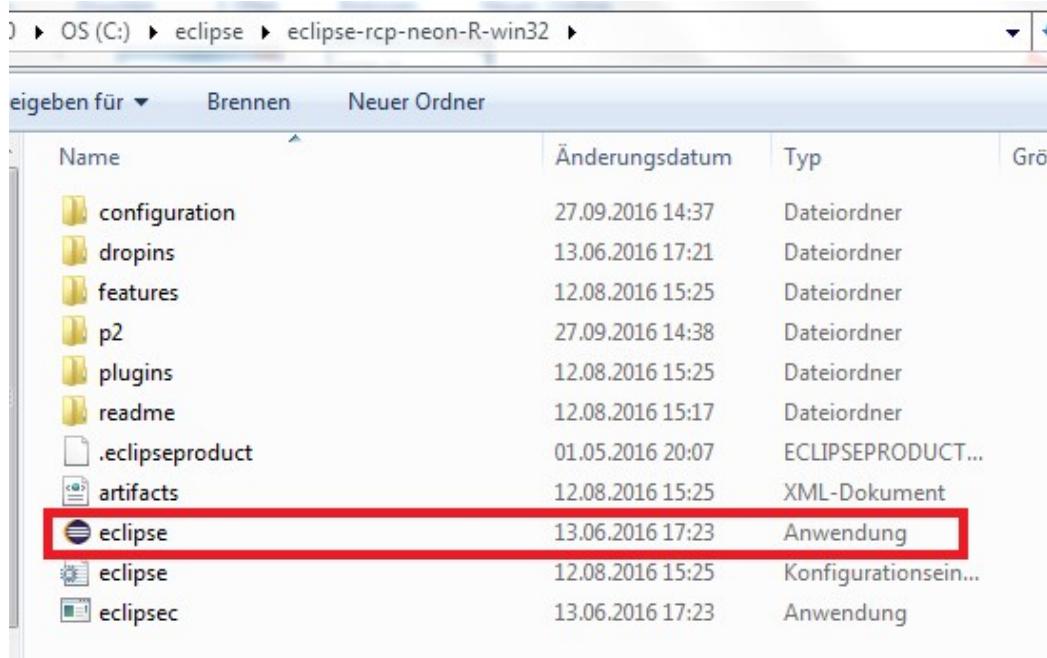


Abbildung 1.5: Starten von Eclipse

Die IDE fragt uns, wo wir unseren Workspace anlegen wollen:

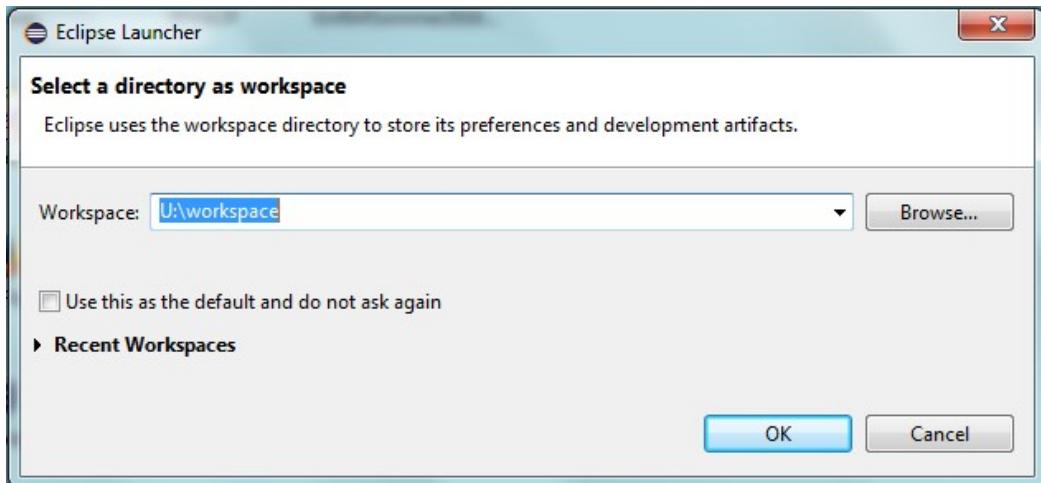


Abbildung 1.6: Starten von Eclipse – Workspace auswählen

Hier geben wir ein Verzeichnis an, in dem unsere künftigen SWT-Projekte abgelegt werden sollen. Jedes SWT-basierte Programm, welches wir erstellen, wird in Form eines *Eclipse-Projektes* organisiert.

KAPITEL 1. GRUNDLAGEN SWT

Da wir derzeit noch keine Projekte erstellt haben, erscheint die Eclipse-IDE mit ihrem Welcome-Fenster:

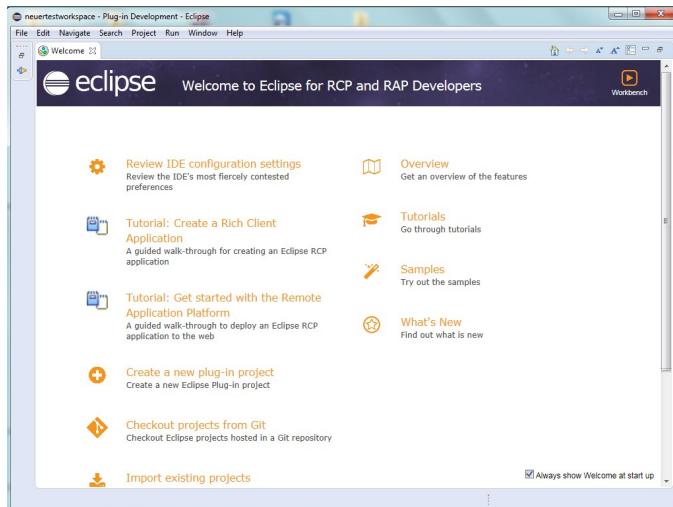


Abbildung 1.7: Initialzustand Eclipse-IDE

Wir klicken den Welcome-Reiter weg – und beginnen mit unserem Hello-World-Projekt.

Schritt 2: In der IDE die Java-Perspektive einstellen

Da die Eclipse-IDE für viele Arten von Software-Projekten eingesetzt werden kann, gibt es verschiedene Grundeinstellungen für das Eclipse-Hauptfenster. Diese Grundeinstellungen werden als **Perspektive** bezeichnet.

Wenn wir RCP-Projekte erstellen wollen, benötigen wir die RCP-Perspektive. Wenn wir C++-Projekte erstellen wollen, benötigen wir die C++-Perspektive. Diese Hauptfenster-Einstellungen liefern uns Zugriff zu den jeweils benötigten Compilern und Code-Generatoren. Für SWT-Projekte benötigen wir zunächst nur die einfache Java-Perspektive, zu der wir allerdings später noch unsere SWT-Bibliotheken hinzubinden müssen.

Wir wechseln auf die Java-Perspektive mit
Window → Open Perspective → Other
und wählen hier **Java**.

Schritt 3: Unter Eclipse ein neues Projekt **Beispiel1** anlegen

Wird mit **File → New Project → Java Project** initiiert. Wir geben dem Projekt den Namen **Beispiel1**. Das Projekt sollte nun folgende Ordnerstruktur haben:

The screenshot shows the Eclipse Java Project Explorer view. The left sidebar shows the project navigation path: 'ace' > 'Beispiel1'. The main area displays the contents of the 'Beispiel1' project. A table lists the files and folders in the project root:

Name	Änderungsdatum	Typ	Größe
.settings	03.10.2016 15:45	Dateiordner	
bin	03.10.2016 15:45	Dateiordner	
src	03.10.2016 15:45	Dateiordner	
.classpath	03.10.2016 15:45	CLASSPATH-Datei	1 KB
.project	03.10.2016 15:45	PROJECT-Datei	1 KB

Abbildung 1.8: Ordnerstruktur SWT-Projekt *Beispiel1*

Das Projekt enthält zwei Unterordner:

- *src* wird künftig unsere Java-Quelldateien (engl. Source) enthalten.
- *bin* wird die compilierten *.class*-Dateien enthalten.
- *.settings* enthält Zustandsdaten Ihres Projektes, die von Eclipse automatisch abgespeichert werden.

Weiterhin finden Sie im Projektordner *Beispiel1* die automatisch erzeugten Dateien

- *.settings* und
- *.project*
- *.classpath*

Diese werden wir nach Fertigstellung des Projektes noch genauer betrachten.

Schritt 4: SWT-Bibliotheken einbinden

KAPITEL 1. GRUNDLAGEN SWT

Bevor wir für unsere erste „Hello-World“-Oberfläche die benötigten SWT-Klassen verwenden können, müssen wir die beiden in Kapitel 1.2 genannten SWT-Bibliotheken in unser *Beispiel1*-Projekt einbinden. Ansonsten würde der Compiler die SWT-Klassen nicht erkennen. Dieses Einbinden erfordert einige Teilschritte:

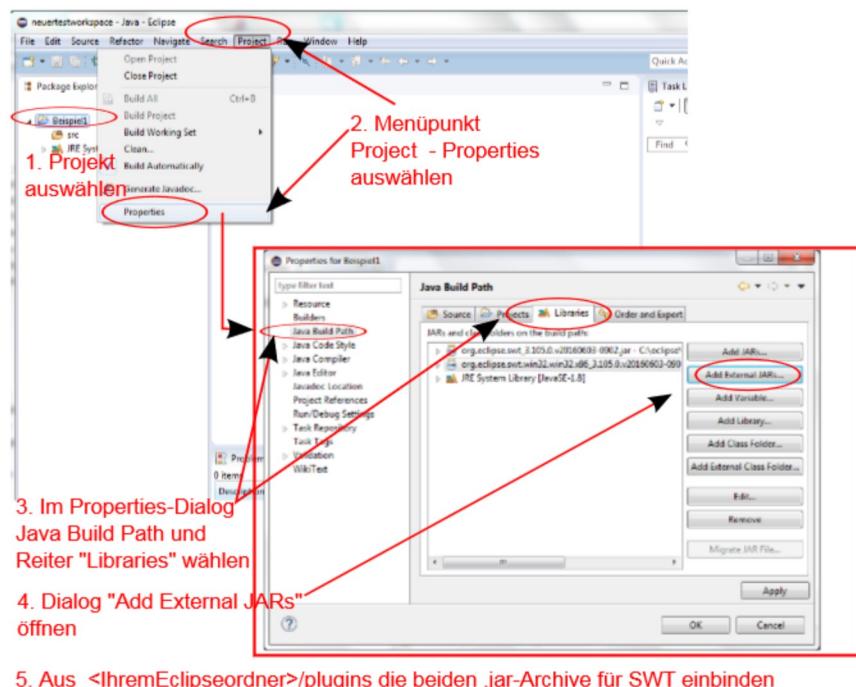


Abbildung 1.9: SWT-Bibliotheken einbinden Teil 1

- Wir gehen auf **Project → Properties** und dort im Dialogfenster auf **Java Build Path**.
- Wir wechseln hier in den Tab **Libraries**.
- Mit dem Button **Add External Jars** bekommen wir einen Datei-Auswahldialog.
- Die auszuwählenden SWT-Bibliotheken liegen im Verzeichnis der **Eclipse-Installation** im Ordner **plugins** als JAR-Archive vor.
- Wir wählen dort die beiden SWT-Jar-Archive aus – sie sind nach der Bestätigung bereits im Dialogfenster zu sehen.
- Nun bestätigen wir ein letztes Mal mit **Ok**.

6. Im Hauptfenster sind die Bibliotheken nun in den "Referenced Libraries" sichtbar:

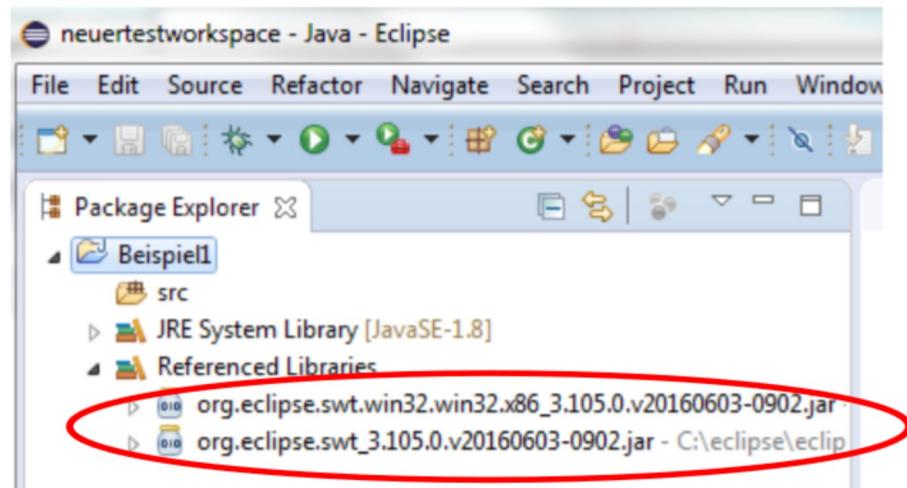


Abbildung 1.10: SWT-Bibliotheken einbinden Teil 2

Schritt 5: Programm eingeben

Wir wählen links im Project-Explorer unser Projekt **Beispiel1** an. In diesem Projekt erzeugen wir mit **File → New → Class** eine neue Java-Klasse. Wir nennen die Klasse **Beispiel1** und geben ihr eine *main()*-Methode. Wir diskutieren den Quelltext:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.widgets.Display;
4 import org.eclipse.swt.widgets.Label;
5 import org.eclipse.swt.widgets.Shell;
6
7
8 public class Beispiel1 {
9
10    public static void main(String[] args) {
11
12        // 1. Display-Objekt stellt Verbindung
13        // zwischen nativem Windowing-System und
14        // aeusserem Widget her.
15        Display display = new Display();
16
17        // 2. Shell ist aeusseres Widget.
18        // Es wird als Kind-Widget des Display-
19        // Objektes betrachtet
20        Shell shell = new Shell(display);
```

Wir legen zunächst ein *Display()*-Objekt an. Dieses übernimmt für uns die Kommunikation mit dem nativen Windowing-System. Das *Shell*-Objekt ist das äußerste, umrahmende Widget mit Titelleiste. In sie werden später alle Kind-Widgets platziert.

```

21 // 3. Die obere, linke Ecke der Shell wird
22 // auf Pixel 100 (x-Offset), 200 (y-Offset)
23 // auf dem Desktop platziert.
24 // Die Shell ist 400 Pixel breit und
25 // 200 Pixel hoch
26 shell.setBounds(100,200,400,200);

```

Hier rufen wir die `setBounds`-Methode der Shell auf. Jedes SWT-Widget erbt diese Methode von der Klasse `Widget`. In unserem Falle werden die geometrischen Eigenschaften der Shell festgelegt. Sie ist 400 x 200 Pixel groß. Ihre linke, obere Ecke wird, sofern das Windowing-System dies zulässt, auf x-Koordinate 100 und y-Koordinate 200 (in Pixeln gerechnet) in den Desktop platziert:

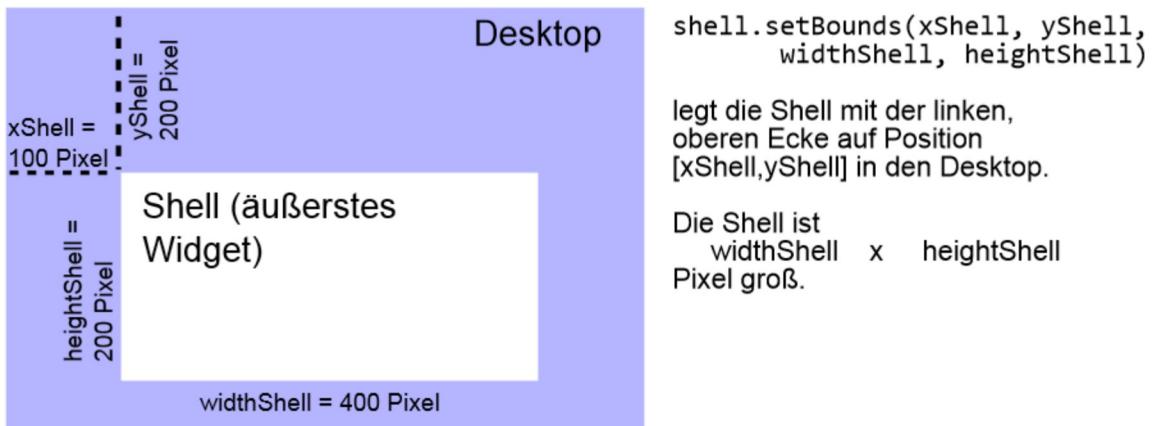


Abbildung 1.11: Wirkungsweise der Methode `setBounds(...)`

KAPITEL 1. GRUNDLAGEN SWT

```
27     // 3. Ein einfaches Textlabel wird
28     // erzeugt und in die Shell gelegt. Die
29     // Schrift auf dem Label ist linksbuendig
30     Label label = new Label(shell, SWT.LEFT);
31
32     // 4. Das Label bekommt seinen
33     // darzustellenden Text
34     label.setText(
35         "Hallo\u2022Mitglieder\u2022der\u2022LVA\u2022BO");
```

Als nächstes erzeugen wir ein Textlabel. Der Konstruktorauftrag bekommt 2 Parameter:

1. Das Eltern-Widget, in dem das Label platziert werden soll.
2. Eine **Style-Konstante**. Hierbei handelt es sich um einen int-Wert, in dem ein oder mehrere Flag-Bits auf 1 gesetzt sind. Die Style-Konstante unseres Labels sagt hier: „Schrift bitte linksbündig!“.

Der Methodenauftrag *setText(...)* gibt dem Label seine Beschriftung mit.

```
36     // 5. Das Label wird so gross, dass es
37     // genau in die Shell passt. Seine linke,
38     // obere Ecke wird auf Koordinate 0/0
39     // (links oben) in die Shell platziert.
40     label.setBounds(shell.getClientArea());
```

Hier werden die Geometrie-Einstellungen des Labels genauer spezifiziert. Dazu verwenden wir eine zweite, ebenfalls von *Widget* ererbte Variante der *setBounds(...)*-Methode. Sie bekommt ein „Rechteck“-Objekt übergeben, welches über

- x-Koordinate der linken oberen Ecke im Parent,
- y-Koordinate der linken oberen Ecke im Parent,
- Breite des Widgets (in Pixeln),
- Höhe des Widgets (in Pixeln)

definiert ist. In unserem Falle ist das „Rechteck“ so definiert, dass das Label genau in den belegbaren Bereich der Shell passt.

```
41     // 6. Ein Font-Objekt für die gewünschte
42     // Schriftart wird erzeugt.
43     Font f= new Font(
44         display, "Arial", 18, SWT.BOLD);
45
46     // 7. Das Font-Objekt wird mit dem Label
47     // verknüpft: Das Label stellt seine
48     // Beschriftung nun in diesem Font dar.
49     label.setFont(f);
```

Hier wird ein *Font*-Objekt erzeugt. Mit diesem definieren wir, welche Schriftart bei der Beschriftung des Labels verwendet werden soll. Auch hier verwenden wir eine Style-Konstante *SWT.BOLD*, welche besagt, dass wir fette Schrift haben wollen.

```
50
51     // 8. Die Shell wird auf "sichtbar"
52     // geschaltet
53     shell.open();
54
55     // 9. Hier wird die (blockierende)
56     // Event-Loop gestartet.
57     while(!shell.isDisposed()) {
58         if(!display.readAndDispatch()){
59             display.sleep();
60         } // end if
61     } // end while
62
63 } // end method main()
64 } // end class
```

Im letzten Abschnitt wird das umrahmende Widget mit all seinen Kind-Widgets auf „sichtbar“ geschaltet. Die Schleifenkonstruktion darunter zeigt unsere blockierende Event-Loop. Sie ist folgendermaßen zu verstehen: So lange die Shell nicht zerstört wurde, wird beim *Display*-Objekt nachgefragt, ob ein weiteres Event ansteht. Falls nicht, wird das *Display*-Objekt schlafen gelegt. Es blockiert die Event-Loop. Sobald ein Event ansteht, wird das *Display*-Objekt wieder „aufgeweckt“ (in den Zustand „Runnable“ – lauffähig zurückversetzt).

KAPITEL 1. GRUNDLAGEN SWT

Es folgen die beiden Projektdateien *.classpath* und *.project*.
Die Datei *.project* enthält die Projektdefinitionen – beispielsweise den Projektnamen.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <projectDescription>
3   <name>Beispiel1</name>
4   <comment> </comment>
5   <projects> </projects>
6   <buildSpec>
7     <buildCommand>
8       <name>org.eclipse.jdt.core.javabuilder</name>
9       <arguments> </arguments>
10      </buildCommand>
11    </buildSpec>
12    <natures>
13      <nature>org.eclipse.jdt.core.javanature</nature>
14    </natures>
15  </projectDescription>
```

Abbildung 1.12: Datei *.project*

Die Datei *.classpath* enthält sämtliche Bibliothekspfade, sowie die Pfadstruktur des Projektes:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <classpath>
3   <classpathentry kind="src" path="src"/>
4   <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.
internal.debug.ui.launcher.StandardVMType/JavaSE-1.7"/>
5   <classpathentry kind="lib" path="c:/Eclipse4.2JavaRCP/eclipse/plugins/org.eclipse.swt_3.
100.1.v4234e.jar"/>
6   <classpathentry kind="lib" path="C:/Eclipse4.2JavaRCP/eclipse/plugins/org.eclipse.swt.win32.
win32-x86_3.100.1.v4234e.jar"/>
7   <classpathentry kind="output" path="bin"/>
8 </classpath>
```

Abbildung 1.13: Datei *.classpath*

1.4 Wichtigste strukturelle Eigenschaften von SWT-GUIs

Merke

Folgende strukturelle Eigenschaften von SWT-Oberflächen sollten Sie sich merken:

- Eine SWT-Oberfläche benötigt ein *Display*-Objekt. Dieses übernimmt in der EventLoop die Kommunikation mit dem nativen Windowing-System.
- Ein *Shell*- oder *Composite*-Objekt wird häufig als äußerstes, umrahmendes Widget verwendet. Es dient als „Leinwand“, in welche sämtliche darzustellende Oberflächen-Elemente gezeichnet werden. Es ist das **Eltern-Widget** der in ihm zu zeichnenden Oberflächen-Elemente.
- Jedes SWT-Widget bekommt bei seiner Erzeugung die Referenz auf sein **Eltern-Widget** über den Konstruktor mit.
- Das **äußerste** umrahmende Widget bekommt als Eltern-Widget immer das *Display*-Objekt bei der Erzeugung übergeben.
- Bei vielen SWT-Widgets können zusätzliche Eigenschaften über **Style**-Konstanten festgelegt werden. Hierbei ist zu beachten:
 - Alle in SWT gültigen Style-Konstanten sind als *public static final*-Elemente in der Klasse *SWT* untergebracht.
 - Zu jedem SWT-Widget ist in der API vermerkt, mit welchen *SWT.xxx*-Konstanten es konfigurierbar ist.
- Die blockierende Eventloop muss in reinen SWT-Anwendungen selbst implementiert werden. In späteren Beispielen wird sie in einer eigenen Methode gekapselt.
- Bei Eclipse-RCP-GUIs und Android-Apps ist die Eventloop nicht mehr direkt sichtbar.

Kapitel 2

Listener und Events

In diesem Kapitel lernen wir, einer Benutzungsoberfläche dynamisches Verhalten hinzuzufügen. So soll eine Oberfläche beispielsweise reagieren, wenn ein Button oder eine Menüoption betätigt wurde.

2.1 Delegationsmodell für Events und Listener

Wie viele andere Widget-Sets verwendet auch das SWT hierzu das **Delegationsmodell** oder umgangssprachlich: Listener und Events.

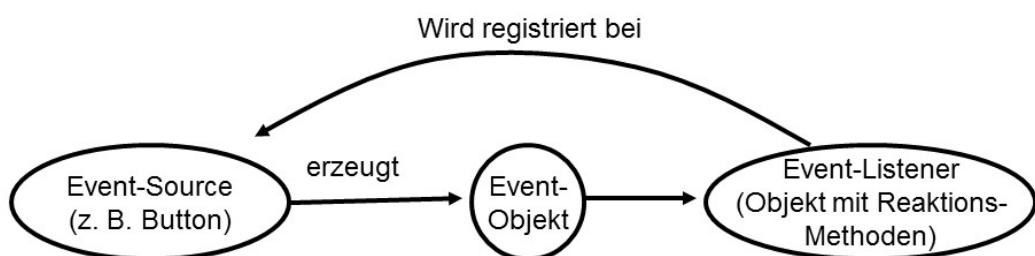


Abbildung 2.1: Delegationsmodell (frei nach [14], S. 648)

Ein Listener-Objekt registriert sich bei der Event-Source (z.B. Button), damit es zukünftig bei den interessierenden Ereignissen automatisch benachrichtigt wird. Bei einer Event-Source können sich auch mehrere Listener registrieren lassen. Nachdem ein Event (z. B. Button anselektieren) aufgetreten ist, wird es von der Event-Source an alle registrierten Listener-Objekte weiter gegeben.

KAPITEL 2. LISTENER UND EVENTS

Die grundlegenden Klassen und Interfaces für die Ereignisverarbeitung befinden sich im Package `org.eclipse.swt.events`. Die verschiedenen Arten von **Events** sind von der Klasse `TypedEvent` abgeleitet, die sich wiederum in folgender Klassenhierarchie befindet:

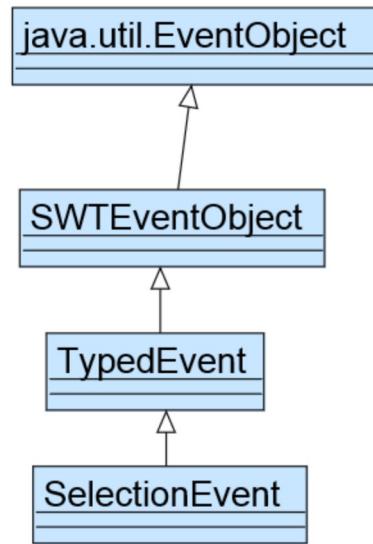


Abbildung 2.2: Klassenhierarchie SWT-Events am Beispiel `SelectionEvent`

Das Klassendiagramm zeigt als unterste Kindklasse das **SelectionEvent**. Von dieser Klasse werden Objekte generiert, wenn z. B. ein Button *selektiert* wird. Für die verschiedenen Reaktionsmöglichkeiten einer SWT-Oberfläche sind **Listener-Interfaces** vorgesehen. Diese Interfaces implementieren wir mit unseren Listener-Klassen. Wir zeigen die Klassen- und Interface-Beziehungen am Beispiel **SelectionListener**:

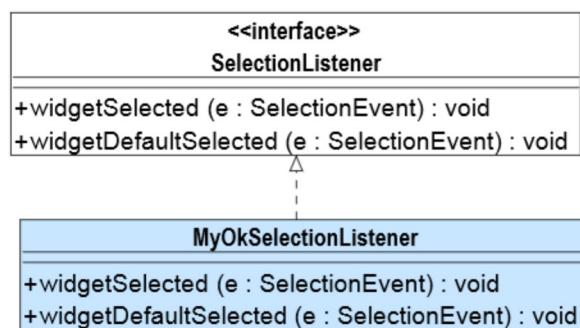


Abbildung 2.3: Klassenhierarchie SWT-Listener am Beispiel `SelectionListener`

Bei den beiden im Interface vorgegebenen Methoden handelt es sich um Reaktionsmethoden, die dann z. B. das Aufklappen eines Dialogfensters nach Drücken eines Buttons

bewerkstelligen sollen.

Das Reaktionsverhalten wird vom Entwickler in einer eigenen Klasse – hier z. B. *MyOkSelectionListener* – implementiert. Diese Klasse wiederum implementiert das Interface *SelectionListener*.

Unbequem für den Entwickler ist es, dass er in seiner Listenerklasse immer **alle** vom Interface vorgegebenen Methoden implementieren muss – selbst wenn er nur eine einzige der vorgegebenen Reaktionsmethoden benötigt.

Daher hat das SWT für die am häufigsten verwendeten Listener schon vorgefertigte *Adapterklassen* bereitgestellt. Sie implementieren leere „Dummy“-Methoden ihrer Interfaces. So muss der Entwickler nur noch eine **Kindklasse** der Adapterklasse schreiben. In ihr **überschreibt** er nur die Reaktionsmethoden, die er in seiner Oberfläche auch wirklich benötigt:

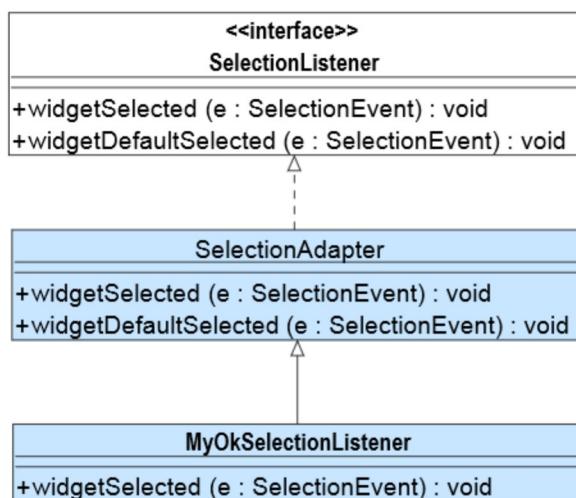


Abbildung 2.4: Klassenhierarchie SWT-Listener am Beispiel *SelectionAdapter*

2.2 Anonyme Listener

Bei sehr kleinen Reaktionsmethoden werden häufig **anonyme Listenerklassen** verwendet. Hierbei handelt es sich um eine spezielle Art von **Inner Classes**. Sie bekommen nicht explizit vom Entwickler ihren Klassennamen. Statt dessen vergibt der Java-Compiler **automatisch** einen Klassennamen für diese Listener. Die Notation dieser anonymen Klassen ist zwar etwas gewöhnungsbedürftig. Sie sind jedoch sehr bequem zu benutzen.

Wir arbeiten zunächst mit einem **SelectionListener**, der die Reaktion auf das *Selektieren* eines Buttons realisieren wird. Das Beispiel, welches wir hier verwenden, beinhaltet

KAPITEL 2. LISTENER UND EVENTS

ein zunächst noch leeres Label und zwei Buttons:

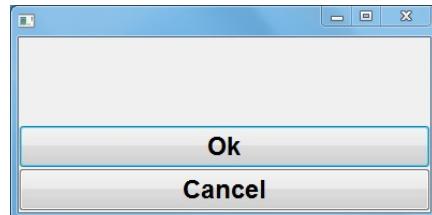


Abbildung 2.5: Beispiel2: Oberfläche mit einem Label und 2 Buttons

Das Label und die Buttons sind untereinander angeordnet. Sobald der Ok-Button gedrückt wird, erscheint der aktualisierte Wert eines Zählers im Label:

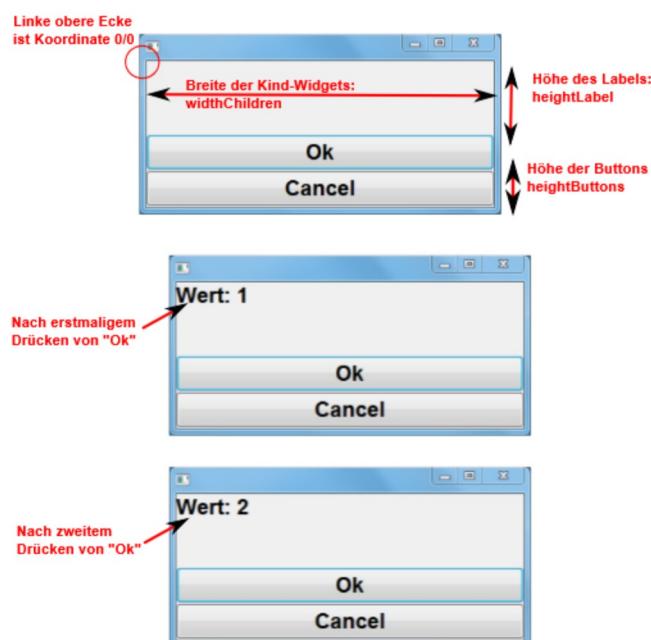


Abbildung 2.6: Verhalten von Beispiel2

Wir zeigen zunächst die Listenerimplementierung des Ok-Buttons:

