Eclipse-GUIs einmal anders

Markus Völter, voelter@acm.org, www.voelter.de
Bernd Kolb, b.kolb@kolbware.de, www.kolbware.de

Einleitung

Wie schafft man es, einheitliche, flexible und gut testbare grafische Oberflächen mit minimalem Zeitaufwand zu erstellen und das auch noch ohne dass sich der Anwendungsprogramierer um die zugrunde liegende Oberflächenbibliothek kümmern muss? Dieser Artikel beschäftigt sich mit einem GUI-Framework, welches entwickelt wurde um genau dies zu ermöglichen.

Der Artikel will dabei nicht Werbung für dieses spezifische Framework machen, sondern anregen, auf diese Art und Weise Oberflächen zu erstellen.

Das Framework welches hier beschrieben wird wurde zunächst für SWT entwickelt, mittlerweile aber ebenfalls nach Swing portiert. Wir wollen uns in diesem Artikel mit der SWT-Version des Frameworks beschäftigen.

SWT

Das wohl bekannteste System mit SWT-Oberfläche ist Eclipse [1]. Den meisten dürfte Eclipse als Java-IDE bekannt sein. Eigentlich ist Eclipse jedoch eine Plattform für verschiedenste Anwendungen. Die entsprechenden Erweiterungen für Eclipse werden Plug-Ins genannt. *Ein* Plug-In ist dabei das JDT, die Java-IDE. Eine der herausragenden Eigenschaften von Eclipse und ihren Plug-Ins ist die Oberfläche welche deutlich schneller ist als traditionell mit Java Swing oder AWT entwickelte Oberflächen und zudem das native *Look and Feel* des jeweiligen Betriebssystems besitzt. Dies ist SWT zu verdanken, da es im Unterschied zu Swing die Oberflächen-API des jeweiligen Betriebssystems verwendet. Auf diese greift es es mit einer spezifischen DLL zu. Daher sehen viele der von SWT zur Verfügung gestellten Widgets exakt gleich aus wie auf dem jeweiligen Betriebssystem.

Zudem verfolgt SWT eine andere Art der Ressourcenverwaltung als Swing. In SWT muss schon beim Anlegen eines GUI Elements das spätere Elternobjekt angegeben werden. Dies hat zur Folge, dass wenn das Elternteil „vernichtet“ wird ebenfalls alle seine Kinder mit beseitigt werden. Dadurch werden automatisch alle nicht mehr benötigten Ressourcen wieder freigegeben. (Eine Ausnahme sind zum Beispiel Images; um deren Entsorgung muss man sich selbst kümmern). Dieses Vorgehen ist insbesondere deshalb wichtig, weil SWT-Ressourcen direkt mit den Ressourcen des zugrundeliegenden Betriebssystems zusammenhängen. Diese dürfen nicht erst freigegeben werden, wenn der Java Garbage-Collector irgendwann einmal aktiv wird (oder auch nicht). Eine deterministische Ressourcenfreigabe ist nötig! Eine tiefere Einführung in SWT gibt es in diversen Artikel unter [2].

Das GUI Framework

Das Framework, welches hier vorgestellt werden soll, ist im Rahmen eines größeren Projekts entstanden in dem Eclipse sowohl als IDE als auch als Zielplattform für den Anwendungsclient verwendet wird. Es besteht aus zwei Hauptbestandteilen:

- Das Editor-Framework dient dazu, Daten in Editoren darzustellen, bzw. zu editieren
- Teil zwei dient zur Darstellung und Bearbeitung von Daten innerhalb eines Baumes.