```

1  Button buttonOk = new Button(shell, SWT.PUSH);
2
3  buttonOk.addSelectionListener(
4      new SelectionListener(){
5          // Eine einfache Zaehl-Variable
6          int i=0;
7
8          @Override
9          public void widgetDefaultSelected
10             (SelectionEvent arg0) {
11                 // tue nichts -- nur wegen Interface-
12                 // Vorgabe eingefuegt
13
14         } // end method
15
16         @Override
17         public void widgetSelected(
18             SelectionEvent arg0) {
19                 // Zaehler inkrementieren
20                 i++;
21                 // Setze den Zaehler in das oben
22                 // stehende Label ein
23                 label.setText("Wert:" + i);
24         } // end method
25
26     } // end class
27 ); // end call .addSelectionListener()

```

Methodenaufruf des Buttons

Definition einer anonymen "Inner Class".
Es wird gleichzeitig ein Objekt dieser Klasse erzeugt

Abbildung 2.7: Listener-Implementierung des Ok-Buttons

Wir sehen, dass hier **im Methodenaufruf** `addSelectionListener(...)` eine Objekterzeugung stattfindet und gleichzeitig eine Klassenimplementierung zu sehen ist.

KAPITEL 2. LISTENER UND EVENTS

Das *new SelectionListener()*-Statement wirkt sich folgendermaßen aus:

1. Hier geschieht eine Objekterzeugung. Da es aber ja keine **Klasse** namens *SelectionListener* gibt, sondern nur ein Interface dieses Namens, muss hier eine Klasse entstehen, die dieses Interface implementiert.
2. Die Listener-Implementierung geschieht in den geschweiften Klammern unterhalb des *new*-Statements. Wir sehen hier **keine Angabe einer expliziten implements-Beziehung!!**

Die Klassenhierarchie des oben erläuterten, anonymenListeners kann folgendermaßen skizziert werden:

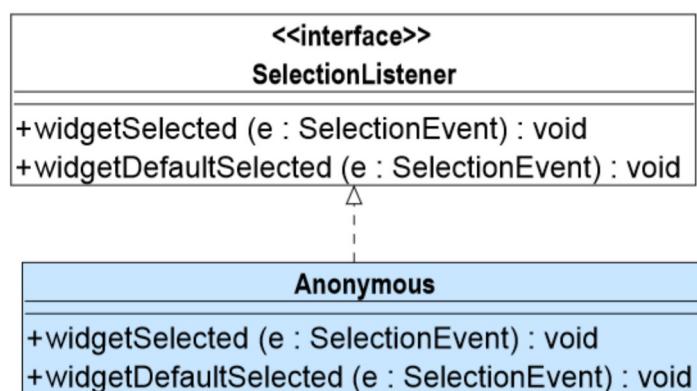


Abbildung 2.8: Vererbungshierarchie einer anonymen *SelectionListener*-Implementierung

Bei unserem *Cancel*-Button erfolgt die Listener-Implementierung mit Hilfe der Adapter-Klasse *SelectionAdapter*:

```

1  Button buttonCancel = new Button(shell, SWT.PUSH);
2  buttonCancel.addSelectionListener(
3      new SelectionAdapter(){
4          @Override
5          public void widgetSelected(
6              SelectionEvent arg0) {
7                  // Aussen-Shell schliessen
8                  // und zerstoeren
9                  shell.dispose();
10
11             } // end method
12     } // end class
13 ); // end call .addSelectionListener()

```

Abbildung 2.9: Listener-Implementierung des Cancel-Button

Wir sehen hier, dass wir nur noch eine der beiden Reaktionsmethoden implementiert haben, da die Klasse *SelectionAdapter* bereits vorgefertigte „Dummy“-Reaktionsmethoden besitzt.

KAPITEL 2. LISTENER UND EVENTS

Die Klassenhierarchie sieht bei der anonymen Listenerklasse des *Cancel*-Buttons wie folgt aus:

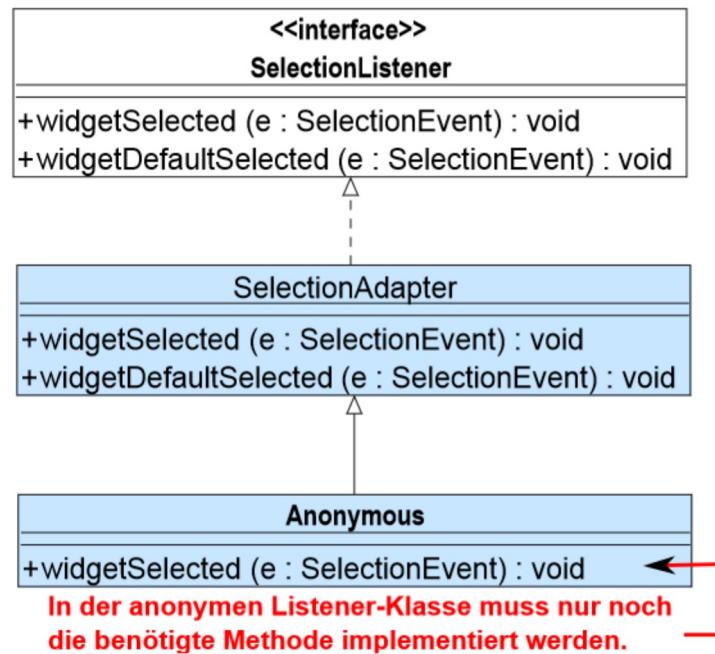


Abbildung 2.10: Vererbungshierarchie einer anonymen `SelectionAdapter`-Kindklasse

Abschließend zeigen wir den vollständigen Code von Klasse *Beispiel2*:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.widgets.Button;
4 import org.eclipse.swt.widgets.Display;
5 import org.eclipse.swt.widgets.Label;
6 import org.eclipse.swt.widgets.Shell;
7 import org.eclipse.swt.events.SelectionEvent;
8 import org.eclipse.swt.events.SelectionListener;
9 import org.eclipse.swt.events.SelectionAdapter;
10
11 public class Beispiel2 {
12
13     // Beispiel 2: Ein Label und
14     // 2 Buttons mit Listenern in
15     // einer Shell
16     public static void main(String[] args) {
17
18         // Groessenangaben fuer umrahmende Shell
19         int widthShell=400, heightShell=200;
20
21         // Positionsangaben fuer umrahmende Shell
22         // (linke obere Ecke)
23         int xShell = 100, yShell=200;
24
25         // Groessenangaben fuer Buttons und Label
26         int widthChildren,
27             heightLabel,
28             heightButtons;
29
30         // 1. Display-Objekt erstellen
31         Display display = new Display();
32
33         // 2. Umrahmende Shell erstellen
34         final Shell shell = new Shell(display);
35
36         // 3. Position und Groesse der Shell
37         // erstellen
38         shell.setBounds(xShell, yShell,
39                         widthShell, heightShell);
```

KAPITEL 2. LISTENER UND EVENTS

```
40
41     // 4. Groesseneinstellungen fuer
42     // Kind-Widgets der Shell berechnen:
43     // Alle Kind-Widgets sind so breit wie
44     // das Darstellungsfeld der Shell
45     // Das Label beansprucht 1/2 Hoehe
46     // der Shell
47     // Die Buttons beanspruchen je 1/4 Hoehe
48     // der Shell
49     widthChildren =
50         shell.getClientArea().width;
51     heightLabel =
52         shell.getClientArea().height/2;
53     heightButtons =
54         shell.getClientArea().height/4;
55
56     // 5. Ein einfaches Textlabel erzeugen und
57     // diesem einen Font mitgeben
58     final Label label =
59         new Label(shell, SWT.LEFT);
60     label.setFont(
61         new Font(display, "Arial", 18, SWT.BOLD));
62
63     // 6. Label ganz oben in die Shell setzen
64     // Label beansprucht die "halbe Hoehe"
65     // der Shell und ist so breit wie diese
66     label.setBounds(0, 0,
67                     widthChildren, heightLabel);
68
69     // 7. Einen Ok-Button erzeugen und
70     // unter das Label platzieren
71     Button buttonOk =
72         new Button(shell, SWT.PUSH);
73     buttonOk.setFont(
74         new Font(display, "Arial", 18, SWT.BOLD));
75     buttonOk.setBounds(0, heightLabel,
76                     widthChildren, heightButtons);
77     buttonOk.setText("Ok");
```

```
78
79 // 8. Dem Ok-Button einen Listener mitgeben
80 buttonOk.addSelectionListener(
81     new SelectionListener(){
82         // Eine einfache Zaehl-Variable
83         int i=0;
84
85         @Override
86         public void widgetDefaultSelected(
87             SelectionEvent arg0) {
88             // tue nichts -- nur wegen Interface-
89             // Vorgabe eingefuegt
90         }
91
92         @Override
93         public void widgetSelected(
94             SelectionEvent arg0) {
95             // Zaehler inkrementieren
96             i++;
97             // Setze den Zaehler in das oben
98             // stehende Label ein
99             label.setText("Wert:" + i);
100        }
101
102   }); // end Listener
103
104 // 9. Einen "Cancel-Button" erzeugen und
105 // unter den Ok-Button platzieren
106 Button buttonCancel = new Button(
107     shell, SWT.PUSH);
108 buttonCancel.setFont(
109     new Font(display, "Arial",
110             18, SWT.BOLD));
111 buttonCancel.setBounds(
112     0, heightLabel+heightButtons,
113     widthChildren, heightButtons);
114 buttonCancel.setText("Cancel");
```

KAPITEL 2. LISTENER UND EVENTS

```
115
116     // 10. Dem Ok-Button einen Listener
117     // mitgeben
118     buttonCancel.addSelectionListener(
119         new SelectionAdapter() {
120
121             @Override
122             public void widgetSelected(
123                 SelectionEvent arg0) {
124
125                 // Shell ausblenden und
126                 // zerstoeren
127                 shell.dispose();
128             }
129         ); // end Listener
130
131     // 11. Die Shell "sichtbar" gesetzt
132     shell.open();
133
134     // 12. Hier wird die (blockierende)
135     // Event-Loop gestartet.
136     while(!shell.isDisposed()) {
137         if(!display.readAndDispatch()){
138             display.sleep();
139         } // end if
140     } // end while
141 } // end main()
142 } // end class Beispiel2
```

Merke

- **Anonyme Listener** sind eine spezielle Art von **Inner Classes**.
- Sie werden mit

```
new InterfaceName() { ... Inhalt der anonymen Klasse ... }
```

erzeugt und direkt ausprogrammiert.
- Sie werden gern zum Testen von Oberflächenprototypen verwendet. In großen Programmsystemen führen sie jedoch zu unübersichtlichem Code.

2.3 3-Schichtenmodell mit Listenern

Bislang haben wir das 3-Schichtenmodell nur theoretisch kennengelernt. Wir haben es noch nicht in einem konkreten Widget-Set, wie beispielsweise SWT angewendet. Wir werden unser bislang in einer einfachen *main()*-Methode untergebrachtes Beispiel ein wenig modifizieren:

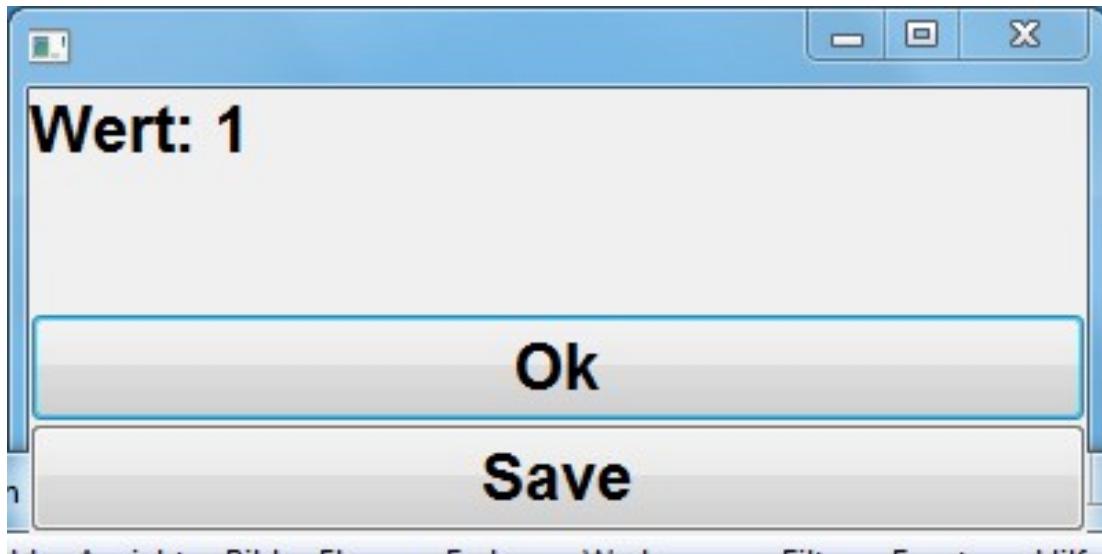


Abbildung 2.11: Oberfläche *Beispiel3*

Der untere Button wurde modifiziert – er speichert nun den Label-Inhalt in einer Datei in unserem Home- Verzeichnis ab.

Die Software-Architektur wurde nun stark modifiziert:

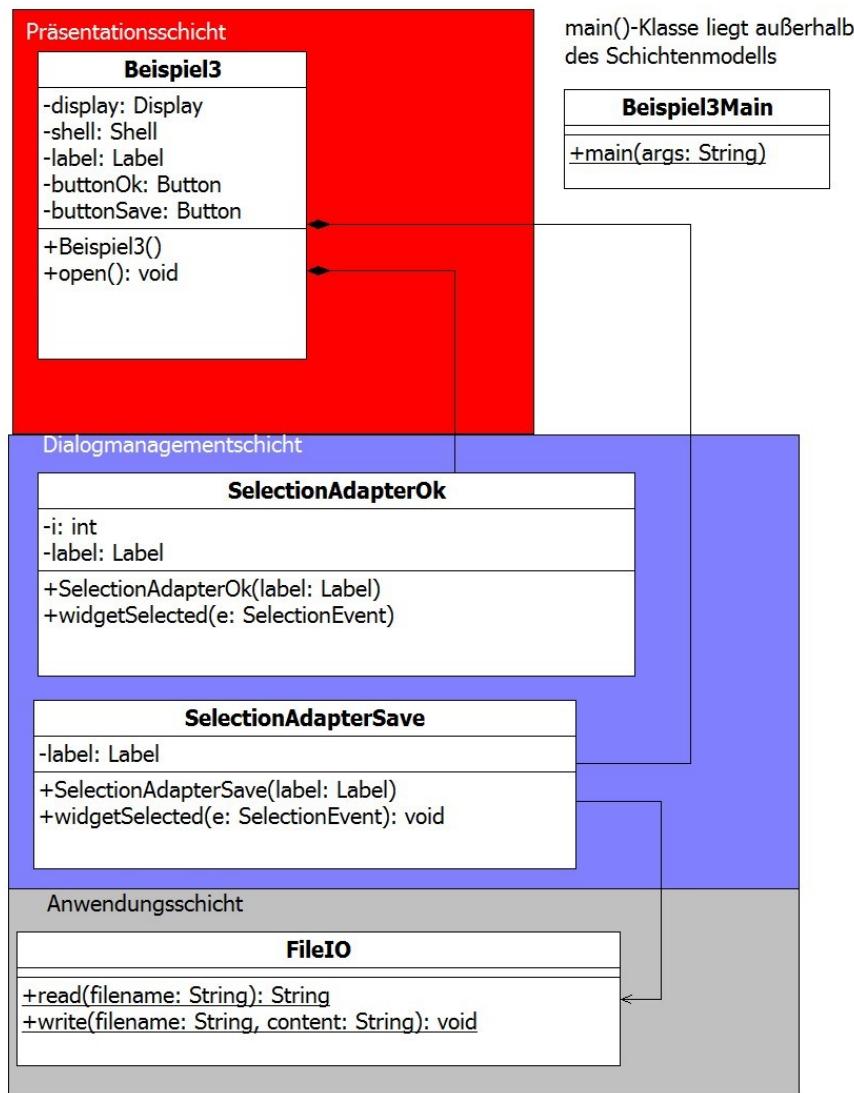


Abbildung 2.12: Softwarearchitektur eines 3-Schichtenmodells im SWT

Das Klassendiagramm deutet bereits an, dass hier zwischen Oberflächencode (Präsentationsschicht + Dialogmanagementschicht) und Nicht-Oberflächencode (Anwendungsschicht mit Dateizugriffen) strikt getrennt wird.

KAPITEL 2. LISTENER UND EVENTS

Wir sehen nun folgende Klassen:

- Klasse *Beispiel3* enthält nun die **Präsentationsschicht**. Sie enthält folgende Bestandteile:
 - Shell, Display, sowie das Label und die Buttons. Sie sind als private Objekt-Variablen untergebracht.
 - Einen *public* Konstruktor. Er übernimmt die Erzeugung von Display und Shell, sowie der in der Shell platzierten Widgets.
 - Eine Methode *open()*. Sie enthält die EventLoop.
- Die **Dialogmanagementschicht** enthält nun die Klassen *SelectionAdapterSave* und *SelectionAdapterOk*. Sie implementieren unsere nun nicht mehr anonymen Listener.
- Die **Anwendungsschicht** enthält eine Dateizugriffsklasse *FileIO*. Sie ist für das Abspeichern des Label-Inhaltes verantwortlich. Ihre Lese- und Schreibmethoden sind statisch.

Wir erläutern im Folgenden die Quelltexte der einzelnen Schichten.

2.3.1 Präsentationsschicht

Die Präsentationsschicht enthält in diesem Beispiel nur die Klasse *Beispiel3*. Alle Widgets der Oberfläche sind in dieser Klasse als private Objektvariablen angelegt:

```

1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.widgets.Button;
4 import org.eclipse.swt.widgets.Display;
5 import org.eclipse.swt.widgets.Label;
6 import org.eclipse.swt.widgets.Shell;
7 import org.eclipse.swt.events.SelectionEvent;
8 import org.eclipse.swt.events.SelectionListener;
9 import org.eclipse.swt.events.SelectionAdapter;
10
11 public class Beispiel3 {
12
13     private Display display;
14     private Shell shell;
15     private Label label;
16     private Button buttonOk;
17     private Button buttonSave;

```

Der Konstruktor ist relativ umfangreich, da er die vollständige Widget-Hierarchie erzeugt und das gesamte Layout aufbaut. Zunächst erzeugt er Display und Shell:

```

18     public Beispiel3(){
19         // Groessenangaben fuer umrahmende Shell
20         int widthShell=400, heightShell=200;
21
22         // Positionsangaben fuer umrahmende Shell
23         // (linke obere Ecke)
24         int xShell = 100, yShell=200;
25
26         // Groessenangaben fuer Buttons und Label
27         int widthChildren,
28             heightLabel,
29             heightButtons;
30
31         // 1. Display-Objekt erstellen
32         display = new Display();
33
34         // 2. Umrahmende Shell erstellen
35         shell = new Shell(display);
36

```

KAPITEL 2. LISTENER UND EVENTS

```
37     // 3. Position und Groesse der Shell  
38     // erstellen  
39     shell.setBounds(xShell, yShell,  
40                       widthShell, heightShell);
```

Für die Shell werden wieder nach der Erzeugung mit der ererbten Methode `setBounds(...)` die Größeneinstellungen vorgenommen.

Im folgenden Codeabschnitt werden die Größen- und Positionsangaben für Label und Buttons berechnet. Danach wird zunächst das Label erzeugt, beschriftet und positioniert:

```
42     // 4. Groesseneinstellungen fuer  
43     // Kind-Widgets der Shell berechnen:  
44     // Alle Kind-Widgets sind so breit wie  
45     // das Darstellungsfeld der Shell  
46     // Das Label beansprucht 1/2 Hoehe  
47     // der Shell  
48     // Die Buttons beanspruchen je 1/4 Hoehe  
49     // der Shell  
50     widthChildren =  
51         shell.getClientArea().width;  
52     heightLabel =  
53         shell.getClientArea().height/2;  
54     heightButtons =  
55         shell.getClientArea().height/4;  
56  
57     // 5. Ein einfaches Textlabel erzeugen und  
58     // diesem einen Font mitgeben  
59     label = new Label(shell, SWT.LEFT);  
60     label.setFont(  
61         new Font(display, "Arial", 18, SWT.BOLD));  
62  
63     // 6. Label ganz oben in die Shell setzen  
64     // Label beansprucht die "halbe Hoehe"  
65     // der Shell und ist so breit wie diese  
66     label.setBounds(0, 0,  
67                     widthChildren, heightLabel);
```

Im letzten Abschnitt des Konstruktors werden die Buttons erzeugt, beschriftet und positioniert:

```

68
69     // 7. Einen Ok-Button erzeugen und
70     // unter das Label platzieren
71     buttonOk = new Button(shell, SWT.PUSH);
72     buttonOk.setFont(
73         new Font(display, "Arial", 18, SWT.BOLD));
74     buttonOk.setBounds(0, heightLabel,
75         widthChildren, heightButtons);
76     buttonOk.setText("Ok");
77
78     // 8. Dem Ok-Button einen NICHT-ANONYMEN
79     // Listener mitgeben
80     buttonOk.addSelectionListener(
81         new SelectionAdapterOk(label));
82
83     // 9. Einen "Cancel-Button" erzeugen und
84     // unter den Ok-Button platzieren
85     buttonSave = new Button(shell, SWT.PUSH);
86     buttonSave.setFont(
87         new Font(display, "Arial",
88             18, SWT.BOLD));
89     buttonSave.setBounds(
90         0, heightLabel+heightButtons,
91         widthChildren, heightButtons);
92     buttonSave.setText("Save");
93
94     // 10. Dem Ok-Button einen NICHT-ANONYMEN
95     // Listener mitgeben
96     buttonSave.addSelectionListener(
97         new SelectionAdapterSave(label));
98 } // end constructor Beispiel3()

```

Beide Buttons bekommen nun Objekte der beiden ***nicht anonymen*** Listener der Dialogmanagement-Schicht mit.

KAPITEL 2. LISTENER UND EVENTS

Der nächste und letzte Abschnitt der Klasse *Beispiel3* zeigt die ausgelagerte EventLoop in der Methode *open()*.

```
99 // Ausgelagerte Event-Loop
100 public void open() {
101     // 11. Die Shell wird auf "sichtbar"
102     // geschaltet
103     shell.open();
104
105     // 12. Hier wird die (blockierende)
106     // Event-Loop gestartet.
107     while(!shell.isDisposed()) {
108         if(!display.readAndDispatch()){
109             display.sleep();
110         } // end if
111     } // end while
112 } // end method open()
113
114 } // end class Beispiel3
```

Da *Shell*-Referenz und *Display*-Referenz nun als Objektvariablen angelegt sind, die im Konstruktor initialisiert wurden, kann die Methode problemlos auf die beiden Variablen zugreifen.

Wir zeigen nun noch die *main()*-Klasse, welche nur noch ein *Beispiel3*-Objekt erzeugt und dessen Methode *open()* startet:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.widgets.Button;
4 import org.eclipse.swt.widgets.Display;
5 import org.eclipse.swt.widgets.Label;
6 import org.eclipse.swt.widgets.Shell;
7 import org.eclipse.swt.events.SelectionEvent;
8 import org.eclipse.swt.events.SelectionListener;
9 import org.eclipse.swt.events.SelectionAdapter;
10
11 public class Beispiel3Main {
12
13     // Beispiel 3: SWT-Oberflaeche mit
14     // ausgelagerter Praesentationsschicht
15     public static void main(String[] args) {
16
17         // Oberflaeche mit allen Kind-Widgets
18         // neu erzeugen
19         Beispiel3 oberflaeche = new Beispiel3();
20
21         // Event-Loop starten
22         oberflaeche.open();
23
24     } // end main()
25 } // end class Beispiel3Main
```

Die *main()*-Klasse gehört zwar nicht wirklich zur Präsentationsschicht, arbeitet aber eng mit ihr zusammen. Daher wurde sie an dieser Stelle gezeigt.

2.3.2 Dialogmanagementschicht

Die Dialogmanagementschicht enthält 2 Listenerklassen. Beide bekommen über ihren Konstruktor die Widget-Referenzen übergeben, auf die sie während ihrer Methodenaufrufe zugreifen müssen. In unserem Fall ist das die *Label*-Referenz.

```
1 import org.eclipse.swt.events.SelectionAdapter;
2 import org.eclipse.swt.events.SelectionEvent;
3 import org.eclipse.swt.widgets.Label;
4
5 public class SelectionAdapterOk
6         extends SelectionAdapter {
7
8     private int i=0;
9     private Label label;
10
11    // Die aufrufende Klasse reicht hier
12    // die Referenz auf ihr Textlabel
13    // hinein.
14    public SelectionAdapterOk(Label label) {
15        this.label = label;
16    } // end constructor
17
18    // Wir ueberschreiben nur die Reaktions-
19    // methode, die wir benoetigen
20    public void widgetSelected(
21        SelectionEvent e){
22        i++;
23
24        // Setze den Zaehler in das
25        // uebergebene Label ein
26        label.setText("Wert:" + i);
27
28    } // end method widgetSelected()
29 } // end class SelectionAdapterOk
```

Wir sehen, dass die Klasse von *SelectionAdapter* abgeleitet ist und den Namen *SelectionAdapterOk* hat. Aus diesem Grund registriert der *Ok*-Button aus Klasse *Beispiel3* den Listener etwas anders als in *Beispiel2*:

```
buttonOk.addSelectionListener(
    new SelectionAdapterOk(label));
```

Der zweite Listener implementiert einen Dateizugriff:

```

1 import java.io.File;
2 import java.io.FileWriter;
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import org.eclipse.swt.events.SelectionAdapter;
6 import org.eclipse.swt.events.SelectionEvent;
7 import org.eclipse.swt.widgets.Label;
8
9 public class SelectionAdapterSave
10         extends SelectionAdapter {
11
12     private Label label;
13
14     // Die aufrufende Klasse reicht hier
15     // die Referenz auf ihr Textlabel hinein.
16     public SelectionAdapterSave(Label label) {
17         this.label = label;
18     } // end constructor
19
20     // Wir ueberschreiben nur die Reaktions-
21     // methode, die wir benoetigen
22     public void widgetSelected(
23         SelectionEvent e){
24
25         // Text aus Label abfragen
26         String textToSave = label.getText();
27
28         // Dateiname bilden: z.B.
29         // /home/matecki/beispiel3.txt
30         String fileName =
31             System.getProperty("user.home") +
32             File.separator + "beispiel3.txt";
33
34         // Inhalt des Labels auf Datei
35         // schreiben: Hier benutzen wir
36         // bereits die Anwendungsschicht
37         FileIO.write(fileName, textToSave);
38
39     } // end method widgetSelected()
40 } // end class SelectionAdapterSave

```

Der Dateizugriff ist hier bereits in der Klasse *FileIO* gekapselt, welche wir im folgenden

Kapitel 2.3.3 betrachten.

2.3.3 Anwendungsschicht

Die Anwendungsschicht besteht hier aus der Klasse *FileIO*. Sie enthält zwei statische Methoden *read(...)* und *write(...)*, welche von einer einfachen Textdatei lesen bzw. auf eine einfache Textdatei schreiben.

```
1 import java.io.BufferedReader;
2 import java.io.FileNotFoundException;
3 import java.io.FileReader;
4 import java.io.FileWriter;
5 import java.io.IOException;
6
7 public class FileIO {
8
9     // Dateiinhalt zeilenweise lesen
10    // und als String zurueckliefern
11    public static String read(String filename) {
12        String content = "";
13        String line;
14        try {
15
16            // Datei oeffnen
17            BufferedReader filereader =
18                new BufferedReader(
19                    new FileReader(filename));
20
21            // Datei zeilenweise lesen
22            while ((line = filereader.readLine())
23                  != null){
24                content += line + '\n';
25            } // end while
26
27            // Datei schliessen
28            filereader.close();

```

```
29      } catch (FileNotFoundException e) {
30          e.printStackTrace();
31      } catch (IOException e) {
32          e.printStackTrace();
33      } // end catch
34
35      // Dateiinhalt zurueckliefern
36      return content;
37  } // end method read()
38
39
40  public static void write(String filename,
41                          String content){
42
43      try {
44          // Datei oeffnen
45          FileWriter writer =
46              new FileWriter(filename);
47
48          // Inhalt aus zweitem Parameter
49          // auf Datei schreiben
50          writer.write(content);
51
52          // Datei schliessen
53          writer.close();
54      } catch (IOException e) {
55          // TODO Auto-generated catch block
56          e.printStackTrace();
57      } // end catch
58
59  } // end method write()
60 } // end class FileIO
```

Merke

Das 3-Schichtenmodell wird für SWT-Oberflächen folgendermaßen umgesetzt:

- Die **Präsentationsschicht** ist eine Klasse, welche die darzustellenden Widgets und Listener als *Komposition* enthält. Sie enthält ausserdem eine Methode `open()` mit der EventLoop.
- Die **Dialogmanagementschicht** besteht aus den Listenern der Oberfläche.
- Entgegen dem Eindruck, den viele Lehrbuchbeispiele hervorrufen, werden anonyme Listener in großen Softwarepaketen weniger für ganze Workflows benutzt, da sie in diesem Fall Präsentationsschicht und Dialogmanagementschicht mischen. Für kurze Codesequenzen, die sich nicht wiederholen, sind sie jedoch sinnvoll.
- Existentiell wichtig ist es, Dateizugriffe, Algorithmen, Netzwerkzugriffe und ähnliches aus dem Oberflächencode herauszuhalten. Hier gilt: **Trennung zwischen Oberfläche und oberflächenunabhängigem Code!**
- Umgesetzt wird diese Trennung durch eine Schicht mit eigenen Klassen, welche die oberflächenunabhängigen Zugriffe und Algorithmen kapseln.

Kapitel 3

Layout-Manager

Wir haben in den Beispielen 1-3 die Platzierung unserer Widgets innerhalb ihrer Shell mit der von Klasse *Widget* geerbten *setBounds(...)*-Methode bewerkstelligt. Für einige, wenige Kind-Widgets mag dies in Ordnung sein. Sobald Sie jedoch komplexere Oberflächen, wie beispielsweise einen Taschenrechner, entwickeln wollen, wird diese Vorgehensweise mühsam und fehleranfällig.

Daher bieten die meisten Widget-Sets Container-Klassen an, die die Layout-Verwaltung von Kind-Widgets innerhalb eines Eltern-Widgets übernehmen. Diese Container werden auch als **Layout-Manager** bezeichnet. Layout-Manager können innerhalb von Widget-Sets meist auf eine oder beide der folgenden Arten verwendet werden:

- Verwendung von Layout-Manager-Klassen direkt im Code. Auf diese Weise werden wir Layout-Manager im SWT kennenlernen.
- Angabe der Layout-Manager mit Bezug auf die anzuordnenden Kind-Widgets als XML- Konfiguration. Diesen Mechanismus lernen wir im Android-SDK kennen.

Im SWT sind die bekanntesten Layout-Manager-Klassen (vgl. [21], S. 145ff):

KAPITEL 3. LAYOUT-MANAGER

Layout-Manager im SWT	
Name des Layout-Managers	Erläuterung
RowLayout	Ordnet seine Kind-Widgets vertikal oder horizontal an. Die Anordnung wird bei der Erzeugung mit <i>new</i> über die Style-Konstanten <i>SWT.VERTICAL</i> bzw. <i>SWT.HORIZONTAL</i> gesteuert. Die Kind-Widgets werden jedoch nicht „bündig“ in die Breite des Eltern-Widgets gezogen, sondern werden so breit, wie es die Beschriftung erfordert.
FillLayout	Ordnet seine Kind-Widgets ebenfalls vertikal oder horizontal an. Die Anordnung wird auch hier bei der Erzeugung mit <i>new</i> über die Style-Konstanten <i>SWT.VERTICAL</i> bzw. <i>SWT.HORIZONTAL</i> gesteuert. Die Kind-Widgets werden „bündig“ in die Breite des Eltern-Widgets gezogen.
GridLayout	Mit diesem Layout können gitterförmige Layouts („Taschenrechner“) erzeugt werden. Die Zeilenanzahl und Spaltenanzahl kann hier vom Entwickler gesteuert werden. Auch lassen sich z. B. Zellen einer Zeile zusammenfassen.
FormLayout	Mit diesem Layout können Widgets an ihren Rändern aneinandergesetzt bzw. an die Ränder ihres Parents „geklebt“ werden.

Tabelle 3.1: Layout-Manager im SWT

Manchen Layoutmanagern können wir noch zusätzliche Konfigurationsdaten über ein *LayoutTypeData*-Objekt mitgeben. Wir zeigen die SWT-Klassenhierarchie der Layout-Manager-Klassen innerhalb des Packages `org.eclipse.swt.layout`:

Wir sehen, dass sämtliche Layout-Manager-Klassen von der SWT-Klasse *Layout* abgeleitet sind. Sie sind **nicht** direkt oder indirekt von der Klasse *Widget* abgeleitet. Die zugehörigen Konfigurationsklassen

- `FormData`
- `GridData`
- `RowData`

stehen außerhalb der SWT-Klassenhierarchie. Sie befinden sich jedoch, ebenso wie

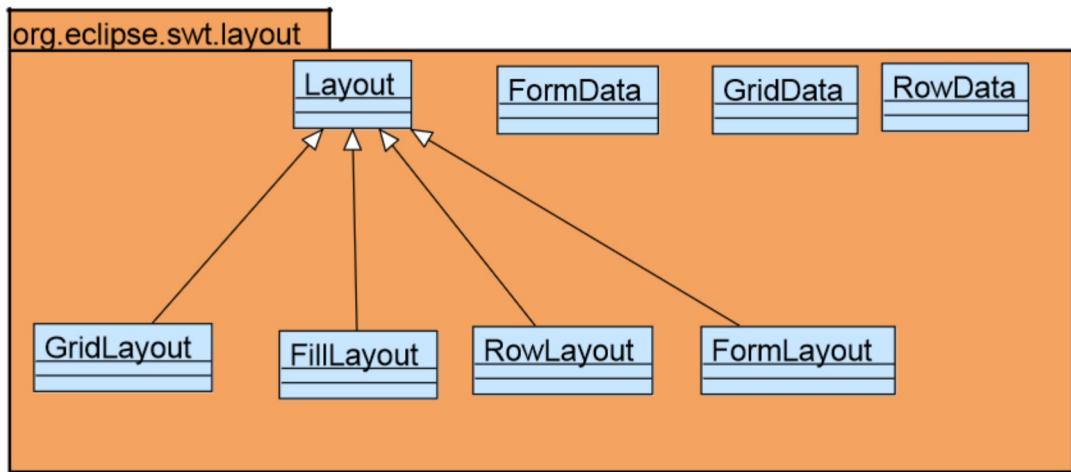


Abbildung 3.1: Klassenhierarchie der Layout-Manager im SWT

die Layoutmanager im Package `org.eclipse.swt.layout`. Wir werden ein Beispiel für diese Konfigurationsklassen beim *RowLayout* kennenlernen.

KAPITEL 3. LAYOUT-MANAGER

Unter SWT werden Layoutmanager immer nach folgendem Muster verwendet:

Schritt 1: Erzeugung des Layoutmanagers:

```
LayoutTyp layout = new LayoutTyp(...);
```

Über den Konstruktor werden dem Layoutmanager noch Einstellungsparameter mitgegeben, die für **alle** anzuordnenden Kindwidgets gelten.

Schritt 2: Layout dem umrahmenden Fenster mitgeben:

```
parent.setLayout(layout);
```

Nach diesem Schritt werden alle Kindwidgets der Shell, der der Layoutmanager übergeben wurde, automatisch von diesem gemäß seiner Konfiguration angeordnet.

Schritt 3: Kindwidgets erzeugen und ggf. mit Layoutdaten parametrisieren

```
WidgetTyp kind1 = new WidgetTyp(parent,...);
```

Falls das Kind-Widget noch zusätzliche Angaben zur Einordnung ins Layout benötigt, bekommt es Layout-Daten mit:

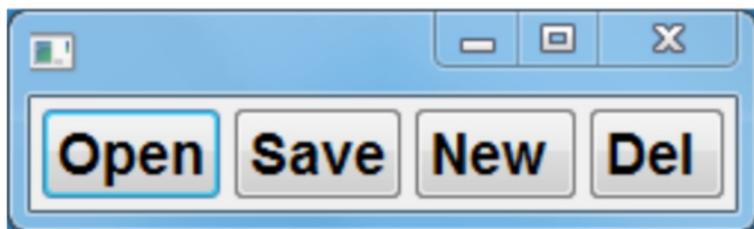
```
LayoutTypData layoutData = new LayoutTypData(...);  
kind1.setLayoutData(layoutData);
```

3.1 RowLayout

Ein *RowLayout* ist der einfachste aller Layout-Manager. Er kann die Kind-Widgets eines umrahmenden Widgets

- vertikal untereinander (Typ *SWT.VERTICAL*) oder
- horizontal nebeneinander (Typ *SWT.HORIZONTAL*)

ausrichten:



Horizontale Anordnung im RowLayout



Vertikale Anordnung im RowLayout

Abbildung 3.2: Auswirkung RowLayout

Die Kind-Widgets werden dabei genau so groß, wie es die Beschriftung erfordert. Sie wachsen und schrumpfen **nicht** mit dem Elternwidget mit.

KAPITEL 3. LAYOUT-MANAGER

Klasse *Beispiel4* zeigt die Implementierung des Layouts:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.layout.RowLayout;
4 import org.eclipse.swt.widgets.Button;
5 import org.eclipse.swt.widgets.Display;
6 import org.eclipse.swt.widgets.Shell;
7
8 public class Beispiel4 {
9
10    private Display display;
11    private Shell shell;
12
13    // 4 Buttons, die in einem Layout angeordnet
14    // werden
15    private Button buttonOpen;
16    private Button buttonSave;
17    private Button buttonNew;
18    private Button buttonDelete;
19
20    public Beispiel4(){
21
22        // RowLayout wird nur lokal von der
23        // Shell benoetigt
24        // SWT.HORIZONTAL ordnet nebeneinander an
25        // SWT.VERTICAL ordnet untereinander an.
26        // Seine Widgets werden gleich hoch und so
27        // breit wie notwendig (abhaengig von
28        // der Beschriftung)
29        RowLayout layout =
30            new RowLayout(SWT.VERTICAL);
```

Die Kind-Widgets sind wie immer als Objektvariablen in Klasse *Beispiel4* untergebracht. Das *RowLayout* wird nur zur Erstellung der Oberfl  e im Konstruktor ben  tigt. Diese Referenz ist daher lokal.

```
31     // 1. Display-Objekt erstellen
32     display = new Display();
33
34     // 2. Umrahmende Shell erstellen
35     shell = new Shell(display);
36
37     // 3. Der Shell den Layout-Manager
38     // mitgeben
39     shell.setLayout(layout);
```

Die Erzeugung von *Display* und *Shell* funktioniert wie immer. Diesmal wird jedoch der *Shell* das vorher erzeugte *RowLayout* mitgegeben.

```
40
41     // 4. Einen "Open-Button" erzeugen
42     buttonOpen = new Button(shell, SWT.PUSH);
43     buttonOpen.setFont(
44         new Font(display, "Arial", 14, SWT.BOLD));
45     buttonOpen.setText("Open");
46
47     // 5. Einen "Cancel-Button" erzeugen
48     buttonSave = new Button(shell, SWT.PUSH);
49     buttonSave.setFont(
50         new Font(display, "Arial",
51                 14, SWT.BOLD));
52     buttonSave.setText("Save");
53
54     // 6. Einen "New-Button" erzeugen
55     buttonNew = new Button(shell, SWT.PUSH);
56     buttonNew.setFont(
57         new Font(display, "Arial",
58                 14, SWT.BOLD));
59     buttonNew.setText("New");
60
61     // 7. Einen "Del-Button" erzeugen
62     buttonDelete = new Button(shell, SWT.PUSH);
63     buttonDelete.setFont(
64         new Font(display, "Arial",
65                 14, SWT.BOLD));
66     buttonDelete.setText("Del");
```

KAPITEL 3. LAYOUT-MANAGER

```
67
68     // 8. Shell um ihren Layout-Manager
69     // herumpacken: So sind die Shell-Grenzen
70     // buendig um die Buttons gelegt.
71     shell.pack();
72 } // end constructor Beispiel4()

73

74 // Ausgelagerte Event-Loop
75 public void open() {
76     // ... wie immer ...
77     // 9. Die Shell wird auf "sichtbar"
78     // geschaltet
79     shell.open();
80
81     // 9. Hier wird die (blockierende)
82     // Event-Loop gestartet.
83     while(!shell.isDisposed()) {
84         if(!display.readAndDispatch()){
85             display.sleep();
86         } // end if
87     } // end while
88 } // end method open()
89 } // end class Beispiel3
```

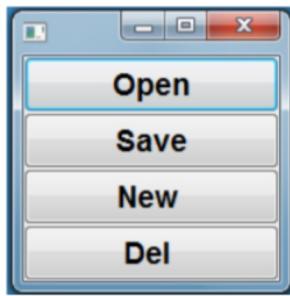
Die in der Shell zu platzierenden Buttons werden nur noch erzeugt. Sie werden nun **nicht** mehr „von Hand“ mit `setBounds(...)` ausgerichtet, sondern vom Layout-Manager. Die Klasse `Beispiel4` enthält noch ihre ausgelagerte EventLoop in der Methode `open()`. Ansonsten sind keine weiteren Elemente in dieser Klasse enthalten. Die `main()`-Klasse ist ähnlich aufgebaut wie in den vorangegangenen Beispielen und wird daher nicht mehr gezeigt.

3.2 FillLayout

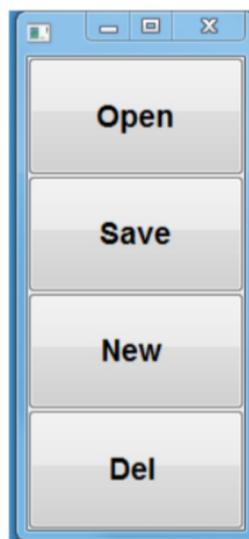
Ein *FillLayout* funktioniert ähnlich einfach wie ein *RowLayout*. Er kann die Kind-Widgets eines umrahmenden Widgets ebenfalls

- vertikal untereinander (Typ *SWT.VERTICAL*) oder
- horizontal nebeneinander (Typ *SWT.HORIZONTAL*)

ausrichten. Allerdings wachsen und schrumpfen die Kind-Widgets mit dem Parent-Widget mit:



FillLayout:



Kind-Widgets
wachsen und
schrumpfen
mit dem Parent-
Widget

Abbildung 3.3: Auswirkung FillLayout

KAPITEL 3. LAYOUT-MANAGER

Klasse *Beispiel5* zeigt die Implementierung des Layouts:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.layout.FillLayout;
4 import org.eclipse.swt.widgets.Button;
5 import org.eclipse.swt.widgets.Display;
6 import org.eclipse.swt.widgets.Shell;
7
8 public class Beispiel5 {
9
10    private Display display;
11    private Shell shell;
12
13    // 4 Buttons, die in einem Layout angeordnet
14    // werden
15    private Button buttonOpen;
16    private Button buttonSave;
17    private Button buttonNew;
18    private Button buttonDelete;
```

Die Referenzen auf *Display*, *Shell* und die Kind-Widgets sind auch hier als Objektvariablen in der Klasse *Beispiel5* untergebracht. Das *FillLayout* wird, wie vorher das *RowLayout* nur lokal im Konstruktor benötigt.

```
20 public Beispiel5(){
21
22    // FillLayout wird nur lokal von der
23    // Shell benoetigt
24    // SWT.HORIZONTAL ordnet nebeneinander an
25    // SWT.VERTICAL ordnet untereinander an.
26    FillLayout layout =
27        new FillLayout(SWT.VERTICAL);
28
29    // 1. Display-Objekt erstellen
30    display = new Display();
31
32    // 2. Umrahmende Shell erstellen
33    shell = new Shell(display);
34
35    // 3. Der Shell den Layout-Manager
36    // mitgeben
37    shell.setLayout(layout);
38
39    // 4. Einen "Open-Button" erzeugen
```

```

40     buttonOpen = new Button(shell, SWT.PUSH);
41     buttonOpen.setFont(
42         new Font(display, "Arial", 14, SWT.BOLD));
43     buttonOpen.setText("Open");
44
45     // 5. Einen "Cancel-Button" erzeugen
46     buttonSave = new Button(shell, SWT.PUSH);
47     buttonSave.setFont(
48         new Font(display, "Arial",
49             14, SWT.BOLD));
50     buttonSave.setText("Save");
51
52     // 6. Einen "New-Button" erzeugen
53     buttonNew = new Button(shell, SWT.PUSH);
54     buttonNew.setFont(
55         new Font(display, "Arial",
56             14, SWT.BOLD));
57     buttonNew.setText("New");
58
59     // 7. Einen "Del-Button" erzeugen
60     buttonDelete = new Button(shell, SWT.PUSH);
61     buttonDelete.setFont(
62         new Font(display, "Arial",
63             14, SWT.BOLD));
64     buttonDelete.setText("Del");
65
66     // 8. Shell um ihren Layout-Manager
67     // herumpacken: So sind die Shell-Grenzen
68     // buendig um die Buttons gelegt.
69     shell.pack();
70 } // end constructor Beispiel5

71
72     // Ausgelagerte Event-Loop
73     public void open() {
74         // wie immer ...

75     } // end method open()
76 } // end class Beispiel5

```

Die Konstruktorimplementierung funktioniert gleich, wie in *Beispiel4*. Der einzige Unterschied ist, dass hier ein *FillLayout* statt eines *RowLayouts* erzeugt wird, welches der Shell mitgegeben wird. Die EventLoop sieht gleich aus, wie in den vorangegangenen Beispielen und wird deshalb nicht noch einmal gezeigt. Das gleiche gilt für die *main()*-Klasse.

3.3 GridLayout

Mit einem *GridLayout* ist es möglich, Widgets gitterförmig anzurichten, so dass Oberflächen folgender Gestalt möglich werden:



Abbildung 3.4: Screenshot Beispiel6 mit *GridLayout*

In diesem Beispiel haben wir ein 2-spaltiges *GridLayout* verwendet. Für das Textlabel in der dritten „Zeile“ des Layouts haben wir die 2 Spalten zusammengefaßt, so dass es sich über beide Spalten erstreckt. Die Befüllung eines *GridLayout* erfolgt – ähnlich wie bei *RowLayout* und *FillLayout* automatisiert. Die Reihenfolge ist hierbei **zeilenweise von links nach rechts**.

Beispiel: Befüllung eines
2-spaltigen GridLayout:

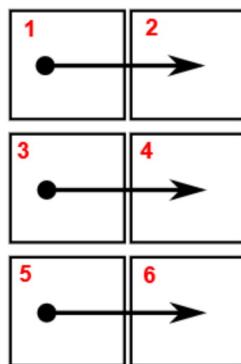


Abbildung 3.5: Befüllungsreihenfolge *GridLayout*

Sofern den einzelnen Kind-Widgets noch zusätzliche Layouteinstellungen mitgegeben werden sollen, erstellen wir für das betroffene Kind ein zusätzliches *GridData*-Objekt, welches diese Layoutinformationen kapselt.

Achtung

Jedes Kind-Widget, welches zusätzliche Layout-Einstellungen benötigt, bekommt ein **eigenes GridData-Objekt!**

Wir diskutieren den Quelltext zu *Beispiel6*:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Color;
3 import org.eclipse.swt.graphics.Font;
4 import org.eclipse.swt.layout.GridData;
5 import org.eclipse.swt.layout.GridLayout;
6 import org.eclipse.swt.widgets.Button;
7 import org.eclipse.swt.widgets.Display;
8 import org.eclipse.swt.widgets.Label;
9 import org.eclipse.swt.widgets.Shell;
10
11 public class Beispiel6 {
12
13     private Display display;
14     private Shell shell;
15
16     // 4 Buttons, die in einem Layout angeordnet
17     // werden
18     private Button buttonOpen;
19     private Button buttonSave;
20     private Button buttonNew;
21     private Button buttonDelete;
22     private Label label;
```

Der obere Teil der Klassenvereinbarung unterscheidet sich nicht von den vorherigen Beispielen.

KAPITEL 3. LAYOUT-MANAGER

```
23 public Beispiel6(){  
24  
25     // Farben fuer das Label:  
26     // rot = 255, gruen = 255, blau = 255  
27     // --> liefert Farbe weiss  
28     int rot=255, gruen=255, blau=255;  
29  
30     // Anzahl der Spalten des GridLayouts  
31     int noOfCols=2;  
32  
33     // Sind die Spalten des Layouts  
34     // gleich breit?  
35     boolean sameWidth = true;  
36  
37     // Waechst das Widget mit seiner  
38     // Grid-Zelle mit?  
39     boolean grabExcessHorizontalSpace= true;  
40     boolean grabExcessVerticalSpace=true;  
41  
42     // ueber wie viele Spalten dehnt  
43     // sich das Widget aus?  
44     int horizontalSpan=1;  
45  
46     // ueber wie viele Zeilen dehnt  
47     // sich das Widget aus?  
48     int verticalSpan=1;  
49  
50     // GridLayout wird nur lokal von der  
51     // Shell benoetigt. Es hat 2 gleich breite  
52     // Spalten  
53     GridLayout layout =  
54         new GridLayout(noOfCols, sameWidth);
```

Hier legen wir einige Variablen an, welche später für die *GridData*-Objekte unserer Buttons verwendet werden. Am Ende des Abschnittes wird ein 2-spaltiges *GridLayout* erstellt.

```
55     // GridData fuer Layout-Zusatz-
56     // einstellungen fuer jedes
57     // Kind-Widget;
58     GridData gdata = null;
59
60     // 1. Display-Objekt erstellen
61     display = new Display();
62
63     // 2. Umrahmende Shell erstellen
64     shell = new Shell(display);
65
66     // 3. Der Shell den Layout-Manager
67     // mitgeben
68     shell.setLayout(layout);
69
70     // 4. Erzeugung der Buttons:
71     // Sie benoetigen kein GridData-Objekt ,
72     // da sie einfach standardmaessig ins
73     // 2-spaltige Grid eingesortiert werden:
74
75     // 4.1 Einen "Open-Button" erzeugen
76     buttonOpen = new Button(shell, SWT.PUSH);
77     buttonOpen.setFont(
78         new Font(display, "Arial", 14, SWT.BOLD));
79     buttonOpen.setText("Open");
80     // Button bekommt seine eigenen
81     // Layout-Einstellungen mit:
```

In diesem Teil werden – wie bisher – *Shell* und *Display* erstellt. Danach wird die *Shell* mit dem *GridLayout* verknüpft. Der erste PushButton wird erstellt.

KAPITEL 3. LAYOUT-MANAGER

```
82     gdata =
83         new GridData(SWT.FILL,
84             SWT.TOP, grabExcessHorizontalSpace,
85             grabExcessVerticalSpace,
86             horizontalSpan, verticalSpan );
87     buttonOpen.setLayoutData(gdata);
88
89 // 4.2 Einen "Cancel-Button" erzeugen
90 buttonSave = new Button(shell, SWT.PUSH);
91 buttonSave.setFont(
92     new Font(display, "Arial",
93             14, SWT.BOLD));
94 buttonSave.setText("Save");
95 // Button bekommt seine eigenen
96 // Layout-Einstellungen mit:
97 gdata =
98     new GridData(SWT.FILL,
99             SWT.TOP, grabExcessHorizontalSpace,
100            grabExcessVerticalSpace,
101            horizontalSpan, verticalSpan );
102 buttonSave.setLayoutData(gdata);
103
104 // 4.3 Einen "New-Button" erzeugen
105 buttonNew = new Button(shell, SWT.PUSH);
106 buttonNew.setFont(
107     new Font(display, "Arial",
108             14, SWT.BOLD));
109 buttonNew.setText("New");
```

Bei der Erstellung der Buttons wird nun jedem Button ein eigenes *GridData*-Objekt mit zusätzlichen Ausrichtungs-Angaben mitgegeben. Hier ist es durchaus sinnvoll, die Einstellungsmöglichkeiten aus der API einmal durchzutesten!

```
110     // Button bekommt seine eigenen
111     // Layout-Einstellungen mit:
112     gdata =
113         new GridData(SWT.FILL,
114             SWT.TOP, grabExcessHorizontalSpace,
115             grabExcessVerticalSpace,
116             horizontalSpan, verticalSpan );
117     buttonNew.setLayoutData(gdata);
118
119     // 4.4 Einen "Del-Button" erzeugen
120     buttonDelete = new Button(shell, SWT.PUSH);
121     buttonDelete.setFont(
122         new Font(display, "Arial",
123             14, SWT.BOLD));
124     buttonDelete.setText("Del");
125     // Button bekommt seine eigenen
126     // Layout-Einstellungen mit:
127     gdata =
128         new GridData(SWT.FILL,
129             SWT.TOP, grabExcessHorizontalSpace,
130             grabExcessVerticalSpace,
131             horizontalSpan, verticalSpan );
132     buttonDelete.setLayoutData(gdata);
133
134     // 5. Ein Label erzeugen.
135     label = new Label(shell, SWT.CENTER);
136     label.setFont(new Font(display, "Arial",
137             14, SWT.BOLD));
138     label.setText("2 Spalten");
139
140     // Farbe des Labels setzen
141     label.setBackground(new Color(display,
142             rot, gruen, blau));
```

KAPITEL 3. LAYOUT-MANAGER

```
143     // Label bekommt seine eigenen
144     // Layout-Einstellungen mit. Es
145     // erstreckt sich ueber 2
146     // Spalten
147     horizontalSpan = 2;
148     gdata =
149         new GridData(SWT.FILL,
150             SWT.TOP, grabExcessHorizontalSpace,
151             grabExcessVerticalSpace,
152             horizontalSpan, verticalSpan );
153     label.setLayoutData(gdata);
154
155     // 6. Shell um ihren Layout-Manager
156     // herumpacken: So sind die Shell-Grenzen
157     // buendig um die Buttons gelegt.
158     shell.pack();
159 } // end constructor Beispiel6()
160
161 // Ausgelagerte Event-Loop
162 public void open() {
163     // wie immer ...
164 }
165 } // end class Beispiel6
```

Das Label soll sich über 2 Spalten erstrecken. Daher wird der Wert *horizontalSpan* seines *GridData*-Objektes auf 2 gesetzt. Dieser Wert gibt an, wie viele Spalten das Widgets innerhalb des *GridLayout* ab seiner Position beansprucht. Die *EventLoop* hat sich gegenüber den vorherigen Beispielen nicht geändert. Die *main()*-Klasse hat sich ebenfalls nicht geändert. Daher werden beide nicht noch einmal gezeigt.

3.4 Verschachtelte Layouts mit Group

Wenn wir Oberflächen mit umrahmten **Gruppen** von Widgets gestalten wollen, so benötigen wir ein Widget vom Typ *Group*:



Abbildung 3.6: Screenshot Beispiel7 mit *Group*

Im folgenden Beispiel sind die drei Buttons auf der linken Seite in eine solche *Group* eingerahmt.

Dieses fungiert als Parent der zu gruppierenden Buttons. zu gruppierenden Kind-Widgets. Die *Group* ist ihrerseits Kind der Shell:

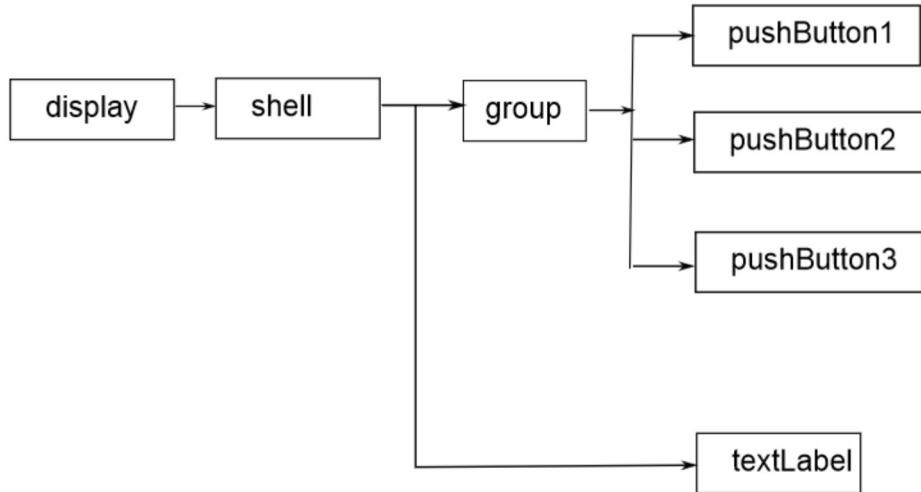


Abbildung 3.7: Widget-Hierarchie von Beispiel7

KAPITEL 3. LAYOUT-MANAGER

Wir diskutieren den Quelltext des GroupBox-Beispiels:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Color;
3 import org.eclipse.swt.graphics.Font;
4 import org.eclipse.swt.layout.FillLayout;
5 import org.eclipse.swt.widgets.Button;
6 import org.eclipse.swt.widgets.Display;
7 import org.eclipse.swt.widgets.Group;
8 import org.eclipse.swt.widgets.Label;
9 import org.eclipse.swt.widgets.Shell;
10
11
12 public class Beispiel7 {
13
14     private Display display;
15     private Shell shell;
16
17     // Verschiedene Buttons: Sie
18     // werden alle ueber die Klasse
19     // Button realisiert.
20     private Button pushButton1;
21     private Button pushButton2;
22     private Button pushButton3;
23     private Label textLabel;
```

Wir benötigen 3 Buttons, sowie einTextLabel.

```
24 public Beispiel7(){  
25  
26     // 2 Layoutmanager erstellen  
27     // Die Shell wird horizontal  
28     // organisiert  
29     // Die GroupBox mit den  
30     // Buttons wird vertikal  
31     FillLayout layoutGroup =  
32         new FillLayout(SWT.VERTICAL);  
33     FillLayout layoutShell =  
34         new FillLayout(SWT.HORIZONTAL);  
35  
36  
37     // 1. Display-Objekt erstellen  
38     display = new Display();  
39  
40     // 2. Umrahmende Shell erstellen  
41     // Ihre Groesse ist nicht veraender-  
42     // bar  
43     shell = new Shell(display);  
44  
45     // 3. Der Shell den horizontalen  
46     // Layout-Manager mitgeben  
47     shell.setLayout(layoutShell);
```

Es werden nun 2 Layout-Manager benötigt:

- *layoutShell*: Ein *FillLayout* für die direkten Kinder der Shell (*Group* und *Label* nebeneinander), und
- *layoutGroup*: Ein *FillLayout* für die Kinder der *Group* (Drei *Buttons* untereinander).

Die *Shell* wird mit dem vertikalen Layout *layoutShell* verknüpft.

KAPITEL 3. LAYOUT-MANAGER

```
48     // 4. GroupBox ist linkes Kind der
49     // Shell
50     Group groupButtons =
51         new Group(shell,
52             SWT.NO_RADIO_GROUP |
53             SWT.SHADOWETCHEDIN);
54
55     // 5. GroupBox bekommt vertikales
56     // Layout
57     groupButtons.setLayout(layoutGroup);
58
59     // 6. GroupBox wird beschriftet
60     groupButtons.setText("Links");
```

Als Nächstes wird die *Group* erzeugt. Sie ist das erste erzeugte Kindwidget der *Shell*. Damit wird sie in deren horizontal orientiertem Layout ganz links eingeordnet.

```
61     // 7. Label ist rechtes Kind der Shell
62     textLabel = new Label(shell, SWT.CENTER);
63     textLabel.setFont(new Font(display, "Arial",
64         14, SWT.BOLD));
65     textLabel.setText("Rechts");
66     // Textlabel wird rot
67     textLabel.setBackground(
68         new Color(display, 255, 0, 0));
```

Das *Label* ist das zweite Kind-Widget der *Shell*. Es wird damit automatisch rechts neben der *Group* platziert.

```
69      // 8. Buttons werden in der
70      // Group angeordnet:
71      pushButton1 = new Button(
72          groupButtons, SWT.PUSH);
73      pushButton1.setFont(
74          new Font(display, "Arial",
75                  14, SWT.BOLD));
76      pushButton1.setText("Button_1");
77
78      pushButton2 = new Button(
79          groupButtons, SWT.PUSH);
80      pushButton2.setFont(
81          new Font(display, "Arial",
82                  14, SWT.BOLD));
83      pushButton2.setText("Button_2");
84
85      pushButton3 = new Button(
86          groupButtons, SWT.PUSH);
87      pushButton3.setFont(
88          new Font(display, "Arial",
89                  14, SWT.BOLD));
90      pushButton3.setText("Button_3");
91      shell.pack();
92 } // end constructor Beispiel7()
93
94 // Ausgelagerte Event-Loop
95 public void open() { // ... wie immer
96
97 } // end method open()
98 } // end class Beispiel7
```

Die drei Button-Objekte sind Kindwiedtets der Group. Diese wurde vorher mit einem vertikal orientierten Layout-Manager verknüpft. Damit werden sie automatisch in Reihenfolge ihrer Erzeugung vertikal untereinander innerhalb der Group platziert. Die EventLoop hat sich nicht geändert, ebenso wenig wie die *main()*-Klasse. Sie werden nicht mehr gezeigt.

KAPITEL 3. LAYOUT-MANAGER

Kapitel 4

Grundlegende Widgets

Wir erinnern uns: Sämtliche SWT-Widgets sind direkt oder indirekt von der abstrakten Klasse *Widget* abgeleitet. Sie **vererbt** bereits einige sehr wichtige Methoden an ihre abgeleiteten Klassen. Diese Methoden haben wir teilweise schon benutzt und erläutern sie nun systematisch. Wir beginnen mit der Klassenhierarchie der wichtigsten Widgets:

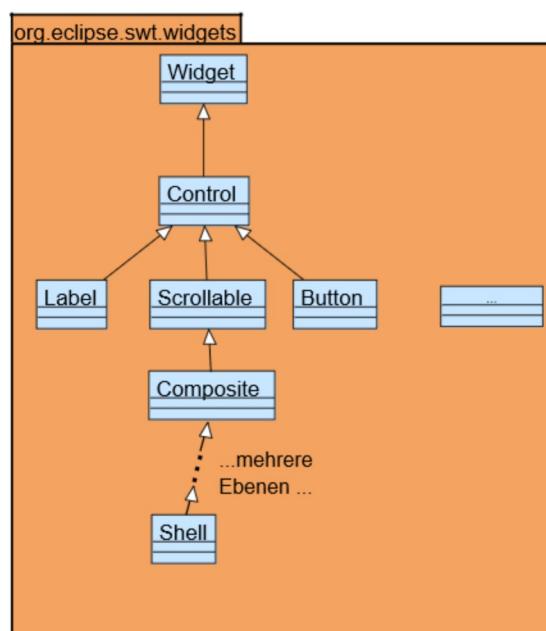


Abbildung 4.1: Einordnung von *Widget* und *Control* in die Klassen-Hierarchie von SWT

KAPITEL 4. GRUNDLEGENDE WIDGETS

Wir sehen, dass viele kleinere Widgets wie z.B. *Label* direkt von der Klasse *Control* abgeleitet sind. Umgangssprachlich werden diese kleineren Widgets auch häufig als *Control* bezeichnet.

4.1 Wichtigste Methoden der Klassen *Widget* und *Control*

Die folgende Tabelle zeigt die wichtigsten Methoden der Superklasse *Widget*:

Methoden der Klasse <i>Widget</i>	
Methode	Erläuterung
<code>void dispose()</code>	„Löscht“ das Widget. Hierbei werden alle Ressourcen, die das Widget und seine Kind-Widgets belegen, für den Garbage-Collector freigegeben.
<code>Display getDisplay()</code>	Gibt die Referenz auf das für dieses Widget verantwortliche <i>Display</i> -Objekt zurück.
<code>void addDisposedListener(DisposeListener listener)</code>	Fügt dem Widget einen Listener hinzu, der ausgeführt wird, <i>bevor</i> der Löschvorgang gestartet wird. Wird beispielsweise angesteuert, wenn ein Widget über die Buttons der Titelleiste geschlossen wird.

Tabelle 4.1: Methoden der Klasse *Widget*

Die folgende Tabelle zeigt die wichtigsten Methoden der Kindklasse *Control*:

Methoden der Klasse <i>Control</i>	
Methode	Erläuterung
<code>void pack()</code>	„Packt“ die Grenzen des Widgets direkt um seine Kind-Widgets.
<code>void setBounds(int x, int y, int width, int height)</code>	<ul style="list-style-type: none">• legt x/y-Offset der linken oberen Ecke des Controls in seinem Parent fest.• legt Breite und Höhe des Controls fest.

Fortsetzung auf nächster Seite

Methode	Erläuterung
<code>void setBackground(Color c)</code>	<p>lege die Hintergrundfarbe des Controls fest. Ein <i>Color</i>-Objekt wird dabei erzeugt mit:</p> <pre>Color c = new Color(display, rot, gruen, blau)</pre> <p>display ist dabei die für das Control verantwortliche <i>Display</i>-Referenz. rot, gruen, blau sind <i>int</i>-Werte zwischen 0 und 255, die die RGB-Werte der gewünschten Farbe festlegen.</p>
<code>void setFont(Font f)</code>	<p>legt die Schriftart für die Beschriftung des Controls fest. Ein <i>Font</i>-Objekt wird dabei festgelegt mit:</p> <pre>Font f = new Font(display, fontname, fontgroesse, fontstil)</pre> <ul style="list-style-type: none"> • display ist dabei die für das Control verantwortliche <i>Display</i>-Referenz. • fontname ist ein <i>String</i> mit dem Namen der Schriftart – z. B. „Arial“. • fontgroesse ist ein <i>int</i>-Wert mit der gewünschten Schriftgröße. • fontstil ist ein <i>int</i>-Wert mit einer SWT-Stilkonstanten für Fontstile – z.B. <i>SWT.BOLD</i>.

Tabelle 4.2: Methoden der Klasse *Control*

Wir werden in den nächsten Kapiteln einige Widgets aus dieser Hierarchie genauer betrachten.

4.2 Buttons

Das SWT kennt für die verschiedenen Arten von Buttons nur die Klasse *Button*. Die Entscheidung, welche Art von Button erzeugt wird, wird über eine Style-Konstante beim Konstruktorauftruf getroffen. Folgende Arten von Buttons können spezifiziert werden:

Style-Konstanten der Klasse <i>Button</i>	
Konstante	Erläuterung
SWT.PUSH	Erzeugt einen PushButton .
SWT.RADIO	Erzeugt einen RadioButton (Einfachauswahl).
SWT.CHECK	Erzeugt eine CheckBox (Mehrfachauswahl)
SWT.TOGGLE	Erzeugt einen ToggleButton (PushButton, der nach Selektieren gedrückt bleibt. Beim nächsten Selektieren wird er wieder auf „ungedrückt“ gesetzt).
SWT.ARROW	Erzeugt einen PushButton mit Pfeil (Arrow)

Tabelle 4.3: Methoden der Klasse *Widget*

Die wichtigsten Methoden der Klasse *Button*:

Wichtigste Methoden der Klasse <i>Button</i>	
Methode	Erläuterung
void addSelectionListener(SelectionListener lst)	verknüpft den Button mit einem <i>SelectionListener</i> . Dies ist ein Listener, welcher auf Anklicken reagiert.
boolean getSelection()	Gibt <i>true</i> zurück, wenn der Button (z. B. ein <i>RadioButton</i>) selektiert ist.
void setText(String text)	Setzt den Titeltext des Buttons
void setImage(Image img)	Bemalt den Button mit einem Icon

Tabelle 4.4: Methoden der Klasse *Widget*

Unser nächstes Beispiel ist eine kleine Testoberfläche, welche

- eine *Group* mit *RadioButtons*,
- eine *Group* mit *CheckBoxen*, sowie
- einen *PushButton*

auswertet:

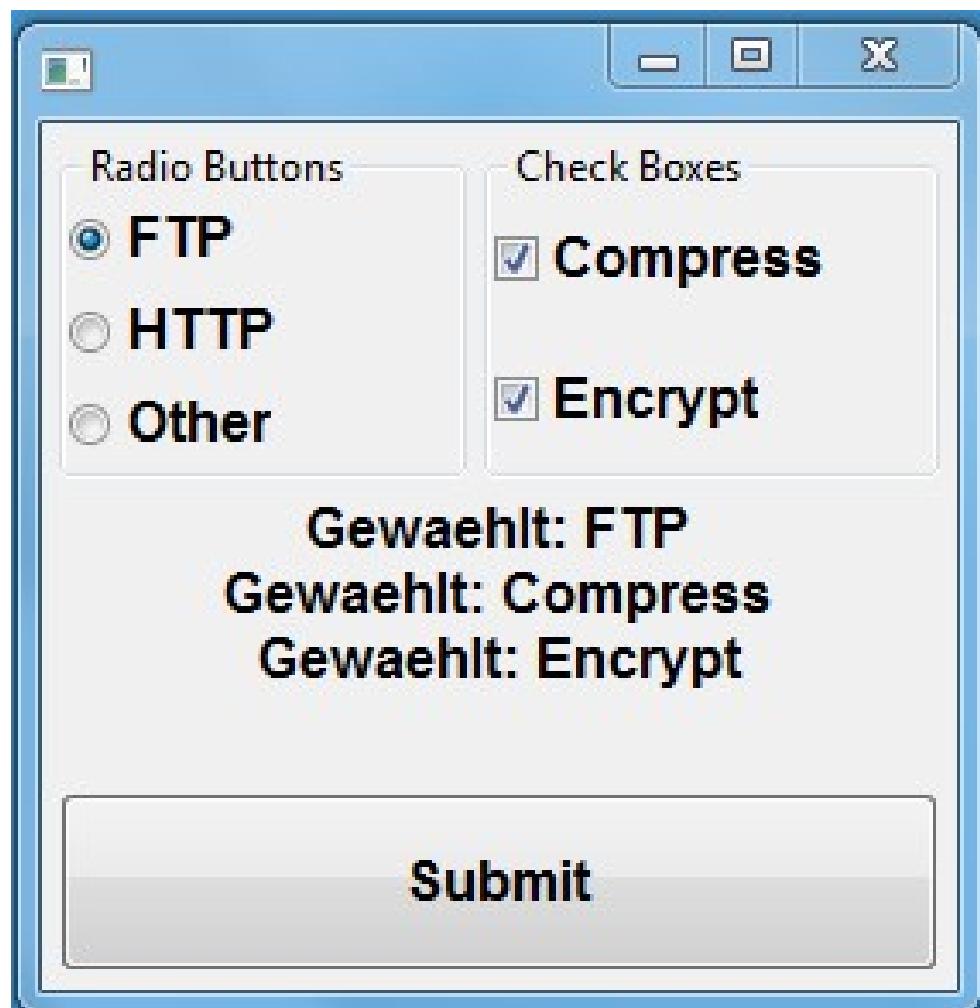


Abbildung 4.2: Beispiel mit *RadioButtons*, *CheckBoxen* und einem *PushButton*

Sobald der *PushButton* gedrückt wird, wertet sein Listener aus, welche *RadioButtons* und *CheckBoxen* selektiert waren. Das Ergebnis wird im *Label* über dem *PushButton* dargestellt.

KAPITEL 4. GRUNDLEGENDE WIDGETS

Die Klassenstruktur sieht folgendermaßen aus:

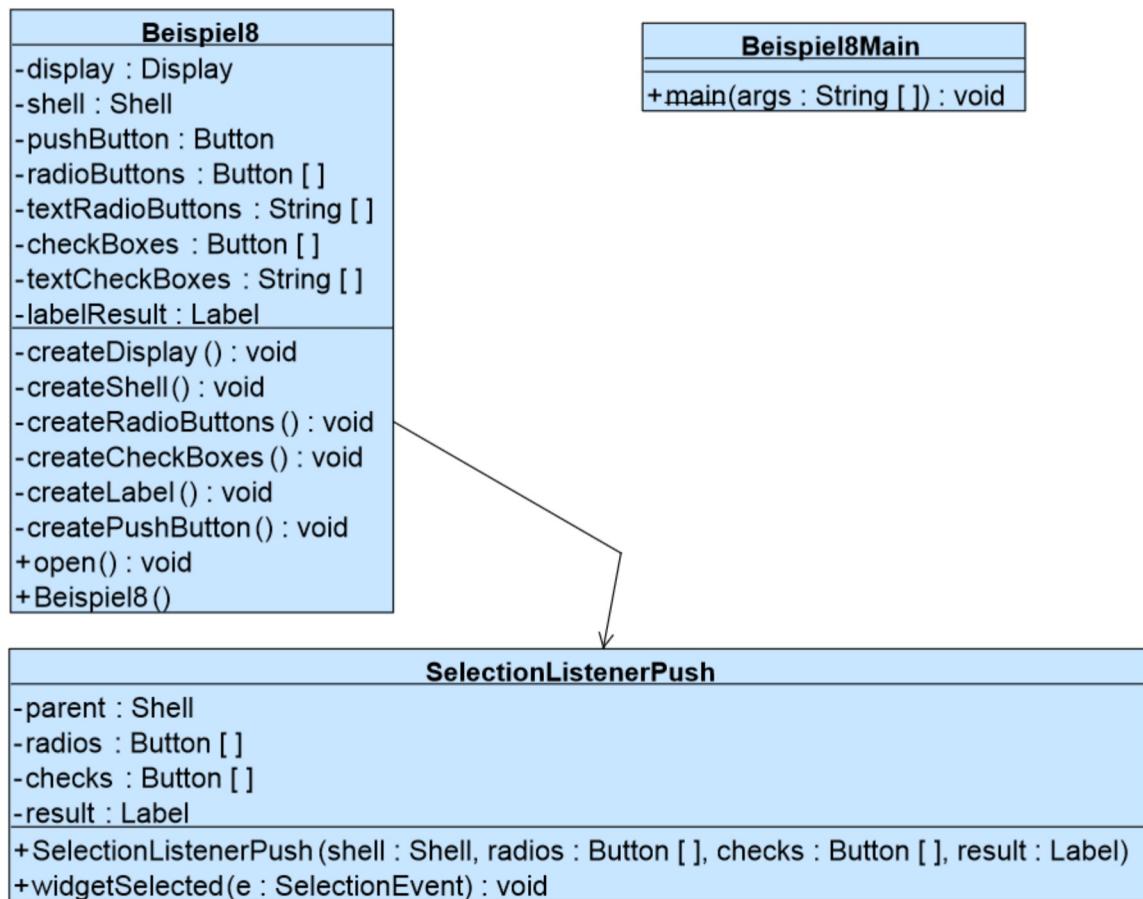


Abbildung 4.3: Klassenstruktur von *Beispiel8*

Wir sehen, dass die Haupt-Widgetklasse einige private `create...()`-Methoden enthält. Sie sind dazu da, die einzelnen Oberflächenbestandteile zu erzeugen und ins Layout einzuordnen. Auf diese Weise wird der Konstruktor übersichtlicher. Weiterhin sind *RadioButtons* und *CheckBoxen* als Arrays aus *Buttons* konzipiert. So wird die Erzeugung der einzelnen Elemente ebenfalls etwas übersichtlicher.

Wir diskutieren die Quelltexte:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.layout.FillLayout;
4 import org.eclipse.swt.layout.GridData;
5 import org.eclipse.swt.layout.GridLayout;
6 import org.eclipse.swt.widgets.Button;
7 import org.eclipse.swt.widgets.Display;
8 import org.eclipse.swt.widgets.Group;
9 import org.eclipse.swt.widgets.Label;
10 import org.eclipse.swt.widgets.Shell;
11
12 public class Beispiel8 {
13
14     private Display display;
15     private Shell shell;
16
17     // Verschiedene Buttons: Sie
18     // werden alle ueber die Klasse
19     // Button realisiert.
20     private Button pushButton;
21
22     // Ein Array mit RadioButtons
23     private Button [] radioButtons;
24
25     // Beschriftung der RadioButtons
26     private String [] textRadioButtons =
27         {"FTP", "HTTP", "Other"};
```

Wir sehen hier in den letzten Zeilen, dass die *RadioButtons* als Array vom Typ *Button* vorliegen. Die Beschriftungen der *RadioButtons* liegen hier als Array vom Typ *String* vor.

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
28 // Ein Array mit CheckBoxen
29 private Button [] checkBoxes;
30
31 // Beschriftung der CheckBoxen
32 private String [] textCheckboxes =
33 {"Compress", "Encrypt"};
34
35 // Ein Label, in dem dar-
36 // gestellt wird, welche
37 // Buttons selektiert wurden:
38 private Label labelResult;
```

Die *CheckBoxen* und ihre Beschriftungen sind ebenfalls als Arrays angelegt. Das *Label*, welches die Auswahlergebnisse anzeigen soll, ist in der letzten Zeile zu sehen.

```
39 // 1. Display erzeugen
40 private void createDisplay() {
41     display = new Display();
42 }
```

Die Erzeugung des *Display*-Objektes wurde ausgelagert in eine Methode *createDisplay()*. Sie wird später im Konstruktor aufgerufen.

```
44 // 2. Shell erzeugen und
45 // mit GridLayout ver-
46 // knuepfen
47 private void createShell(){
48     boolean sameWidth = false;
49     int noOfCols = 2;
50     // Shell erzeugen
51     shell = new Shell(display);
52
53     // 2-spaltiges GridLayout
54     // erzeugen
55     GridLayout gridShell =
56         new GridLayout(noOfCols, sameWidth);
57
58     // Shell mit dem GridLayout
59     // verknuepfen
60     shell.setLayout(gridShell);
61 } // end method createShell()
```

Die Erzeugung der *Shell* ist nun ausgelagert in die Klasse *createShell()*. Sie legt auch sofort ein 2-spaltiges *GridLayout* für die *Shell* an.

KAPITEL 4. GRUNDLEGENDE WIDGETS

Nun folgen die *create...()*-Methoden für die Kind-Widgets:

```
62 // 3. Drei RadioButtons innerhalb
63 // einer Group erzeugen
64 private void createRadioButtons() {
65
66     // Group dehnt sich ueber
67     // 1 Spalte aus
68     int horizontalSpan=1;
69
70     // Group dehnt sich ueber
71     // 1 Zeile aus
72     int verticalSpan=1;
73
74     // Waechst die Group mit ihrer
75     // Grid-Zelle mit?
76     boolean grabExcessHorizontalSpace= true;
77     boolean grabExcessVerticalSpace=true;
78
79     // Eine Group fuer die Radiobuttons
80     Group radioGroup =
81         new Group( shell,
82                 SWT.SHADOW_ETCHED_IN);
83     radioGroup.setText("RadioButtons");
84
85     // Ein vertikales Layout fuer
86     // die Group
87     FillLayout layoutRadio =
88         new FillLayout(SWT.VERTICAL);
89     radioGroup.setLayout(layoutRadio);
```

Der erste Teil der Methode *createRadioButtons()* definiert eine *Group* und ein *FillLayout* für die drei Buttons. Die *Group* wird Parent für die Buttons.

```
90 // Array fuer RadioButtons erzeugen
91 radioButtons =
92     new Button[textRadioButtons.length];
93
94 // RadioButtons selbst erzeugen
95 for(int i=0; i<radioButtons.length; i++) {
96     radioButtons[i] =
97         new Button(
98             radioGroup, SWT.RADIO);
99     radioButtons[i].setFont(
100         new Font(display,
101             "Arial",12,SWT.BOLD));
102     radioButtons[i].setText(
103         textRadioButtons[i]);
104 } // end for
105
106 // Ausrichtungsdaten
107 // fuer die Group innerhalb
108 // des GridLayout der Shell
109 GridData gdata =
110     new GridData(SWT.FILL,
111                 SWT.FILL, grabExcessHorizontalSpace,
112                 grabExcessVerticalSpace,
113                 horizontalSpan, verticalSpan );
114
115 // Group bekommt ihre Ausrichtungs-
116 // Daten
117 radioGroup.setLayoutData(gdata);
118
119 } // end method createRadioButtons()
```

Der zweite Teil der Methode `createRadioButtons()` erzeugt das Array für die *RadioButtons*. Da die Beschriftungen ebenfalls in einem festen Array eingebaut sind, wird das *Button*-Array einfach genau so lang wie das Beschriftungsarray. Die Schleife darunter erzeugt und beschriftet dann die einzelnen Buttons.

KAPITEL 4. GRUNDLEGENDE WIDGETS

Sehr ähnlich funktioniert die Methode *createCheckboxes()*:

```
120 // 4. Drei CheckBoxes innerhalb
121 // einer Group erzeugen
122 private void createCheckboxes() {
123     // Group dehnt sich ueber
124     // 1 Spalte aus
125     int horizontalSpan=1;
126
127     // Group dehnt sich ueber
128     // 1 Zeile aus
129     int verticalSpan=1;
130
131     // Waechst die Group mit ihrer
132     // Grid-Zelle mit?
133     boolean grabExcessHorizontalSpace= true;
134     boolean grabExcessVerticalSpace=true;
135
136     // Eine Group fuer die CheckBoxen
137     Group checkGroup =
138         new Group( shell,
139                     SWT.SHADOW_ETCHED_IN );
140     checkGroup.setText("Check\u25a1Boxes");
141
142     // Ein vertikales Layout fuer
143     // die Group
144     FillLayout layoutCheck =
145         new FillLayout(SWT.VERTICAL);
146     checkGroup.setLayout(layoutCheck);
147
148     // Array fuer RadioButtons erzeugen
149     checkBoxes =
150         new Button[textCheckboxes.length];
151
152     // RadioButtons selbst erzeugen
153     for(int i=0; i<checkBoxes.length; i++) {
154         checkBoxes[i] =
155             new Button(
156                 checkGroup, SWT.CHECK);
157         checkBoxes[i].setFont(
158             new Font(display,
159                     "Arial",12,SWT.BOLD));
160         checkBoxes[i].setText(
```

```

161         textCheckBoxes[i]);
162     } // end for
163
164     // Ausrichtungsdaten
165     // fuer die Group innerhalb
166     // des GridLayout der Shell
167     GridData gdata =
168         new GridData(SWT.FILL,
169                     SWT.FILL, grabExcessHorizontalSpace,
170                     grabExcessVerticalSpace,
171                     horizontalSpan, verticalSpan );
172
173     // Group bekommt ihre Ausrichtungs-
174     // Daten
175     checkGroup.setLayoutData(gdata);
176
177 } // end method createCheckboxes()

```

Das Label für die Ergebnisdarstellung belegt 2 Spalten des *GridLayout* der *Shell*. Es hat die *Shell* als direktes Parent:

```

178     // 5. Ein Label als Kind der
179     // Shell erzeugen
180     private void createLabel() {
181         // Label dehnt sich ueber
182         // 2 Spalten aus
183         int horizontalSpan=2;
184
185         // Label dehnt sich ueber
186         // 1 Zeile aus
187         int verticalSpan=1;
188
189         // Waechst das Label mit seiner
190         // Grid-Zelle mit?
191         boolean grabExcessHorizontalSpace= true;
192         boolean grabExcessVerticalSpace=true;
193
194         // Label als Kind der Shell erzeugen
195         labelResult =
196             new Label(shell,SWT.CENTER);
197
198         // Font des Labels setzen
199         labelResult.setFont(
200             new Font(display, "Arial",

```

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
201         12, SWT.BOLD));  
202  
203     // Label enthaelt noch  
204     // keinen Text  
205     labelResult.setText("");  
206  
207     // Ausrichtungsdaten  
208     // fuer das Label innerhalb  
209     // des GridLayout der Shell  
210     GridData gdata =  
211         new GridData(SWT.FILL,  
212             SWT.FILL, grabExcessHorizontalSpace,  
213             grabExcessVerticalSpace,  
214             horizontalSpan, verticalSpan);  
215  
216     // Label bekommt seine Ausrichtungs-  
217     // Daten  
218     labelResult.setLayoutData(gdata);  
219  
220 } // end method createLabel()
```

Der untere Teil der Methode erzeugt die Layout-Daten für das *Label* und übergibt sie ihm.

Die Erzeugung des *PushButton* wird mit der Methode *createPushButton()* erledigt.

```
221 // 5. Einen PushButton als Kind der
222 // Shell erzeugen
223 private void createPushButton(){
224     // PushButton dehnt sich ueber
225     // 2 Spalten aus
226     int horizontalSpan=2;
227
228     // PushButton dehnt sich ueber
229     // 1 Zeile aus
230     int verticalSpan=1;
231
232     // Waechst der Button mit seiner
233     // Grid-Zelle mit?
234     boolean grabExcessHorizontalSpace= true;
235     boolean grabExcessVerticalSpace=true;
236
237     // Button als Kind der Shell erzeugen
238     pushButton =
239         new Button(shell, SWT.PUSH);
240
241     // Font des Buttons setzen
242     pushButton.setFont(
243         new Font(display, "Arial",
244             12, SWT.BOLD));
245
246     // Button beschriften
247     pushButton.setText("Submit");
```

Auch der *Button* ist ein direktes Kind der *Shell*. Er erstreckt sich – wie das *Label* aus den beiden vorherigen Seiten – über 2 Spalten des *GridLayouts* der Shell.

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
249     // Ausrichtungsdaten
250     // fuer den Button innerhalb
251     // des GridLayout der Shell
252     GridData gdata =
253         new GridData(SWT.FILL,
254             SWT.FILL, grabExcessHorizontalSpace,
255             grabExcessVerticalSpace,
256             horizontalSpan, verticalSpan );
257
258     // Button bekommt seine Ausrichtungs-
259     // Daten
260     pushButton.setLayoutData(gdata);
261
262     // Button bekommt einen
263     // Listener, der RadioButtons
264     // und CheckBoxen auswertet
265     pushButton.addSelectionListener(
266         new SelectionListenerPush(
267             shell,
268             radioButtons,
269             checkBoxes,
270             labelResult));
271
272 } // end method createPushButton()
```

Der zweite Teil der Methode gibt dem *PushButton* einen Listener mit. Hierbei handelt es sich wieder um einen nicht-anonymen Listener, den wir noch auf den folgenden Seiten zeigen werden. Er bekommt die Referenzen aller Widgets über den Konstruktor mit, die er für seine Auswertung benötigt.

Der Konstruktor ruft nun nur noch die *create...()*-Methoden auf.

```
273 // 6. Der Konstruktor ruft
274 // nur noch die create...()-Methoden auf
275 public Beispiel8(){
276
277     createDisplay();
278     createShell();
279     createRadioButtons();
280     createCheckBoxes();
281     createLabel();
282     createPushButton();
283     shell.pack();
284
285 } // end constructor Beispiel8()
286
287 // 7. Ausgelagerte Event-Loop
288 public void open() { // wie immer ...
289
290 } // end method open()
291 } // end class Beispiel8
```

Die EventLoop ist gleich geblieben wie in den vorherigen Beispielen.

KAPITEL 4. GRUNDLEGENDE WIDGETS

Der Konstruktor des Listener bekommt die Referenzen auf RadioButtons, CheckBoxen, Shell und Label.

```
1 import org.eclipse.swt.events.SelectionAdapter;
2 import org.eclipse.swt.events.SelectionEvent;
3 import org.eclipse.swt.widgets.Button;
4 import org.eclipse.swt.widgets.Label;
5 import org.eclipse.swt.widgets.Shell;
6
7 public class SelectionListenerPush
8     extends SelectionAdapter {
9     private Shell parent;
10    private Button [] radios;
11    private Button [] checks;
12    private Label result;
13
14    public SelectionListenerPush(
15        Shell parent,
16        Button[] radios,
17        Button[] checks,
18        Label result) {
19        this.parent = parent;
20        this.radios = radios;
21        this.checks = checks;
22        this.result = result;
23    } // end constructor
24
25    public void widgetSelected(
26        SelectionEvent e) {
27        String resultString = "";
```

```
28     // Alle Radio-Buttons durchsehen
29     for(int i=0; i<radios.length; i++) {
30
31         // Wenn RadioButton selektiert ist,
32         // dann seinen Titeltext im
33         // Result vermerken.
34         if(radios[i].getSelection()) {
35             resultString += "Gewaehlt: " +
36                 radios[i].getText() + "\n";
37         } // end if
38     } // end for
39
40     // Alle Check-Boxen durchsehen
41     for(int i=0; i<checks.length; i++) {
42
43         // Wenn RadioButton selektiert ist,
44         // dann seinen Titeltext im
45         // Result vermerken.
46         if(checks[i].getSelection()) {
47             resultString += "Gewaehlt: " +
48                 checks[i].getText() + "\n";
49         } // end if
50     } // end for
51     // Label mit Selektionsergebnis belegen
52     result.setText(resultString);
53     // Shell passend um das nun mehrzeilige
54     // Label packen
55     parent.pack();
56 } // end method widgetSelected()
57 } // end class
```

Die Reaktionsmethode wertet zunächst die *RadioButtons* und *CheckBoxen* aus. Jeden „Treffer“ vermerkt sie in einem Ergebnisstring. Dieser wird am Ende der Methode ins *Label* eingesetzt. Die *main()*-Klasse sieht gleich aus wie vorher – sie wird nicht mehr gezeigt.

Merke

- Buttons werden anhand der Style-Konstante unterschieden, die dem Konstruktor der Klasse *Button* bei der Erzeugung mitgegeben wird.
- Für PushButtons gilt:
 - Sie werden über die Style-Konstante *SWT.PUSH* erzeugt.
 - Ihr am häufigsten benutzter Listener ist bei Buttons der *SelectionListener*. Für ihn gibt es bereits die vorgefertigte Adapterklasse *SelectionAdapter*.
- Für Check-Boxen und Radio-Buttons gilt:
 - werden über die Style-Konstanten *SWT.CHECK* bzw. *SWT.RADIO* erzeugt.
 - Ihr Zustand (selektiert / nicht selektiert) wird über ihre Methode *getSelection()* abgefragt.

4.3 Eingabe-Widgets

In diesem Kapitel werden einige häufig benutzte Eingabe-Widgets diskutiert:

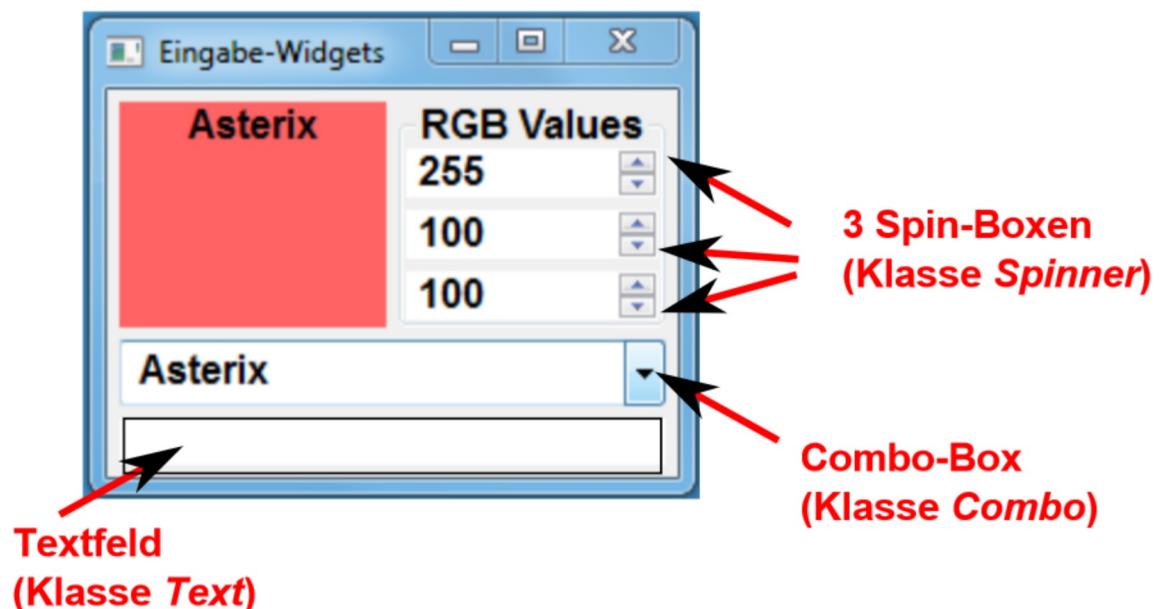


Abbildung 4.4: Beispiel10 mit Textfeld, Combo-Box und 3 Spin-Boxen

- Textfeld (SWT-Klasse *Text*): Ein ggf. scrollbares Textfeld für einfache Texteingaben.
- Combo-Box (SWT-Klasse *Combo*): Eine Klappliste, mit der einzelne Textelemente ausgewählt werden können.
- Spin-Box (SWT-Klasse *Spinner*): Ein Textfeld für die Eingabe von Zahlenwerten. Es kann mit Pfeiltasten in- bzw. dekrementiert werden.

KAPITEL 4. GRUNDLEGENDE WIDGETS

Bei allen drei Eingabewidgets sind die wichtigsten Methoden:

Wichtigste Methoden der Klassen <i>Text</i> , <i>Combo</i> und <i>Spinner</i>	
Methode	Erläuterung
void addModifyListener(ModifyListener listener)	verknüpft das Feld mit einem <i>ModifyListener</i> . Dies ist ein Listener, welcher auf Veränderung des Textinhaltes reagiert.
String getText()	Gibt den Textinhalt des Eingabewidgets als <i>String</i> zurück
void setText(String text)	Setzt den Textinhalt des Eingabewidgets

Tabelle 4.5: Wichtigste Methoden der Klassen *Text*, *Combo* und *Spinner*

Unsere Beispielanwendung für Eingabewidgets aus Abbildung 4.4 arbeitet folgendermaßen:

- Links ist ein nicht-editierbares *Label*. In ihm sind unsere Testausgaben zu sehen.
- Rechts ist eine *Group* mit 3 Spin-Boxen (SWT-Klasse *Spinner*). Mit ihnen können wir die RGB-Werte des links stehenden *Labels* einstellen. Jeder der *Spinner* ist mit einem nicht-anonymen *ModifyListener* verknüpft. Dieser modifiziert die Farbwerte des *Labels* entsprechend den 3 Zahlenwerten in den *Spinners*.
- Unten folgt eine Combo-Box (SWT-Klasse *Combo*). Mit ihr können wir einige feste Strings selektieren. Auch die *Combo* ist mit einem *ModifyListener* verknüpft. Dieser modifiziert den Text des *Labels* auf den gerade in der *Combo* selektierten String.
- Ganz unten folgt ein Textfeld(SWT-Klasse *Text*). In dieses können wir frei wählbare Texte eintippen. Auch das *Text*-Feld ist mit einem *ModifyListener* verknüpft. Dieser modifiziert den Text des *Labels* auf den gerade eingetippten Text.

Die Klassenstruktur von *Beispiel10* sieht folgendermaßen aus:

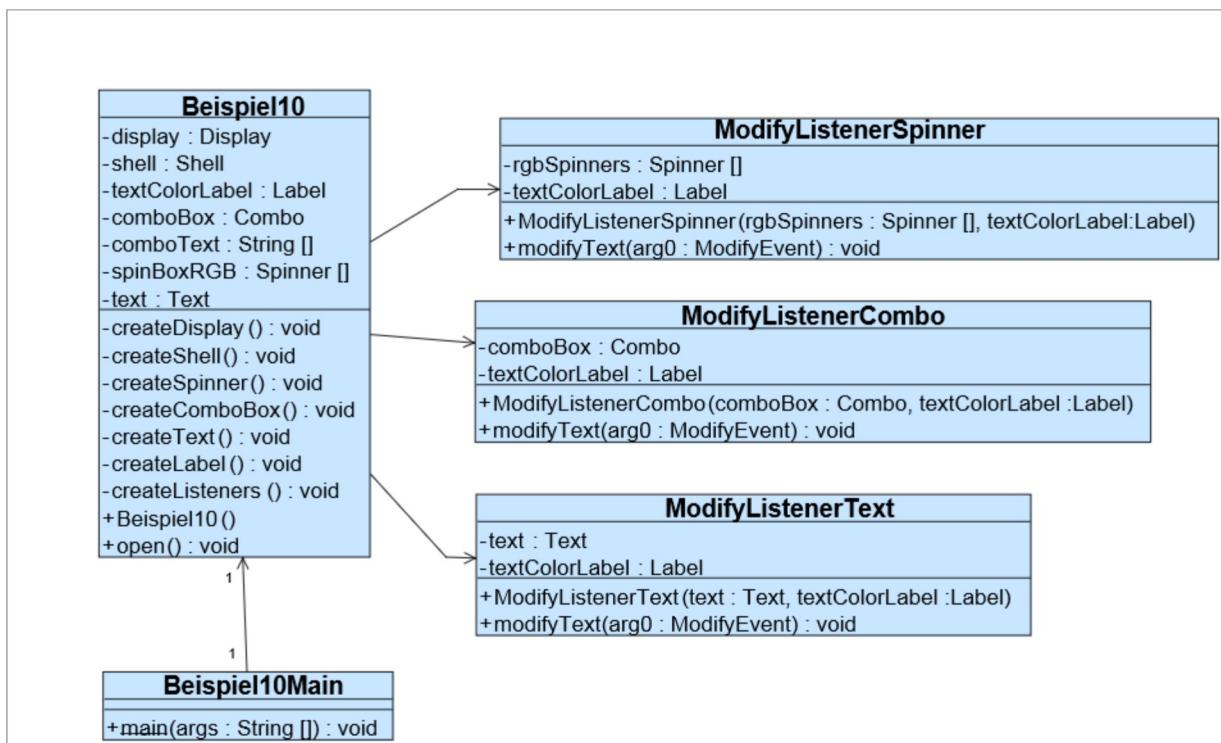


Abbildung 4.5: Klassenstruktur *Beispiel10*

Wir sehen drei verschiedene *ModifyListener*-Klassen – für jedes Eingabewidget in der Klasse *Beispiel10* eine. Für drei *Spinner* ist ein Array aus Referenzen vorgesehen. Sie tun alle das Gleiche – daher arbeiten sie mit der gleichen Listener-Klasse *ModifyListenerSpinner*. Allerdings bekommt jeder *Spinner* ein eigenes Objekt dieser Klasse.

KAPITEL 4. GRUNDLEGENDE WIDGETS

Wir diskutieren die Quelltexte:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.graphics.Font;
3 import org.eclipse.swt.layout.FillLayout;
4 import org.eclipse.swt.layout.GridData;
5 import org.eclipse.swt.layout.GridLayout;
6 import org.eclipse.swt.widgets.Combo;
7 import org.eclipse.swt.widgets.Display;
8 import org.eclipse.swt.widgets.Group;
9 import org.eclipse.swt.widgets.Label;
10 import org.eclipse.swt.widgets.Shell;
11 import org.eclipse.swt.widgets.Spinner;
12 import org.eclipse.swt.widgets.Text;
13
14 public class Beispiel10 {
15
16     private Display display;
17     private Shell shell;
18
19
20     // Ein Label zur Text-
21     // und Farbanzeige
22     private Label textColorLabel;
23
24     // Eine Combo-Box fuer
25     // Test-Text im Label
26     private Combo comboBox;
27
28     // Test-Strings fuer
29     // die Combo-Box
30     private String [] comboBoxText =
31         {"Asterix", "Obelix", "Majestix",
32          "Miraculix", "Gutemine"};
```

Label, *Text-Feld* und *Combo* sind wie gewohnt als einfache, private Objektvariablen angelegt.

```
34 // Eine Spin-Box fuer rgb-
35 // Farbeinstellung im
36 // Label
37 private Spinner [] spinBoxRGB;
38
39 // Ein Textfeld fuer
40 // Test-Text im Label
41 private Text text;
```

Die Spin-Boxen werden als Array vereinbart. Das Textfeld ist eine einfache Referenz.

```
42 // 1. Display erzeugen
43 private void createDisplay() {
44     display = new Display();
45 }
```

Die *create...()*-Methode für die *Display*-Referenz ist unverändert.

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
46 // 2. Shell erzeugen und
47 // mit GridLayout ver-
48 // knuepfen
49 private void createShell(){
50     boolean sameWidth = true;
51     int noOfCols = 2;
52     // Shell erzeugen
53     shell = new Shell(display);
54     shell.setText("Eingabe-Widgets");
55
56     // 2-spaltiges GridLayout
57     // erzeugen
58     GridLayout gridShell =
59         new GridLayout(noOfCols, sameWidth);
60
61     // 5 Pixel zwischen den Layout-
62     // Elementen
63     gridShell.horizontalSpacing = 5;
64     gridShell.verticalSpacing = 5;
65
66     // Shell mit dem GridLayout
67     // verknuepfen
68     shell.setLayout(gridShell);
69 } // end method createShell()
```

Die `create...()`-Methode für die `Shell`-Referenz erzeugt wieder eine `Shell` mit einem zweispaltigen `GridLayout`.

```
70 //3. Spin-Box innerhalb
71 // des GridLayout der Shell
72 // erzeugen
73 private void createSpinner() {
74
75     // Array fuer Spin-Boxen erzeugen
76     spinBoxRGB = new Spinner[3];
77
78     // Group-Box fuer Spin-Boxen
79     // erzeugen
80     Group gSpin = new Group(shell,
81         SWT.NO_RADIO_GROUP |
82         SWT.SHADOWETCHEDIN);
83
84     // Der Group einen Font
85     // mitgeben
86     gSpin.setFont(
87         new Font(display, "Arial",
88             12, SWT.BOLD));
89
90     // Group-Box beschriften
91     gSpin.setText("RGB Values");
92
93     // FillLayout fuer die Group
94     FillLayout layoutRGB =
95         new FillLayout(SWT.VERTICAL);
96
97     // 5 Pixel zwischen den
98     // Layout-Elementen
99     layoutRGB.spacing = 5;
100
101    // Group mit FillLayout verknuepfen
102    gSpin.setLayout(layoutRGB);
```

Die *create...()*-Methode für die *Spinner*-Referenzen erzeugt zunächst ein 3-elementiges Array für die Referenzen. Da die *Spinner* in einer *Group* liegen, wird zunächst eine solche erzeugt. Danach wird ein vertikal orientiertes *FillLayout* erzeugt und mit der *Group* verknüpft.

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
103     for(int i=0; i<spinBoxRGB.length; i++){  
104  
105         // Spin-Box erzeugen  
106         spinBoxRGB[i] =  
107             new Spinner(gSpin, SWT.NONE);  
108  
109         // Dem Spinner einen Font  
110         // mitgeben  
111         spinBoxRGB[i].setFont(  
112             new Font(display, "Arial",  
113                 12, SWT.BOLD));  
114  
115         // Dem Spinner einen Werte-  
116         // Bereich mitgeben  
117         spinBoxRGB[i].setMinimum(0);  
118         spinBoxRGB[i].setMaximum(255);  
119  
120         // Default-Wert fuer den Spinner  
121         spinBoxRGB[i].setSelection(100);  
122  
123     } // end for  
124  
125 } // end method createSpinner()
```

In einer Schleife werden die drei *Spinner*-Objekte erzeugt. Sie bekommen einen Wertebereich für die einzugebenden Farbwerte (0 bis 255) zugewiesen.

```
126 // 4. ComboBox erzeugen und
127 // befuellen
128 private void createComboBox() {
129     // Combo dehnt sich ueber
130     // 2 Spalten aus
131     int horizontalSpan=2;
132
133     // Combo dehnt sich ueber
134     // 1 Zeile aus
135     int verticalSpan=1;
136
137     // Waechst die Combo mit ihrer
138     // Grid-Zelle mit?
139     boolean grabExcessHorizontalSpace= true;
140     boolean grabExcessVerticalSpace=true;
141
142     // GridData-Referenz
143     GridData gdata;
144
145     // ComboBox wird Kind der Shell
146     // Style ist wie ein
147     // Pulldown-Menue
148     comboBox = new Combo(
149         shell, SWT.DROP_DOWN);
150
151     // Der ComboBox einen Font
152     // mitgeben
153     comboBox.setFont(
154         new Font(display, "Arial",
155             12, SWT.BOLD));
```

Die *create...()*-Methode für die *Combo* legt zunächst deren Layout-Einstellungen fest. Die *Combo* erstreckt sich über 2 Spalten des *GridLayout* der *Shell*. Danach wird das *Combo*-Objekt erzeugt. Der Style-Parameter *SWT.DROP_DOWN* sagt, dass die Klappliste sich beim Aufklappen wie ein Drop-Down-Menü verhält.

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
156     // Ausrichtungsdaten
157     // fuer die ComboBox innerhalb
158     // des GridLayout der Shell
159     gdata =
160         new GridData(SWT.FILL,
161             SWT.FILL, grabExcessHorizontalSpace,
162             grabExcessVerticalSpace,
163             horizontalSpan, verticalSpan );
164
165     // Der ComboBox die Aus-
166     // richtungsdaten mitgeben
167     comboBox.setLayoutData(gdata);
168
169     // ComboBox mit Text-Liste
170     // bestuecken
171     for(int i=0; i<comboText.length; i++){
172
173         comboBox.add(comboText[i]);
174
175     } // end for
176
177 } // end method createComboBox()
```

Im letzten Abschnitt der Methode bekommt die *Combo-Box* ihre Layout-Daten. Danach wird sie in einer Schleifenkonstruktion mit sämtlichen Strings des Arrays *comboText* bestückt.

```
178 //5. Textfeld erzeugen
179 private void createText() {
180     // Textfeld dehnt sich ueber
181     // 2 Spalten aus
182     int horizontalSpan=2;
183
184     // Textfeld dehnt sich ueber
185     // 1 Zeile aus
186     int verticalSpan=1;
187
188     // Waechst das Textfeld mit ihrer
189     // Grid-Zelle mit?
190     boolean grabExcessHorizontalSpace= true;
191     boolean grabExcessVerticalSpace=true;
192
193     // GridData-Referenz
194     GridData gdata;
195
196     // Textfeld wird Kind der Shell
197     // Style ist mehrzeilig mit
198     // automat. Zeilenumbruch
199     text = new Text(
200         shell, SWT.MULTI | SWT.WRAP);
201
202     // Dem Textfeld einen Font
203     // mitgeben
204     text.setFont(
205         new Font(display, "Arial",
206             12, SWT.BOLD));
```

Die `create...()`-Methode für das Textfeld legt zunächst dessen Layout-Einstellungen fest. Das Feld erstreckt sich über 2 Spalten des `GridLayout` der `Shell`. Danach wird das Textfeld erzeugt. Die bitweise veroderten Style-Parameter `SWT.MULTI` / `SWT.WRAP` sagen aus, dass das Textfeld mehrzeilfähig sein soll. Es soll ggf. den eingetippten Text umbrechen (Zeilenumbruch).

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
207     // Ausrichtungsdaten
208     // fuer das Textfeld innerhalb
209     // des GridLayout der Shell
210     gdata =
211         new GridData(SWT.FILL,
212             SWT.FILL, grabExcessHorizontalSpace,
213             grabExcessVerticalSpace,
214             horizontalSpan, verticalSpan );
215
216     // Dem Textfeld die Aus-
217     // richtungsdaten mitgeben
218     text.setLayoutData(gdata);
219
220 } // end method createText()
```

Im letzten Abschnitt der Methode bekommt das Textfeld seine Layout-Daten.

Die *create...()*-Methode für das *Label* funktioniert fast gleich wie diejenige für das Textfeld.

```
221     // 6. Label innerhalb
222     // des GridLayout der Shell
223     // erzeugen
224     private void createLabel() {
225
226         // Group dehnt sich ueber
227         // 1 Spalte aus
228         int horizontalSpan=1;
229
230         // Group dehnt sich ueber
231         // 1 Zeile aus
232         int verticalSpan=1;
233
234         // Waechst die Group mit ihrer
235         // Grid-Zelle mit?
236         boolean grabExcessHorizontalSpace= true;
237         boolean grabExcessVerticalSpace=true;
238
239         // GridData-Referenz
240         GridData gdata;
241
242         // Label wird Kind der Shell
243         // Schrift-Style ist zentriert
244         textColorLabel = new Label(shell,
245             SWT.CENTER);
```

```

246     // Dem Label einen Font
247     // mitgeben
248     textColorLabel.setFont(
249         new Font(display, "Arial",
250             12, SWT.BOLD));
251
252     // Ausrichtungsdaten
253     // fuer Label innerhalb
254     // des GridLayout der Shell
255     gdata =
256         new GridData(SWT.FILL,
257             SWT.FILL, grabExcessHorizontalSpace,
258             grabExcessVerticalSpace,
259             horizontalSpan, verticalSpan );
260
261     // Dem Label die Aus-
262     // richtungsdaten mitgeben
263     textColorLabel.setLayoutData(gdata);
264
265 } // end method createLabel()

```

Diesmal haben wir eine eigene Methode für die Erzeugung und Verknüpfung der Listener-Objekte erstellt. Grund: Bevor die Listener erzeugt werden, müssen **alle** Widget-Objekte, deren Referenzen wir durch den Konstruktor hineinreichen, **fertig erzeugt** sein. Diese Methode wird im Konstruktor zuletzt aufgerufen!

```

266     // 7. Listener - Verknuepfungen
267     // erstellen
268     private void createListeners() {
269         // Den Spin-Boxen einen Modify-Listener
270         // mitgeben
271         for(int i=0; i < spinBoxRGB.length; i++) {
272             spinBoxRGB[i].addModifyListener(
273                 new ModifyListenerSpinner(
274                     spinBoxRGB, textColorLabel));
275         }
276
277         // Der ComboBox einen
278         // Listener mitgeben
279         comboBox.addModifyListener(
280             new ModifyListenerCombo(
281                 comboBox, textColorLabel));
282
283         // Dem Textfeld einen

```

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
284     // Listener mitgeben
285     text.addModifyListener(
286         new ModifyListenerText(
287             text, textColorLabel));
288 } // end createListeners()
289
290 // 8. Konstruktor ruft nur noch
291 // die create...()-Methoden auf
292 public Beispiel10(){
293
294     createDisplay();
295     createShell();
296     createLabel();
297
298     createSpinner();
299     createComboBox();
300     createText();
301     createListeners();
302
303     shell.pack();
304
305 } // end constructor Beispiel10()
306
307 // 9. Ausgelagerte Event-Loop
308
309     } // end while
310 } // end method open()
```

Der Konstruktor ruft nur noch die *create...()*-Methoden auf.

Achtung

In der Reihenfolge, in der die *create...()* - Methoden aufgerufen werden, werden die erzeugten Widgets in das *GridLayout* der *Shell* eingeordnet!

Wir diskutieren die Listener-Klassen. Zunächst der Listener für das Textfeld:

```

1 import org.eclipse.swt.events.ModifyEvent;
2 import org.eclipse.swt.events.ModifyListener;
3 import org.eclipse.swt.widgets.Label;
4
5 import org.eclipse.swt.widgets.Text;
6
7 public class ModifyListenerText
8     implements ModifyListener {
9
10    Text text;
11    Label textColorLabel;
12
13    // Listener benoetigt Textfeld zum
14    // auswerten
15    // Listener benoetigt Label zum
16    // Beschriften
17    public ModifyListenerText(
18        Text text, Label textColorLabel) {
19
20        this.text = text;
21        this.textColorLabel = textColorLabel;
22    } // end constructor

```

Da der Listener das Textfeld auswerten muss und in das *Label* hineinschreiben muss, bekommt er über den Konstruktor beide Referenzen übergeben.

```

23 @Override
24 public void modifyText(
25     ModifyEvent arg0) {
26
27     // Text aus dem Textfeld
28     // abfragen ...
29     String txtText = text.getText();
30
31     // ... und ins Label einsetzen
32     textColorLabel.setText(txtText);
33
34 } // end method modifyText()
35
36 } // end class ModifyListenerText

```

Die Reaktionsmethode fragt den aktuellen Text aus dem Textfeld ab und setzt ihn ins Label ein. Da diese Methode bei **jeder** Zeicheneingabe ausgelöst wird, erscheint das

KAPITEL 4. GRUNDLEGENDE WIDGETS

Einsetzen ins *Label* wie ein „Mitschreiben“ mit dem Benutzer, der die Zeichen ins Textfeld eingibt.

```
1 import org.eclipse.swt.events.ModifyEvent;
2 import org.eclipse.swt.events.ModifyListener;
3 import org.eclipse.swt.widgets.Combo;
4 import org.eclipse.swt.widgets.Label;
5
6 public class ModifyListenerCombo
7     implements ModifyListener {
8
9     Combo comboBox;
10    Label textColorLabel;
11
12    // Listener benoetigt ComboBox zum
13    // auswerten
14    // Listener benoetigt Label zum
15    // Beschriften
16    public ModifyListenerCombo(Combo comboBox,
17        Label textColorLabel) {
18
19        this.comboBox = comboBox;
20        this.textColorLabel = textColorLabel;
21    }
```

Der Listener für die *Combo* bekommt – ähnlich wie der vorherige Listener – die Widget-Referenzen über den Konstruktor mit, die er für seine Auswertung benötigt

```
23 @Override
24 public void modifyText(ModifyEvent arg0) {
25
26     String comboBoxText;
27
28     // Selektierten Text aus
29     // ComboBox abfragen ...
30     comboBoxText = comboBox.getText();
31
32     // .. und ins Label einsetzen
33     textColorLabel.setText(comboBoxText);
34
35 } // end method modifyText()
36
37 } // end class ModifyListenerCombo
```

Die Reaktionsmethode fragt den selektierten String aus der *Combo* ab und setzt ihn ins

Label ein.

```

1 import org.eclipse.swt.events.ModifyEvent;
2 import org.eclipse.swt.events.ModifyListener;
3 import org.eclipse.swt.graphics.Color;
4 import org.eclipse.swt.widgets.Display;
5 import org.eclipse.swt.widgets.Label;
6 import org.eclipse.swt.widgets.Spinner;
7
8 public class ModifyListenerSpinner
9     implements ModifyListener {
10
11    Spinner [] rgbSpinners;
12    Label textColorLabel;
13
14    //Listener benoetigt Spinner zum
15    // auswerten
16    // Listener benoetigt Label zum
17    // Beschriften
18    public ModifyListenerSpinner(
19        Spinner[] rgbSpinners,
20        Label textColorLabel) {
21
22        this.rgbSpinners = rgbSpinners;
23        this.textColorLabel = textColorLabel;
24    }

```

Der Listener für die *Spinner* bekommt das Array mit allen 3 *Spinner*-Referenzen übergeben, da er alle 3 RGB-Werte auswerten muss. Weiterhin bekommt er das *Label* übergeben, dessen Farbe geändert werden soll.

```

25    @Override
26    public void modifyText(
27        ModifyEvent arg0) {
28
29        // neu zu setzende
30        // RGB-Werte fuer das Label
31        int red, green, blue;
32        // Wir holen das "alte" Color-
33        // Objekt fuer die Hintergrund-
34        // farbe des Labels ...
35        Display display =
36            textColorLabel.getDisplay();
37        red = Integer.parseInt(
38            rgbSpinners[0].getText());

```

KAPITEL 4. GRUNDLEGENDE WIDGETS

```
39     green = Integer.parseInt(
40         rgbSpinners[1].getText());
41     blue = Integer.parseInt(
42         rgbSpinners[2].getText());
43     Color color =
44         textColorLabel.getBackground();
45
46     // ... und geben es frei.
47     color.dispose();
```

Die Reaktionsmethode fragt zunächst aus allen 3 *Spinners* die Werte ab. Da diese als Strings vorliegen, müssen sie noch in int-Werte konvertiert werden, damit sie als Farbwerte verarbeitet werden können. Bevor das *Label* ein neues *Color*-Objekt für die Hintergrund-Farbe bekommen kann, muss sein altes zerstört werden. Dies liegt daran, dass **Ressourcen** wie Farb- und Font-Objekte nicht automatisch vom Garbage-Collector zerstört werden, sondern erst dann, wenn sie freigegeben werden.

```
48     // Dann erzeugen wir es neu
49     // mit den Farbwerten, die in
50     // den Spinners stehen ...
51     color = new Color(display,
52         red, green, blue);
53
54     // ... und verknuepfen es mit
55     // dem Label
56     textColorLabel.setBackground(color);
57
58 } // end method modifyText()
59
60 } // end class ModifyListenerSpinner
```

Erst danach verknüpfen wir das *Label* mit einem neuen *Color*-Objekt für die Hintergrundfarbe.

Die *main()*-Klasse hat sich strukturell nicht verändert – sie wird nicht mehr gezeigt.

Merke

Für Eingabewidgets wie *Text*, *Combo* oder *Spinner* ist der *ModifyListener* der am häufigsten genutzte Listener. Er wird aktiv, wenn sich der Inhalt des Eingabewidgets geändert hat.

Kapitel 5

Menügesteuerte Oberflächen

Menüstrukturen werden im SWT aus folgenden Bestandteilen erzeugt:

- Fenster, in denen sich Menüs befinden (z. B. *Shell* für Pulldown-Menüs oder *Text*) für Popup-Menüs.
- Klasse *Menu* für die Erzeugung von Menübalken und Menüs. Der Menübalken wird hierbei als **horizontal orientiertes** Menü interpretiert.
- Klasse *MenuItem* für die Erzeugung von Menüoptionen. Die Menütitel auf einem Menübalken werden hierbei ebenfalls als *MenuItem* erzeugt.

Die verschiedenen Arten von Menüs (Klasse *Menu*) lassen sich anhand ihrer Style-Konstanten unterscheiden:

Style-Konstanten der Klasse <i>Menu</i>	
Konstante	Erläuterung
SWT.BAR	Es wird ein horizontal ausgeprägtes Menü – also ein Menübalken – erzeugt.
SWT.DROP_DOWN	Es wird ein Pulldown- bzw. Dropdown-Menü erzeugt.
SWT.POP_UP	Es wird ein Kontext- bzw. Popup-Menü erzeugt.
SWT.NO_RADIO_GROUP	Es sind keine Radio-Buttons im Menü enthalten.
SWT.LEFT_TO_RIGHT	Wird i.d.R. in Kombination mit <i>SWT.BAR</i> verwendet. Die Pulldown-Menüs in einem solchen Menübalken werden dann (wie unter Windows üblich) linksbündig, von links nach rechts angeordnet.

Fortsetzung auf nächster Seite

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

Konstante	Erläuterung
SWT.RIGHT_TO_LEFT	Wird oft in Kombination mit <i>SWT.BAR</i> verwendet. Die Pulldown-Menüs in einem solchen Menübalken werden dann von rechts nach links angeordnet.

Tabelle 5.1: Style-Konstanten der Klasse *Menu*

Die verschiedenen Arten von Menüoptionen (Klasse *MenuItem*) lassen sich ebenfalls anhand ihrer Style-Konstanten unterscheiden:

Style-Konstanten der Klasse <i>MenuItem</i>	
Konstante	Erläuterung
SWT.CHECK	Die Menüoption enthält links eine Check-Box mit Häkchen, um Oberflächeneigenschaften an- oder auszuschalten.
SWT.CASCADE	Die Menüoption öffnet ein weiteres (Unter-)Menü. Alle Menüoptionen im Menübalken sind von diesem Typ.
SWT.PUSH	Eine „normale“ Menüoption. Sie verhält sich ähnlich wie ein PushButton.
SWT.RADIO	Eine Menüoption für eine 1-aus-N-Auswahl innerhalb des Menüs. Verhält sich ähnlich wie ein RadioButton in einer <i>Group</i> .
SWT.SEPERATOR	Ein waagrechter Strich, mit dem visuell einzelne Gruppen von Menüoptionen voneinander getrennt werden können.

Tabelle 5.2: Style-Konstanten der Klasse *MenuItem*

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

Die nächste Anwendung zeigt einen sehr kleinen Editor mit zwei Pulldown-Menüs und einem Kontext-Menü:

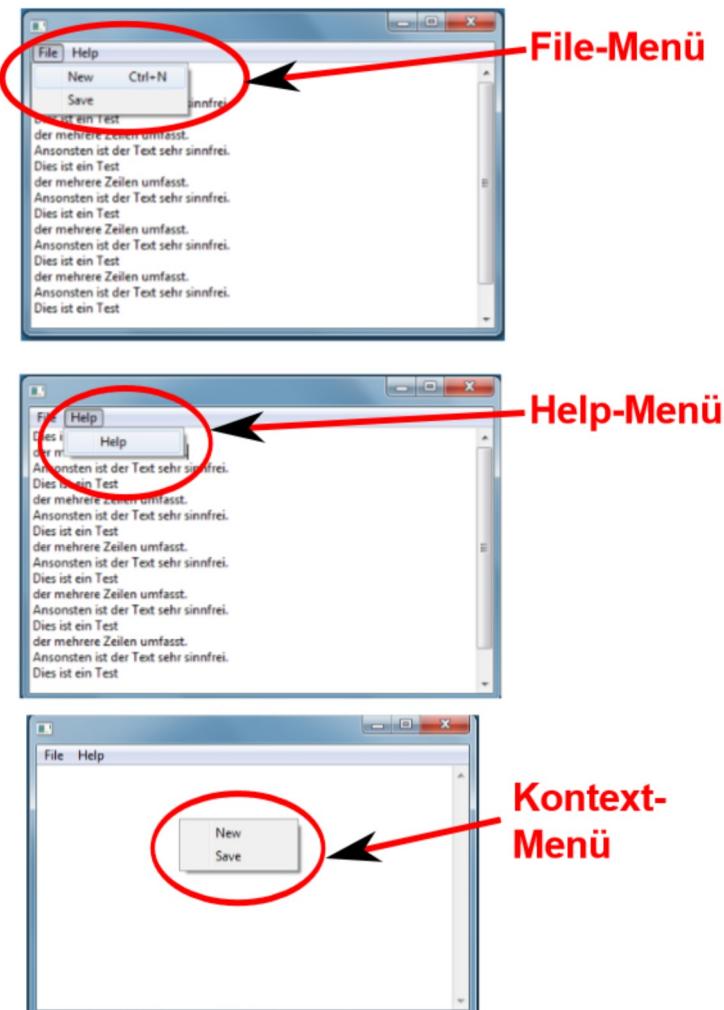


Abbildung 5.1: Beispiel11 mit einer menügesteuerten Oberfläche

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

Die Klassenstruktur sieht folgendermaßen aus:

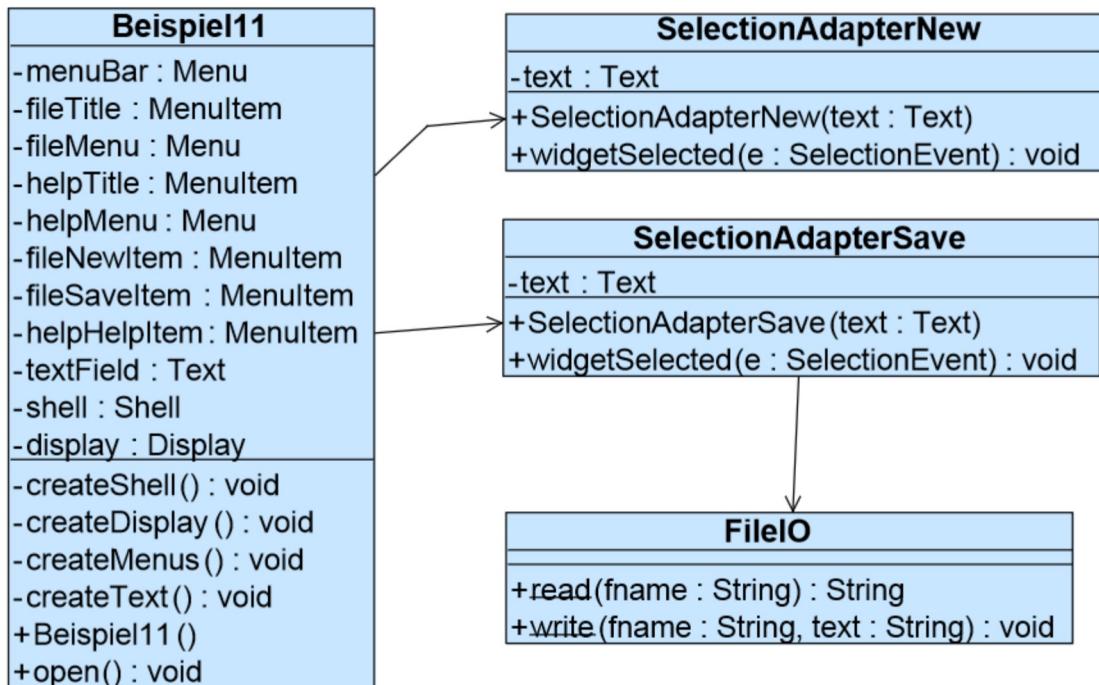


Abbildung 5.2: Klassenstruktur von Beispiel 11

Hier haben wir wieder eine vollständige 3-Schicht-Architektur:

- Präsentationsschicht: Sie wird durch Klasse *Beispiel11* implementiert. In ihr sind alle Widgets (Menüs, Textfeld) untergebracht.
- Dialogmanagementschicht: Sie wird durch die beiden Listener implementiert.
- Anwendungsschicht: Sie wird durch die Dateizugriffsklasse *FileIO* implementiert.

Wir diskutieren die Quelltexte. Am interessantesten ist hier die Präsentationsschicht, denn sie setzt die gesamte Menüstruktur auf:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.layout.FillLayout;
3 import org.eclipse.swt.widgets.Display;
4 import org.eclipse.swt.widgets.Menu;
5 import org.eclipse.swt.widgets.MenuItem;
6 import org.eclipse.swt.widgets.Shell;
7 import org.eclipse.swt.widgets.Text;
8
9
10 public class Beispiel11 {
11
12     //Menue-Balken
13     private Menu menuBar;
14
15     //Titel des File-Menus
16     private MenuItem fileTitle;
17
18     //File-Menu
19     private Menu fileMenu;
20
21     // Titel des Help-Menus
22     private MenuItem helpTitle;
23
24     // Help-Menu
25     private Menu helpMenu;
26
27     // Menue-Optionen der beiden
28     // Pulldown-Menues
29     private MenuItem fileNewItem;
30     private MenuItem fileSaveItem;
31     private MenuItem helpHelpItem;
```

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

Im ersten Teil von Klasse *Beispiel11* sehen wir die Vereinbarungen von

- Menübalken,
- 2 Pulldown-Menüs und deren Menütiteln, sowie
- deren Menüoptionen.

Es folgt die Vereinbarung des Kontextmenüs:

```
32 // Kontext-Menu im Textfeld
33 // (bekommt keinen Menu-Titel)
34 private Menu contextMenue;
35
36 // Menu-Items fuer Kontext-Menu
37 private MenuItem contextFileNewItem;
38 private MenuItem contextFileSaveItem;
39
40 //Textfeld zum Editieren
41 private Text textField;
42
43 private Shell shell;
44 private Display display;
```

Dieses werden wir mit dem danach vereinbarten Textfeld verknüpfen. Außerdem haben wir hier noch *Shell*- und *Display*-Objekt.

```
45 // 1. Shell erzeugen
46 private void createShell(){
47
48     shell = new Shell(display);
49
50     // Vertikales FillLayout
51     shell.setLayout(
52         new FillLayout(SWT.VERTICAL));
53 } // end method createShell()

54
55 // 2. Display erzeugen
56 private void createDisplay()
57 {
58     display = new Display();
59 } // end method createDisplay()
```

In diesem Abschnitt werden *Display* und *Shell* erzeugt. Die *Shell* bekommt ein *FillLayout* mit vertikaler Orientierung. In dieses werden später Menübalken und Textfeld automatisch bei ihrer Erzeugung platziert.

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

Es folgt die Methode *createMenus()* zur Erstellung des Menübalkens mit seinen Pulldown-Menüs:

```
60 // 3. Menues einrichten
61 private void createMenus() {
62     // 3.1. MenuBar einrichten: Klasse Menu,
63     // Typ SWT.BAR
64     menuBar = new Menu(shell, SWT.BAR);
65
66     // 3.2. MenuBar in Toplevel-Shell
67     // einhaengen
68     shell.setMenuBar(menuBar);
69
70     // 3.3.
71     // 3.3.1. File-Menue einrichten:
72     // erst Menue-Titel
73     // Parent-Widget ist menuBar;
74     // Titeltext ist "File"
75     fileTitle = new MenuItem(
76         menuBar, SWT.CASCADE);
77     fileTitle.setText("&File");
78
79     // 3.3.2 Nun das eigentliche Menue
80     // einrichten: Typ SWT.DROP_DOWN
81     // (entspricht Pulldown-Menue); Parent-
82     // Widget ist die Toplevel-Shell;
83     fileMenu = new Menu(shell, SWT.DROP_DOWN);
84
85     // 3.3.3 Verknuepfen des Pulldown-Menues
86     // mit dem Menue-Titel
87     fileTitle.setMenu(fileMenu);
```

Hier geschieht folgendes:

- Erstellung des Menübalkens als horizontal orientiertes *Menu*-Objekt mit Style-Konstante *SWT.BAR*
- Die Menütitel sind *MenuItem*-Objekte mit Style-Konstante *SWT.CASCADE*. Dies bedeutet, dass diese „Optionen“ ein weiteres Menü aufklappen können.
- Die beiden Pulldown-Menüs werden als *Menu*-Objekte mit Style-Konstante *SWT.DROP_DOWN* angelegt.

```
88 // 3.3.4 Help-Menue analog zu 3.3.1
89 // bis 3.3.3
90 helpTitle = new MenuItem(
91     menuBar, SWT.CASCADE);
92 helpTitle.setText("&Help");
93
94 helpMenu = new Menu(
95     shell, SWT.DROP_DOWN);
96
97 helpTitle.setMenu(helpMenu);
```

Beim Help-Menü sehen wir noch einmal die gleichen Mechanismen wie auf den beiden vorhergehenden Seiten.

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

In diesem Abschnitt sehen wir nun, wie die Menüoptionen angelegt werden:

```
98     // 3.4. Meneeinträge einrichten:  
99     // 3.4.1 File --> New: Mene-Eintrag  
100    // ist PushButton-Eintrag  
101    // (ohne Untermenue);  
102    // Parent ist das Mene "fileMene";  
103    // Titeltext ist "New"  
104    fileNewItem = new MenuItem(  
105        fileMenu, SWT.PUSH);  
106  
107    // &New heisst: N ist unterstrichen als  
108    // Mnemonic  
109    // \tCtrl+N heisst: Nach einem Tab  
110    // steht der Name des Shortcuts  
111    fileNewItem.setText("&New\tCtrl+N");  
112  
113    // Hier setzen wir den Shortcut: Er be-  
114    // rechnet sich aus Style-Konstante  
115    // fuer Control-Key und Unicode fuer 'N'  
116    fileNewItem.setAccelerator(SWT.CTRL + 'N');  
117    System.out.println("SWT.CTRL:" + SWT.CTRL);  
118    System.out.println("Unicode N:" + (int)'N');  
119    fileNewItem.addSelectionListener(
```

Im Titeltext der Menüoption geben wir gleich einen Shortcut mit an. Er muss jedoch noch separat mit dem `setAccelerator(...)`-Aufruf an das `MenuItem` übergeben werden. Die Option **File → New** wird sofort mit einem nicht-anonymen Listener verknüpft. Ihn zeigen wir später.

```

120
121     // 3.4.2 File --> Save: analog zu 3.4.1
122     fileSaveItem = new MenuItem(
123         fileMenu, SWT.PUSH);
124     fileSaveItem.setText("&Save");
125     fileSaveItem.addSelectionListener(
126         new SelectionAdapterSave(textField));
127
128     // 3.4.3 Help --> Help: analog zu 3.4.1
129     helpHelpItem = new MenuItem(
130         helpMenu, SWT.PUSH);
131     helpHelpItem.setText("&Help");

```

In diesem letzten Abschnitt werden die Optionen **File → Save** und **Help → Help** aufgebaut. Damit ist die Methode *createMenus()* zuende.

Es folgt die Methode *createText()* zum Aufbau des Editierfeldes samt Kontextmenü:

```

133
134     //4. Textfeld mit Kontextmenue
135     // erzeugen
136     private void createText(){
137
138         // Textfeld kann mehrzeilig sein.
139         // Automatischer Zeilenumbruch
140         // wird unterstuetzt; Wir wollen einen
141         // vertikalen Scrollbar
142         textField = new Text(shell, SWT.MULTI |
143                             SWT.WRAP |
144                             SWT.V_SCROLL) ;
145
146         // Kontext-Menu fuer Textfeld erstellen
147         contextMenue = new Menu(
148             shell, SWT.POP_UP);
149
150         // Kontext-Menu mit Textfeld verknuepfen
151         textField.setMenu(contextMenue);
152
153         // MenuItem erstellen, beschriften
154         contextFileNewItem = new MenuItem(
155             contextMenue, SWT.PUSH);
156         contextFileNewItem.setText("&New");
157
158         // MenuItem mit Listener bestuecken
159         contextFileNewItem.addSelectionListener(

```

KAPITEL 5. MENÜGESTEUERTE OBERFLÄCHEN

```
160         new SelectionAdapterNew(textField));
161
162     // MenuItem erstellen, beschriften
163     contextFileSaveItem = new MenuItem(
164         contextMenu, SWT.PUSH);
165     contextFileSaveItem.setText("&Save");
166
167     // MenuItem mit Listener bestuecken
168     contextFileSaveItem.addSelectionListener(
169         new SelectionAdapterSave(textField));
170
171 }
172 } // end method open()
```

Das Kontextmenü wird mit Style-Konstante *SWT.POP_UP* erzeugt. Es wird dem Textfeld mit dessen *setMenu()*-Methode mitgegeben. Die Bestückung des Menüs mit Menüoptionen erfolgt analog zu den oben erläuterten Pulldown-Menüs. Hier sehen wir noch den Konstruktor. Die EventLoop hat sich nicht verändert.

Die beiden Listener-Klassen für die Menüoptionen **File** → **New** und **File** → **Save** bekommen beide über den Konstruktor die Referenz auf das vorher erzeugte Textfeld mit. Sie müssen beide auf dessen Inhalt zugreifen können. Wir zeigen beispielhaft den Listener für **File** → **New**:

```
1 import org.eclipse.swt.events.SelectionAdapter;
2 import org.eclipse.swt.events.SelectionEvent;
3 import org.eclipse.swt.widgets.Text;
4
5 public class SelectionAdapterNew
6         extends SelectionAdapter {
7
8     private Text text;
9
10    // Die aufrufende Klasse reicht hier
11    // die Referenz auf ihr Textfeld
12    // hinein.
13    public SelectionAdapterNew(Text text) {
14        this.text = text;
15    } // end constructor
16
17    // Reaktionsmethode leert
18    // das Textfeld
19    public void widgetSelected(
20        SelectionEvent e){
21
22        text.setText("");
23
24    } // end method widgetSelected()
25 } // end class SelectionAdapterNew
```

Seine Reaktionsmethode leert das Textfeld, indem er es mit einem Leerstring belegt. Der zweite Listener *SelectionAdapterSave* wird hier nicht gezeigt. Die Klasse *FileIO* hat sich gegenüber *Beispiel3* nicht mehr geändert. Sie wird hier nicht noch einmal gezeigt. Ebenso verzichten wir hier auf das Zeigen der Klasse *Beispiel11Main*, da sich ihre Struktur ebenfalls nicht geändert hat. Alle hier nicht aufgeführten Klassen sind jedoch im Komplettbeispiel auf der E-Learning-Plattform vorhanden.

Merke

- Menüs werden im SWT aus Objekten der Klassen *Menu* und *MenuItem* konstruiert.
- Ein Menübalken ist im SWT ein horizontal orientiertes *Menu*-Objekt mit Style-Konstante *SWT.BAR*. Er verhält sich wie ein horizontal ausgerichteter Layout-Manager
- Ein Menütitel ist im SWT ein *MenuItem* mit Style-Konstante *SWT.CASCADE*. Es wird auf dem Menübalken standardmäßig horizontal von links positioniert.
- Ein Pulldown-Menü ist im SWT ein *Menu*-Objekt mit Style-Konstante *SWT.DROP_DOWN*. Es verhält sich wie ein vertikal ausgerichteter Layout-Manager.
- Ein Popup-Menü ist im SWT ein *Menu*-Objekt mit Style-Konstante *SWT.POP_UP*.
- Eine Menüoption ist im SWT ein *MenuItem* mit einer der Style-Konstante *SWT.PUSH*, *SWT.CHECK*, *SWT.RADIO* oder *SWT.CASCADE*.

Kapitel 6

Dialogfenster

Bisher haben wir noch nicht verwendet: von SWT vorgefertigte oder vom Entwickler selbst entwickelte Dialogfenster, welche nur kurzzeitig eingeblendet werden und nach Erledigung ihrer Aufgabe wieder ausgeblendet werden. Bekannte Beispiele sind hier:

- Dialogfenster für Dateiauswahl
- Dialogfenster für Druckerausgabe
- Dialogfenster für einfache Nachrichten an den Nutzer

Sie werden für temporäre Aufgaben verwendet. Die in ihnen steckenden Funktionalitäten werden aber innerhalb der GUI nicht permanent benötigt.

6.1 Dialogfenster unter SWT

Wir haben bislang nur sehr selten von Vererbungsmechanismen unter SWT Gebrauch gemacht. Dies wird sich bei der Anwendung von Dialogfenstern ändern:

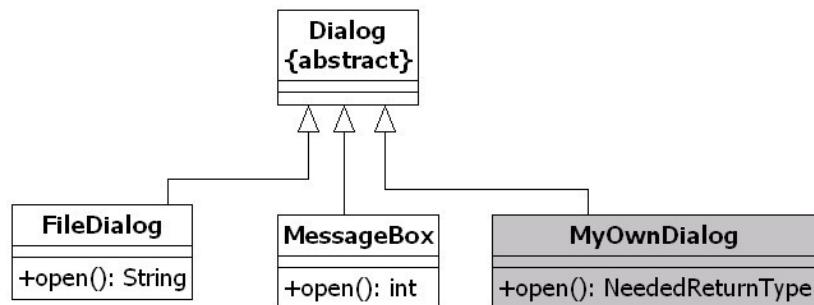


Abbildung 6.1: Klassenhierarchie von Dialogfenstern im SWT

KAPITEL 6. DIALOGFENSTER

Dialogfenster sind im SWT immer von der vorgefertigten, abstrakten Klasse *Dialog* abgeleitet. Sie implementieren immer eine eigene Methode *open()*, die das Bestücken und Aufklappen des Dialogfensters übernimmt.

Achtung

Ein Dialogfenster ist kein Widget! Es ist auch nicht von der Klasse *Widget* abgeleitet. Es enthält aber durchaus Widgets, wie z. B Buttons oder Textfelder.

Für Dialogfenster gibt es unter SWT folgende Modalitätsstufen:

Modalitätsstufen der Klasse <i>Dialog</i>	
Konstante	Erläuterung
SWT.APPLICATION_MODAL	Die gesamte Applikation wird vom Dialogfenster gesperrt.
SWT.SYSTEM_MODAL	Der gesamte Desktop wird vom Dialogfenster gesperrt.
SWT.PIMARY_MODAL	Das unter dem Dialogfenster liegende Parent ist NICHT gesperrt.

Tabelle 6.1: Modalitätsstufen der Klasse *Dialog*

Diese Modalitätsstufen werden vom nativen Windowing-System jedoch häufig nur als „Hint“ – also als Hinweis, der so weit wie möglich befolgt wird, aufgefasst. Die Standard-Auswertung eines SWT-Dialogfensters (gleichgültig ob vorgefertigt oder selbst gebaut) entspricht immer folgendem Muster:

```

1 // Modalitäts-Eigenschaften festlegen
2 int style = SWT.APPLICATION_MODAL;
3
4 // Dialogfenster erzeugen
5 <DialogType> dlg = new <DialogType>(parent, style);
6
7 // Ggf. Daten im Dialogfenster setzen
8 dlg.setSomeData(data);
9
10 // Dialogfenster aufklappen -- open()-Methode liefert
11 // zurück, mit welchem Button wir ausgestiegen sind
12 <ReturnType> retval = dlg.open();
13 if(retval == value1){
14     doSomething1();
15 }
16 else{
17     doSomething2();
18 }
```

Listing 6.1: Dialogfenster-Auswertung unter SWT

Der Return-Typ der `open()`-Methode muss hier nicht unbedingt `int` sein. Ein `FileDialog` liefert beispielsweise den Namen der selektierten Datei als String zurück. Falls keine Datei selektiert wurde (Cancel-Fall) liefert sie eine `null`-Referenz.

Bei der Implementierung eigener Dialogfenster sieht das Vorgehensmuster folgendermaßen aus:

```

1 public class MyDialog extends Dialog {
2
3     // Rückgabewert der open()-Methode; Wird meist
4     // in einem der Listener des Dialogfensters belegt.
5     private Object input;
6
7     // Konstruktor. Er ruft nur den Superklassen-
8     // konstruktor der Klasse Dialog auf.
9     public MyDialog(Shell parent,
10         int style) {
11
12     super(parent, style);
13 }
14
15     // Erzeugt das Layout des Dialogfensters
16     // und gibt das Ergebnis der Eingabe zurück
17     public Object open(){
18 }
```

KAPITEL 6. DIALOGFENSTER

```
19     // Eigene Shell Shell des Dialogfensters
20     Shell shellDialogWindow = new Shell(getParent());
21
22     // Inhalt der Shell wird erzeugt (Buttons, etc)
23     createContents(shellDialogWindow);
24
25     // Shell des Dialogfensters aufklappen
26     shellDialogWindow.open();
27
28     // getParent() liefert die Shell des Hauptfensters
29     // getDisplay() liefert das Display des Hauptfensters
30     Display display = getParent().getDisplay();
31
32     // Dies ist die eigene Event-Loop des Dialogfensters!
33     while(!shellDialogWindow.isDisposed()){
34         if(!display.readAndDispatch()){
35             display.sleep();
36         }
37     } // end while
38
39     // ACHTUNG: An dieser Stelle ist die
40     // Shell des Dialogfensters disposed!
41     return input;
42 } // end method open()
43
44 // private Hilfsmethode für die Erzeugung des Fensterinhalts
45 private void createContents(final Shell shellDlgWindow) {
46     // Kind-Widgets der Shell des Dialogfensters erzeugen
47     // und Layout aufbauen.
48
49     // anonyme Listener der Buttons werden hier implementiert
50 } // end method createContents
51
52 } // end class MyDialog
```

Listing 6.2: Dialogfenster-Aufbau unter SWT

Merke

- Dialogfenster sind im SWT immer direkt oder indirekt von der abstrakten SWT-Klasse *Dialog* abgeleitet.
- Dialogfenster sind im SWT keine Widgets!
- Die Methode *open()* erledigt bei SWT-Dialogfenstern 3 Aufgaben:
 - Layout-Erzeugung und Bestückung der *Shell* des Dialogfensters
 - Start der EventLoop des Dialogfensters
 - Rückgabe der Information des Dialogfensters (z. B. Dateiname)

6.2 Funktionsweise **MessageBox**

Eine MessageBox ist ein kleines Nachrichtenfenster folgender Bauart: Es wird für kurze

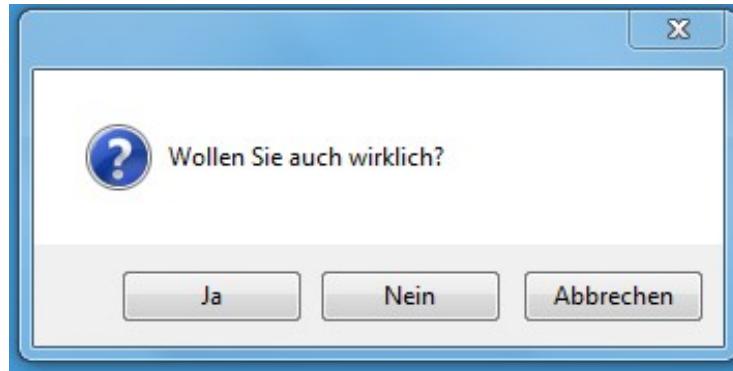


Abbildung 6.2: Screenshot einer MessageBox

Nachfragen oder Benachrichtigungen aufgeblendet. Bei Drücken eines der PushButtons wird es wieder geschlossen.

Das SWT unterscheidet anhand des eingeblendeten Icons folgende Arten von MessageBoxen:

Arten von MessageBoxen im SWT		
Style-Konstante	Icon	Erläuterung
SWT.ICON_ERROR	×	Wird in Fehlersituationen angezeigt.
SWT.ICON_INFORMATION	i	Wird angezeigt, wenn der Nutzer über ein Ereignis informiert werden soll.
SWT.ICON_QUESTION	?	Wird angezeigt, wenn der Nutzer eine Frage beantworten soll.
SWT.ICON_WARNING	!	Wird in Warnungs-Situationen angezeigt, die nicht ganz so kritisch sind wie Fehler-situationen.

Fortsetzung auf nächster Seite

Style-Konstante	Icon	Erläuterung
SWT.ICON_WORKING		Wird angezeigt, wenn das System gerade blockiert, weil es arbeitet. Unter Windows 10 sieht das Icon genau so aus wie ein Information-Icon.

Tabelle 6.2: Icons der Klasse *MessageBox*

Die Icons beziehen sich in dieser Tabelle auf Windows 10. Unter anderen Betriebssystemen / Desktops können sie anders aussehen.

Weiterhin bietet das SWT die Möglichkeit, MessageBoxen bezüglich ihrer PushButtons zu konfigurieren. Auch dies geschieht über Style-Konstanten:

PushButtons der Klasse <i>MessageBox</i>	
Konstante	Erläuterung
SWT.OK	Normaler Ok-Button
SWT.CANCEL	Abbrechen-Button
SWT.YES	Ja-Button
SWT.NO	Nein-Button
SWT.RETRY	Wiederholen-Button
SWT.ABORT	Abbrechen-Button, der in den meisten Window-Managern links in der Message-Box platziert wird. →siehe Tabelle unten.
SWT.IGNORE	Ignorieren-Button

Tabelle 6.3: PushButtons der Klasse *MessageBox*

Von den PushButtons sind folgende Kombinationen möglich:

PushButton-Kombinationen der Klasse <i>MessageBox</i>	
SWT.OK	
SWT.OK SWT.CANCEL	
SWT.YES SWT.NO	
SWT.YES SWT.NO SWT.CANCEL	

Fortsetzung auf nächster Seite

KAPITEL 6. DIALOGFENSTER

Kombination

```
SWT.RETRY | SWT.CANCEL  
SWT.ABORT | SWT.RETRY | SWT.IGNORE
```

Tabelle 6.4: PushButton-Kombinationen der Klasse *MessageBox*

Das folgende Code-Snippet zeigt, wie eine typische *MessageBox*-Anzeige funktioniert:

```
1 // MessageBox erzeugen: Sie hat die Buttons: Yes , No ,  
2 // Cancel; sie hat ein Question -Icon  
3 // Der Message -Text wird umgebrochen, wenn er zu lang ist.  
4 // Die Style -Konstanten werden für Parameter 2 einfach  
5 // bitweise miteinander verodert.  
6 MessageBox msg = new MessageBox(parent , SWT.ICON_QUESTION  
7 | SWT.YES | SWT.NO | SWT.CANCEL | SWT.WRAP);  
8  
9 // Nachrichtentext fuer die MessageBox setzen  
10 msg.setMessage("Wollen Sie auch wirklich?");  
11  
12 // MessageBox aufklappen: Die open()-Methode liefert  
13 // als int-Wert zurueck, mit welchem PushButton sie  
14 // zugeklappt wurde  
15 int retval = msg.open();  
16 switch(retval)  
17 {  
18     case SWT.YES: tuwasYes(); break;  
19  
20     case SWT.NO: tuwasNo(); break;  
21  
22     case SWT.CANCEL: break;  
23 }
```

Listing 6.3: MessageBox-Anzeige und -Auswertung unter SWT

6.3 Funktionsweise *FileDialog*

Ein *FileDialog* ist ein Datei-Auswahl dialogfenster folgender Bauart: Er ermöglicht es,

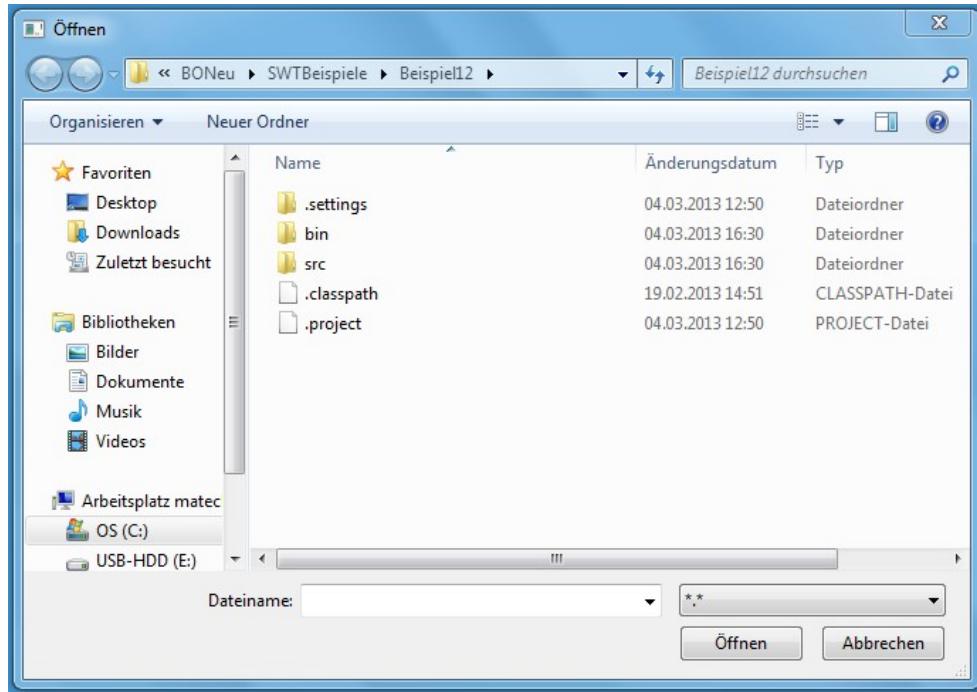


Abbildung 6.3: Screenshot eines *FileDialog*

Dateien anhand ihres Namens auszuwählen. Er liefert jedoch beim Zuklappen lediglich den **Namen der anselektierten Datei** zurück. Der Dateizugriff selbst muss also separat vorgenommen werden – beispielsweise mit einer Dateizugriffsklasse wie *FileIO* aus Beispiel 3. Dies hat den Hintergrund, dass das Dialogfenster nicht wissen kann, in welchem Format die gewünschte Datei vorliegt. Dies kann nur der Entwickler der Benutzeroberfläche wissen. Dementsprechend muss er in einer gesonderten Schicht seine Dateizugriffsklasse(n) entwickeln – i.d.R. ist das für jedes Dateiformat eine eigene Klasse.

KAPITEL 6. DIALOGFENSTER

Das SWT unterscheidet anhand folgender Style-Konstanten verschiedene Formen von Datei-Auswahldialogen mit Hilfe der Klasse *FileDialog*:

Style-Konstanten der Klasse <i>FileDialog</i>	
Konstante	Erläuterung
SWT.OPEN	Auswahldialog zum Öffnen einer Datei
SWT.SAVE	Auswahldialog zum Speichern einer Datei
SWT.MULTI	Auswahldialog zum Öffnen mehrerer Dateien (MDI-Oberfläche)

Tabelle 6.5: Style-Konstanten der Klasse *FileDialog*

Das folgende Code-Snippet zeigt, wie ein *FileDialog* verwendet wird:

```
1 // FileDialog zum Öffnen einer Datei erzeugen
2 FileDialog f = new FileDialog(parent, SWT.OPEN);
3
4 // FileDialog aufklappen: --> seine open()-
5 // Methode liefert den selektierten Dateinamen
6 // als String zurück
7 String fname = f.open();
8
9 // Inhalt aus der selektierten Datei in String
10 // txt einlesen
11 if(fname != null) {
12     String txt = FileIO.read(fname);
13
14     // String txt ins Textfeld einblenden
15     text.setText(txt);
16 }
```

Listing 6.4: FileDialog-Anzeige und -Auswertung unter SWT

6.4 Selbststudium: Größere Beispielimplementierung mit Dialogfenstern

Im Folgenden betrachten wir die Editoranwendung aus dem letzten Beispiel – erweitert um die Anwendung von Dialogfenstern:

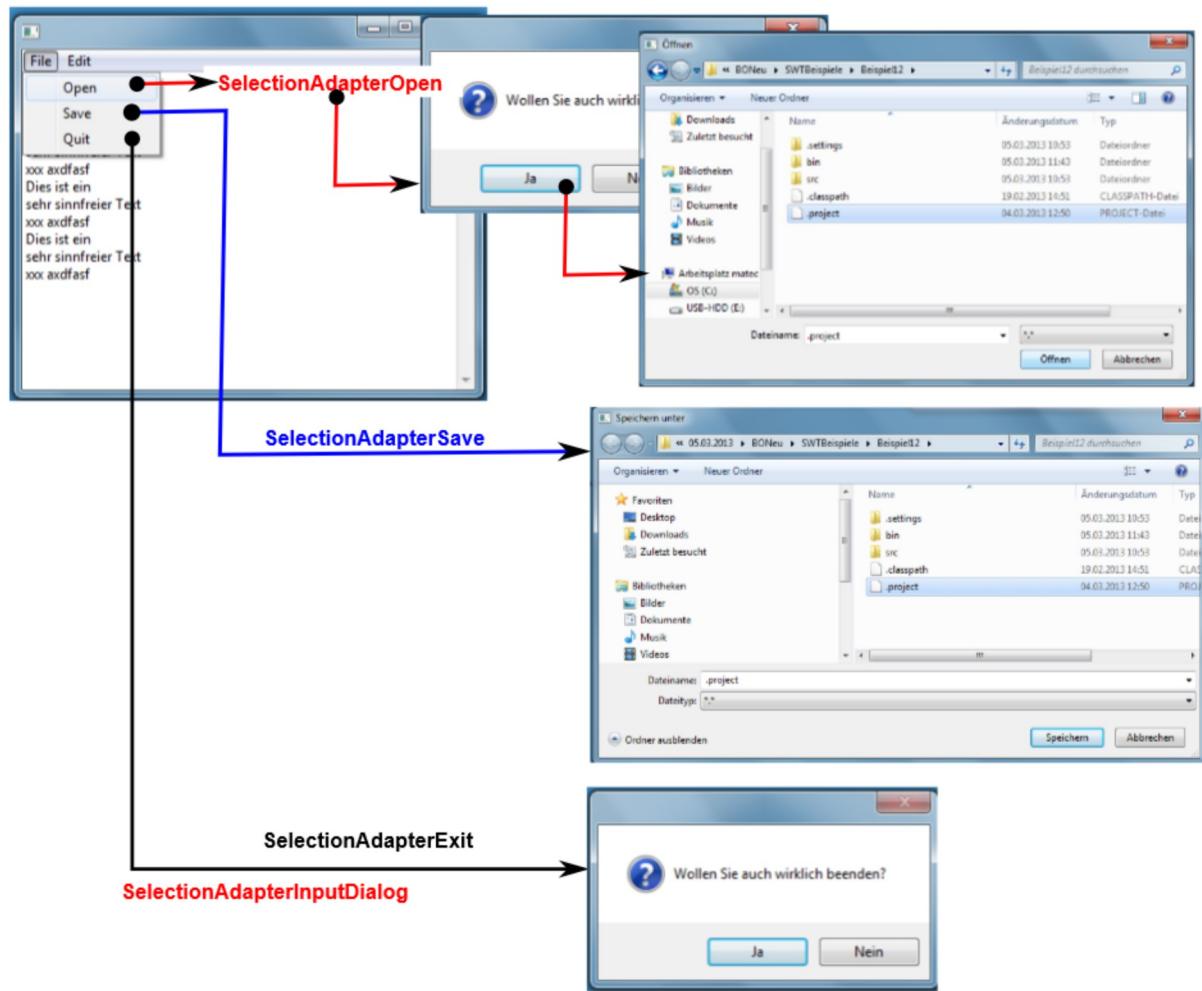


Abbildung 6.4: Beispiel12 – Teil 1: Dialogfenster

Die drei Listener, welche mit den Menüoptionen aus dem **File**-Menü verknüpft sind, verwenden vorgefertigte SWT-Standard-Dialogfenster (**FileDialog** und **MessageBox**).

KAPITEL 6. DIALOGFENSTER

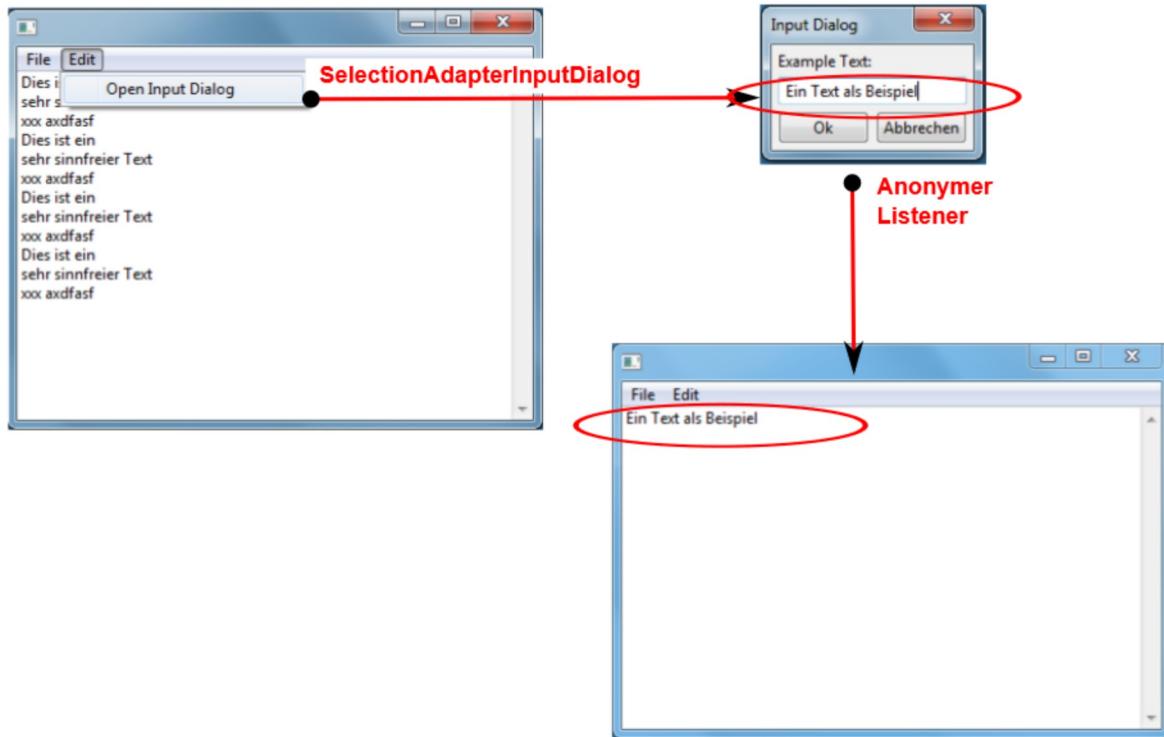


Abbildung 6.5: Beispiel12 – Teil 2: Dialogfenster

Der Listener, welcher mit der Menüoption **Edit → Open Input Dialog** verknüpft ist, öffnet ein selbst gestaltetes Dialogfenster. Nach Schließen dieses Dialogfensters fragt er den dort eingetippten Text ab und setzt ihn ins Editierfeld des Hauptfensters ein.

Das Klassendiagramm zeigt die Struktur der Gesamtanwendung:

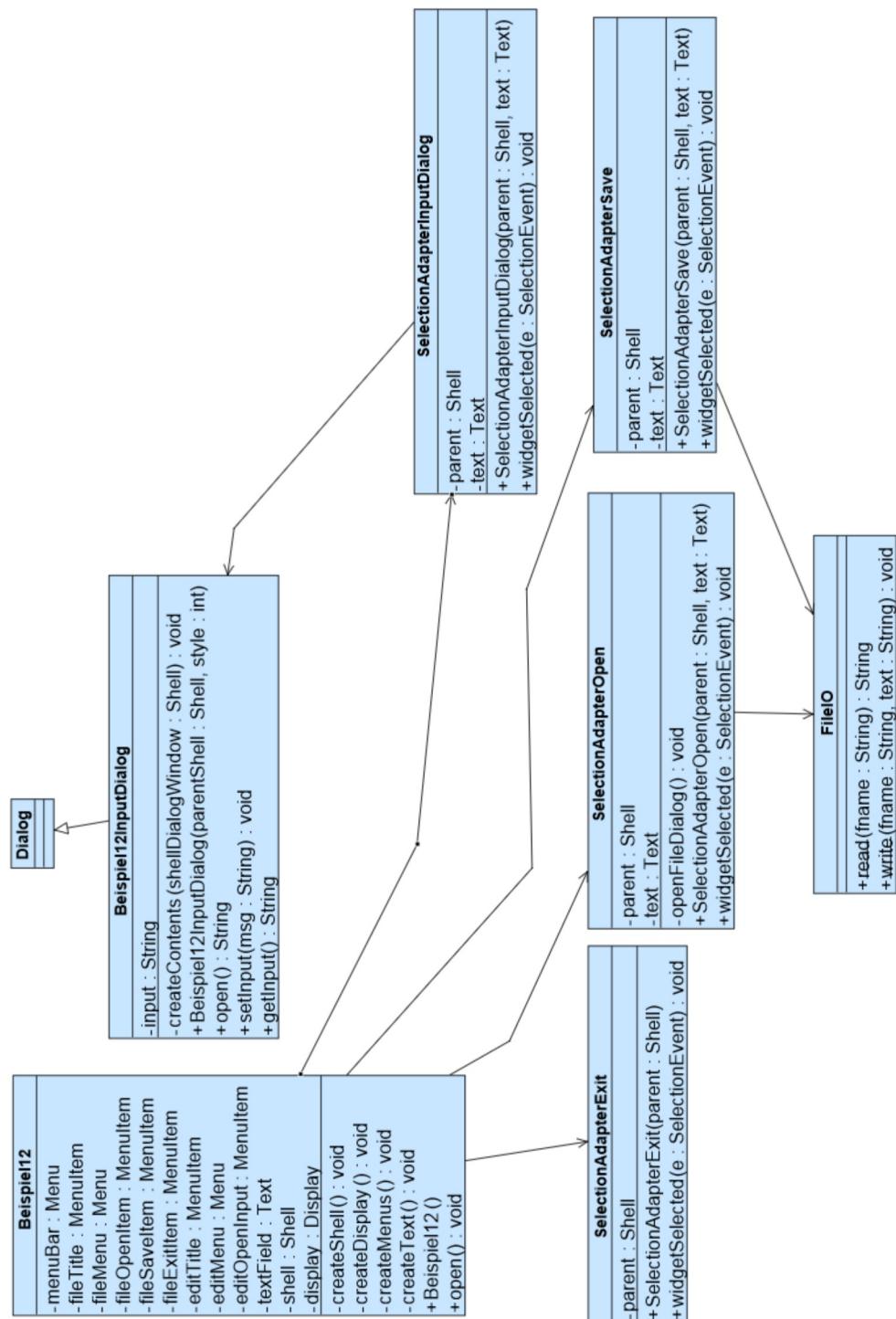


Abbildung 6.6: Beispiel12 – Klassenstruktur mit Dialogfenstern

KAPITEL 6. DIALOGFENSTER

Interessant sind für uns hier die Listener – sie sind für das Ein- und Ausblenden der Dialogfenster zuständig. Ebenfalls neu ist hier die Klasse *Beispiel12InputDialog*. Hierbei handelt es sich um das selbst implementiertes Eingabedialogfenster. Der Rest der Anwendung ist gleich wie in *Beispiel11*. Er wird daher nicht noch einmal diskutiert.

Wir zeigen im Folgenden die Implementierung der 4 Listener, sowie die des selbst gestalteten Dialogfensters. Die Gesamtimplementierung ist im E-Learning-Verzeichnis verfügbar.

Wir beginnen mit dem einfachsten Listener *SelectionAdapterExit*:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.events.SelectionAdapter;
3 import org.eclipse.swt.events.SelectionEvent;
4 import org.eclipse.swt.widgets.MessageBox;
5 import org.eclipse.swt.widgets.Shell;
6
7 public class SelectionAdapterExit
8     extends SelectionAdapter {
9
10    // Parent-Shell des Editors
11    private Shell parent;
12
13    public SelectionAdapterExit(
14        Shell parent)
15    {
16        this.parent = parent;
17    }
```

Er benötigt in seiner Reaktionsmethode nur die Parent-Shell des Hauptfensters, da er nur für das Zuklappen der Anwendung zuständig ist. Dies wird über den Konstruktor übergeben.

Die Reaktionsmethode auf der folgenden Seite

- blendet eine *MessageBox* ein, die nachfragt, ob der Nutzer die Anwendung wirklich beenden will.
- Das Einblenden geschieht mit der *open()*-Methode der *MessageBox*.
- Sobald die *MessageBox* mit einem der PushButtons zugeklappt wird, liefert die *open()*-Methode den *int*-Code des Buttons zurück, der gedrückt wurde.
- Dieser kann anschließend über eine *switch-case*-Anweisung ausgewertet werden.

```
19 public void widgetSelected(
20     final SelectionEvent e)
21 {
22     // MessageBox erzeugen: Sie hat die
23     // Buttons: Yes, No
24     // Sie hat ein Question-Icon
25     // Der Message-Text wird umgebrochen,
26     // wenn er zu lang ist.
27     MessageBox msg = new MessageBox(parent,
28         SWT.ICON_QUESTION |
29         SWT.YES | SWT.NO);
30
31     // Nachrichtentext fuer die MessageBox
32     // setzen
33     msg.setMessage(
34         "Wollen Sie auch wirklich beenden?");
35
36     // MessageBox aufklappen: Die open()-
37     // Methode liefert als int-Wert zurueck,
38     // mit welchem PushButton sie zugeklappt
39     // wurde
40     int retval = msg.open();
41     switch(retval)
42     {
43         case SWT.YES: parent.dispose(); break;
44
45         case SWT.NO: break;
46     }
47 } // end method widgetSelected()
48 } // end class SelectionAdapterExit
```

KAPITEL 6. DIALOGFENSTER

Es folgt Klasse *SelectionAdapterSave*:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.events.SelectionAdapter;
3 import org.eclipse.swt.events.SelectionEvent;
4 import org.eclipse.swt.widgets.FileDialog;
5 import org.eclipse.swt.widgets.Shell;
6 import org.eclipse.swt.widgets.Text;
7
8 public class SelectionAdapterSave
9         extends SelectionAdapter {
10
11     // Shell des Editors
12     private Shell parent;
13
14     // Textfeld des Editors
15     private Text text;
16
17     public SelectionAdapterSave(
18         Shell parent, Text text) {
19         this.parent = parent;
20         this.text = text;
21     }
```

Da zum Abspeichern des Editorinhaltes sein Textfeld benötigt wird, wird dessen Referenz an den Listener weitergegeben. Die Reaktionsmethode auf der folgenden Seite

- blendet einen *FileDialog* ein, mit dem der gewünschte Dateiname zum Abspeichern eingegeben werden kann.
- Das Einblenden geschieht mit der *open()*-Methode des *FileDialog*.
- Sobald die *FileDialog* mit einem der PushButtons zugeklappt wird, liefert die *open()*-Methode den eingegebenen Dateinamen zurück. Falls **Cancel** gedrückt wurde, wird eine *null*-Referenz zurückgeliefert.
- Der eingegebene Dateiname wird anschließend der *write()*-Methode der Klasse *FileIO* übergeben.

```
23     public void widgetSelected(
24         final SelectionEvent e) {
25         // FileDialog zum Speichern einer
26         // Datei erzeugen
27         FileDialog f =
28             new FileDialog(parent, SWT.SAVE);
```

```
29
30     // FileDialog aufklappen
31     // --> seine open()-Methode liefert
32     // den selektierten Dateinamen als
33     // String zurueck
34     String fname = f.open();
35
36     // Inhalt des Textfeldes auf
37     // die selektierte Datei
38     // schreiben
39     if(fname != null) {
40         FileIO.write(fname,
41             text.getText());
42     }
43
44 } // end method widgetSelected()
45 } // end class SelectionAdapterSave
```

KAPITEL 6. DIALOGFENSTER

Es folgt Klasse *SelectionAdapterOpen*:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.events.SelectionAdapter;
3 import org.eclipse.swt.events.SelectionEvent;
4 import org.eclipse.swt.widgets.FileDialog;
5 import org.eclipse.swt.widgets.MessageBox;
6 import org.eclipse.swt.widgets.Shell;
7 import org.eclipse.swt.widgets.Text;
8
9 public class SelectionAdapterOpen
10    extends SelectionAdapter {
11
12    // Shell des Editors
13    private Shell parent;
14
15    // Textfeld des Editors
16    private Text text;
17
18    public SelectionAdapterOpen(
19        Shell parent, Text text) {
20        this.parent = parent;
21        this.text = text;
22    } // end constructor
```

Der Konstruktor bekommt auch hier die Referenz des Hauptfenster-Textfeldes übergeben. Die Reaktionsmethode auf der folgenden Seite

- blendet eine *MessageBox* ein, die nachfragt, ob der Nutzer wirklich eine Datei öffnen will.
- Das Einblenden geschieht mit der *open()*-Methode der *MessageBox*.
- Sobald die *MessageBox* mit einem der PushButtons zugeklappt wird, liefert die *open()*-Methode den *int*-Code des Buttons zurück, der gedrückt wurde.
- Im Falle von **YES** erfolgt der Aufruf der privaten Hilfsmethode *openFileDialog()*.
- Diese bewerkstelligt auf der folgenden Seite das Einblenden und Auswerten des *FileDialog*.

```
23    public void widgetSelected(
24        final SelectionEvent e){
25        // MessageBox erzeugen: Sie hat die
26        // Buttons: Yes, No, Cancel
27        // Sie hat ein Question-Icon
```

```
28     // Der Message-Text wird umgebrochen,
29     // wenn er zu lang ist.
30     MessageBox msg = new MessageBox(parent,
31         SWT.ICON_QUESTION | SWT.YES
32         | SWT.NO | SWT.CANCEL | SWT.WRAP);
33
34     // Nachrichtentext fuer die MessageBox
35     // setzen
36     msg.setMessage(
37         "Wollen Sie auch wirklich?");
38
39     // MessageBox aufklappen: Die open()-
40     // Methode liefert als int-Wert zurueck,
41     // mit welchem PushButton sie zugeklappt
42     // wurde
43     int retval = msg.open();
44     switch(retval) {
45         case SWT.YES: openFileDialog(); break;
46         case SWT.NO: break;
47         case SWT.CANCEL: break;
48     }
49 } // end method widgetSelected()
```

KAPITEL 6. DIALOGFENSTER

```
50 //private Hilfsmethode zum
51 // Öffnen einer Datei
52 private void openFileDialog()
53 {
54     // FileDialog zum Öffnen einer
55     // Datei erzeugen
56     FileDialog f =
57         new FileDialog(parent, SWT.OPEN);
58
59     // FileDialog aufklappen
60     // --> seine open()-Methode liefert
61     // den selektierten Dateinamen als
62     // String zurück
63     String fname = f.open();
64
65     // Inhalt aus der selektierten
66     // Datei in String txt einlesen
67     if(fname != null) {
68         String txt = FileIO.read(fname);
69
70         // String txt ins Textfeld
71         // einblenden
72         text.setText(txt);
73     }
74 }
75 } // end method openFileDialog()
76 } // end class SelectionAdapterOpen
```

Die Datei mit dem selektierten Dateinamen wird geöffnet und ausgelesen. Ihr Inhalt wird anschließend ins Textfeld des Editorfensters eingeblendet.

Es folgt Klasse *SelectionAdapterInputDialog*:

```

1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.events.SelectionAdapter;
3 import org.eclipse.swt.events.SelectionEvent;
4 import org.eclipse.swt.widgets.Shell;
5 import org.eclipse.swt.widgets.Text;
6
7 public class SelectionAdapterInputDialog
8     extends SelectionAdapter {
9
10    // Parent-Shell des Editors
11    private Shell parent;
12
13    // Textfeld des Editors
14    private Text text;
15
16    public SelectionAdapterInputDialog(
17        Shell parent, Text text)
18    {
19        this.parent = parent;
20        this.text = text;
21    } // end constructor

```

Der Konstruktor bekommt auch hier die Referenz des Hauptfenster-Textfeldes übergeben. Die Reaktionsmethode auf der folgenden Seite

- blendet ein Dialogfenster vom Typ *Beispiel12InputDialog* ein. Der Nutzer kann hier einen Teststring eintippen.
- Das Einblenden geschieht mit der *open()*-Methode des *Beispiel12InputDialog*.
- Sobald der *Beispiel12InputDialog* mit **Ok** zugeklappt wird, liefert die *open()*-Methode den eingetippten String zurück.
- Dieser wird ins Textfeld des Hauptfensters eingeblendet.

```

22
23     public void widgetSelected(final SelectionEvent e)
24     {
25         // Dialogfenster erzeugen.
26         // Dialogfenster soll modal angezeigt
27         // werden.
28         Beispiel12InputDialog in =
29         new Beispiel12InputDialog(

```

KAPITEL 6. DIALOGFENSTER

```
30             parent , SWT.APPLICATION_MODAL) ;  
31  
32         // Dialogfenster aufklappen  
33         String s = in.open();  
34  
35         if(s != null) {  
36             // Eingetippten Text ins Editor-  
37             // feld einblenden  
38             text.setText(s);  
39         }  
40     } // end method widgetSelected()  
41 } // end class SelectionAdapterInputDialog
```

Zuletzt folgt die Implementierung des Dialogfensters in der Klasse *Beispiel12InputDialog*:

```
1 import org.eclipse.swt.SWT;
2 import org.eclipse.swt.events.SelectionAdapter;
3 import org.eclipse.swt.events.SelectionEvent;
4 import org.eclipse.swt.layout.GridData;
5 import org.eclipse.swt.layout.GridLayout;
6 import org.eclipse.swt.widgets.Button;
7 import org.eclipse.swt.widgets.Dialog;
8 import org.eclipse.swt.widgets.Display;
9 import org.eclipse.swt.widgets.Label;
10 import org.eclipse.swt.widgets.Shell;
11 import org.eclipse.swt.widgets.Text;
12
13 // Klasse ist frei nach Warner et. al.
14 // S. 205 ff
15 public class Beispiel12InputDialog
16     extends Dialog {
17
18     private String message;
19     private String input;
20
21     //1. Konstruktor
22     public Beispiel12InputDialog(Shell parent,
23         int style) {
24         super(parent, style);
25         message = "Example\u2014Text:" ;
26     } // end constructor
```

Die Klasse ist von *Dialog* abgeleitet. Der Konstruktor bekommt die Parent-Shell des Hauptfensters, sowie eine Style-Konstante mit der Modalitätsstufe (z. B. *SWT.APPLICATION_MODAL*) übergeben. Er ruft nur den Superklassenkonstruktor auf.

KAPITEL 6. DIALOGFENSTER

```
27 // 2. Erzeugt Layout, startet Eventloop
28 // gibt eingetippten String zurueck
29 public String open(){
30     // Shell wird nur intern verarbeitet:
31     // getParent() liefert die Shell des
32     // Hauptfensters
33     // Shell des Dialogfensters ist Kind
34     // der Shell des Hauptfensters
35     Shell shellDialogWindow =
36         new Shell(getParent());
37     shellDialogWindow.setText("InputDialog");
38
39     // Inhalt der Shell wird erzeugt
40     createContents(shellDialogWindow);
41     shellDialogWindow.pack();
```

Im ersten Abschnitt der *open()*-Methode wird eine *Shell* für den Inhalt des Dialogfensters erzeugt. Sie wird mit der privaten Hilfsmethode *createContents()* befüllt.

```
42
43     // EventLoop wird gestartet:
44     shellDialogWindow.open();
45
46     // this.getParent() liefert die Shell des
47     // Hauptfensters
48     // getDisplay() liefert das Display des
49     // Hauptfensters
50     Display display =
51         getParent().getDisplay();
52     // ACHTUNG: Das ist nun die eigene
53     // Event-Loop des Dialogfensters !!
54     while(!shellDialogWindow.isDisposed()){
55         if(!display.readAndDispatch()){
56             display.sleep();
57         }
58     }
59
60     // Achtung: An dieser Stelle ist die Shell
61     // des Dialogfensters disposed!
62     return input;
63
64 } // end method open()
```

Die zweite Hälfte der *open()*-Methode startet die EventLoop des Dialogfensters. Die Hilfsmethode *createContents()* befüllt die *Shell* des Dialogfensters. Sie nutzt hierbei ein

2-spaltiges *GridLayout*, in dem Beschriftungslabel und Textfeld jeweils beide Spalten einnehmen.

```
65 // 3. private Hilfsmethode zum
66 // Befuellen der Shell des Dialogfensters
67 private void createContents(
68     final Shell shellDialogWindow) {
69     Label label;
70     Button ok;
71     Button cancel;
72     final Text text;
73
74     // Shell des Dialogfensters bekommt
75     // 2-spaltiges GridLayout
76     shellDialogWindow.setLayout(
77         new GridLayout(2,true));
78
79     // Label erzeugen
80     label =
81         new Label(shellDialogWindow, SWT.NONE);
82     label.setText(message);
83     GridData grid = new GridData();
84
85     //Label belegt 2 Spalten
86     grid.horizontalSpan = 2;
87     label.setLayoutData(grid);
```

KAPITEL 6. DIALOGFENSTER

```
89
90     // Textfeld erzeugen
91     text =
92         new Text(shellDialogWindow,
93                 SWT.BORDER);
94     grid = new GridData(
95         GridData.FILL_HORIZONTAL);
96     // Textfeld belegt 2 Spalten
97     grid.horizontalSpan = 2;
98     text.setLayoutData(grid);
99
100    // Ok-Button erzeugen
101    ok = new Button(
102        shellDialogWindow, SWT.PUSH);
103    ok.setText("Ok");
104    grid = new GridData(
105        GridData.FILL_HORIZONTAL);
106    ok.setLayoutData(grid);
107    ok.addSelectionListener(
108        new SelectionAdapter() {
109
110            public void widgetSelected(
111                SelectionEvent e){
112                // Wenn ok gedrueckt, uebernimm
113                // Text aus Textfeld in interne
114                // String-Variablen!
115                input = text.getText();
116
117                // Zerstoere Shell des
118                // Dialogfensters --> Klapp zu!
119                shellDialogWindow.dispose();
120            } // end method widgetSelected()
121       }); // end Listener
```

```
122
123     // Cancel-Button erzeugen
124     cancel =
125         new Button(shellDialogWindow,
126                     SWT.PUSH);
127     cancel.setText("Abbrechen");
128     grid = new GridData(
129         GridData.FILL_HORIZONTAL);
130     cancel.setLayoutData(grid);
131
132     cancel.addSelectionListener(
133         new SelectionAdapter() {
134             public void widgetSelected(
135                 SelectionEvent e){
136                 input = null;
137
138                 // Zerstoere Shell des
139                 // Dialogfensters --> Klapp zu!
140                 shellDialogWindow.dispose();
141             }
142         });
143     } // end method createContents
144 } // end class Beispiel12InputDialog
```

Die beiden Buttons bekommen jeweils einen anonymen Listener, der das Zuklappen des Dialoges bewerkstelltigt. Außerdem sind die beiden Listener für das Setzen des Strings zuständig, der später von der *open()*- Methode zurückgegeben wird.

Achtung

Für sehr kurze Ablaufsequenzen sind anonyme Listener sehr gebräuchlich. Lange Workflows werden aber i. d. R. in nicht-anonyme Listener ausgelagert (Eigene Dialogmanagementschicht).

KAPITEL 6. DIALOGFENSTER

Kapitel 7

Internationalisierung

Die folgenden Abschnitte erläutern, auf was geachtet werden muss, wenn gleichartige Oberflächensoftware für Kundenkreise unterschiedlicher Nationalitäten geschrieben wird.

7.1 Internationalisierung vs. Lokalisierung

Internationalisierung von Software heisst, SW so aufzubauen, dass sie ohne Quellcode-Änderung an andere LandesSprachen angepasst werden kann. Im englischsprachigen Raum wird der Begriff **Internationalization** auch mit *i18n* abgekürzt. Die Zahl 18 steht hierbei für die 18 Zeichen zwischen i und n.

Das Gegenstück dazu ist die **Lokalisierung** von Software: Dieser Begriff meint die Anpassung/Installation einer Software auf einem konkreten System in einer konkreten Landes-Sprache. Auch hierfür gibt es im englischsprachigen Raum eine Abkürzung gleicher Logik für Localization: *l10n*.

Sie selbst haben vermutlich schon häufig mit internationalisierter Software gearbeitet. Bei den gängigen Office-Paketen wird beispielsweise zum Installationszeitpunkt entschieden, in welcher Landessprache die Darstellung der Benutzungsoberflächen erfolgt.

7.2 Vorgehen bei der Internationalisierung von Software

- Bisher: Titeltexte und Menübeschriftungen wurden im Quelltext „hard coded“ eingefügt.
- Nachteil dieser Vorgehensweise: Bei ursprünglich z. B. englischsprachigen Oberflächen ist die Sprachumstellung zeitaufwendig und fehleranfällig („Vergessen“ von Titel-Strings).

KAPITEL 7. INTERNATIONALISIERUNG

- Neu: Internationalisierung einer einmal erstellten und betitelten Oberfläche mittels Konfigurationsdateien / -datenbanken.

Die unterschiedlichen Frameworks haben unterschiedliche Vorgehensweisen für die Durchführung von Internationalisierungsmaßnahmen. Allen gleich ist jedoch: Die zu übersetzenden Strings werden in Übersetzungsdateien ausgelagert. Die Dateinamen enthalten in der Regel die **Länderkürzel** nach **ISO 3166** des Landes, in dessen Landessprache die Sprachübersetzung erfolgt.

Es folgt eine Tabelle mit einigen bekannten Landeskürzeln:

Landeskürzel nach ISO 3166	
Kürzel	Land
DE	Deutschland
ES	Spanien
FR	Frankreich
GR	Griechenland
MY	Malaysia
NL	Niederlande
PT	Portugal
RU	Russland
TH	Thailand
TR	Türkei
US	USA

Tabelle 7.1: Länderkürzel nach ISO 3166 (Auszug aus [19])

7.3 i18n in Java

In Java und damit auch im SWT erstellen wir Übersetzungsdateien, die i.d.R. unter dem Dateinamen `MessageBundle_kürzel_KÜRZEL.properties` abgelegt werden.
Einige Beispiele:

Namen von Übersetzungsdateien	
Dateiname	Land
<code>MessageBundle_de_DE.properties</code>	Deutschland
<code>MessageBundle_es_ES.properties</code>	Spanien
<code>MessageBundle_fr_FR.properties</code>	Frankreich
<code>MessageBundle_ru_RU.properties</code>	Russland
<code>MessageBundle_tr_TR.properties</code>	Türkei
<code>MessageBundle_us_US.properties</code>	USA

Tabelle 7.2: Dateinamen für i18n-Übersetzungsdateien in Java

In Java werden häufig die Originalbeschriftungen englischsprachig durchgeführt. Sie landen dann in einer Datei mit dem Namen

`MessageBundle.properties`

Die Übersetzungen der anderen Landessprachen werden dann anschließend in einer Datei mit passendem Landeskürzel im Namen angelegt. Das Format der Übersetzungsdateien ist recht überschaubar:

```

1 count=Count
2 save=Save
3 value=Value:
4 open=&Open

```

Listing 7.1: Datei `MessageBundle.properties`

KAPITEL 7. INTERNATIONALISIERUNG

```
1 count=Zählen  
2 save=Speichern  
3 value=Wert:  
4 open=Öffnen
```

Listing 7.2: Datei *MessageBundle_de_DE.properties*

Auf der linken Seite steht ein „Key“ für das zu übersetzende Wort. Auf der rechten Seite der Zeile steht jeweils das übersetzte Wort selbst. Der „Key“ ist der Name, unter dem das Wort dann in den Quelltext eingebunden wird.

Das Vorgehen bei der Internationalisierung wird nun anhand von *Beispiel3i18n* gezeigt. Das Gesamtbeispiel finden Sie auf der E-Learning-Plattform.

Schritt 1: Erstellung der Internationalisierungsdateien für die Oberflächenbeschriftung:

Hierbei werden die Internationalisierungsdateien in allen benötigten Landessprachen erstellt. In der Praxis wird dieser Schritt oft von Sprachexperten durchgeführt.

Schritt 2: Internationalisierung der *main()*-Klasse:

Damit die Internationalisierungsdateien zur Laufzeit korrekt interpretiert werden, müssen sie beim Start der Applikation eingelesen werden. Dies geschieht mit Hilfe der Java-Klassen *Locale* und *ResourceBundle*. Die Klasse *Locale* ist für die Interpretation der Länderkürzel zuständig. Die Klasse *ResourceBundle* ist für die Interpretation der übersetzten Strings zuständig.

Wir zeigen die internationalisierte *main()*-Klasse *Beispiel3Main*. Das Programm wird z. B. mit den Kommandozeilenparametern *de* und *DE* gestartet:

```
1 import java.util.Locale;
2 import java.util.ResourceBundle;
3
4 public class Beispiel3Main {
5
6
7     // Beispiel 3: SWT-Oberflaeche mit
8     // ausgelagerter Praesentationsschicht
9     public static void main(String[] args) {
10
11         String language; // Sprache
12         String country; // Landeskuerzel
13
14         // Wenn keine Internationalisierungsdatei
15         // angegeben wurde: Nimm die Datei ohne
16         // Laenderkuerzel!
17         if (args.length != 2) {
18             language = new String("en");
19             country = new String("US");
20         } else {
21             language = new String(args[0]);
22             country = new String(args[1]);
23         }
24     }
25 }
```

Über die Kommandozeile wird dem Programm bei Bedarf ein Landeskürzel mitgegeben. Falls dies nicht der Fall ist, so wird angenommen, dass die „Standard-Übersetzungsdatei“ mit den englischsprachigen Begriffen verwendet wird.

KAPITEL 7. INTERNATIONALISIERUNG

```
24     // Lokalisierung fuer die
25     // verlangte Landessprache
26     Locale currentLocale;
27
28     // ResourceBundle haelt
29     // die uebersetzten
30     // Nachrichten
31     ResourceBundle messages;
32
33     // Interpretation der Landeskuerzel
34     currentLocale =
35         new Locale(language, country);
36
37     // Wir suchen nach einer Datei
38     // MessageBundle_xx_XX.properties
39     // xx ist der Landeskuerzel
40     // z. B. de DE fuer Deutschland
41     messages = ResourceBundle.getBundle(
42         "MessageBundle",
43         currentLocale);
44
45     // Oberflaeche mit allen Kind-Widgets
46     // neu erzeugen
47     Beispiel3 oberflaeche = new Beispiel3(messages);
48
49     // Event-Loop starten
50     oberflaeche.open();
51
52 } // end main()
53 } // end class Beispiel3Main
```

Zuerst wird anhand der Landeskürzel das *Locale*-Objekt erzeugt. Dieses wird anschließend dem neu zu erzeugenden *ResourceBundle*-Objekt übergeben. Das *ResourceBundle*-Objekt benötigt außerdem das Datei-Präfix (hier: „MessageBundle“). Nach der Erzeugung wird es an alle Java-Objekte, die Strings übersetzen müssen, über den Konstruktor weitergereicht.

Schritt 3: Internationalisierung der Klassen, die übersetzte Strings verwenden:

Alle Klassen, Beschriftungen oder Nachrichtenausgaben durchführen, müssen nun noch wie folgt instrumentiert werden:

```

1 import java.util.ResourceBundle;
2 import org.eclipse.swt.SWT;
3 import org.eclipse.swt.graphics.Font;
4 import org.eclipse.swt.layout.FillLayout;
5 import org.eclipse.swt.widgets.Button;
6 import org.eclipse.swt.widgets.Display;
7 import org.eclipse.swt.widgets.Label;
8 import org.eclipse.swt.widgets.Shell;
9
10 public class Beispiel3 {
11
12     private Display display;
13     private Shell shell;
14
15     // Kind-Widgets der Shell
16     private Label label;
17     private Button buttonCount;
18     private Button buttonSave;
19
20     // Internationalisierungs-
21     // Schnittstelle
22     private ResourceBundle messages;
23
24     public Beispiel3(ResourceBundle messages){
25
26         // Internationalisierungs-Datei
27         // laden
28         this.messages = messages;
29
30         // ... Rest des Konstruktors ...

```

Überall dort, wo früher direkte Beschriftungen durchgeführt wurden, ändert sich nun die Beschriftungstechnik wie folgt:

```

32     // 10. Titel-String aus MessageBundle-
33     // Datei herausfischen
34     buttonSave.setText(
35         messages.getString("save"));
36
37 } // end constructor Beispiel3()

```

Hier wird auf das *ResourceBundle*-Objekt zugegriffen. Mit seiner Methode *getString()*

wird die Übersetzungsdatei nach einem „Key“ namens „save“ durchsucht. Seine Übersetzung wird als Beschriftung eingesetzt.

Merke

- Internationalisierung wird mit **i18n** abgekürzt. Hiermit ist gemeint, Software so zu schreiben, dass sie für das Hinzufügen einer weiteren Landessprache nicht mehr geändert werden muss.
- Lokalisierung wird mit **I10n** abgekürzt. Hiermit ist gemeint, eine Software auf eine konkrete Landessprache anzupassen.
- Kernstück von Internationalisierungsframeworks sind Übersetzungsdateien. Sie werden i. d. R. für jede gewünschte Landessprache geschrieben.
- Die Beschriftung von GUI-Elementen erfolgt dann nicht mehr direkt, sondern mit Hilfe der Strings aus den Übersetzungsdateien.
- Es gibt einen **ISO-Standard 3166**, in dem **Landeskürzel** für die jeweiligen Länder festgelegt sind.
- In Java benötigen Sie:
 - die Klasse *Locale* für die Interpretation der Landeskürzel an den Namen der Übersetzungsdateien, und
 - die Klasse *ResourceBundle* für die Interpretation des Inhaltes der Übersetzungsdateien.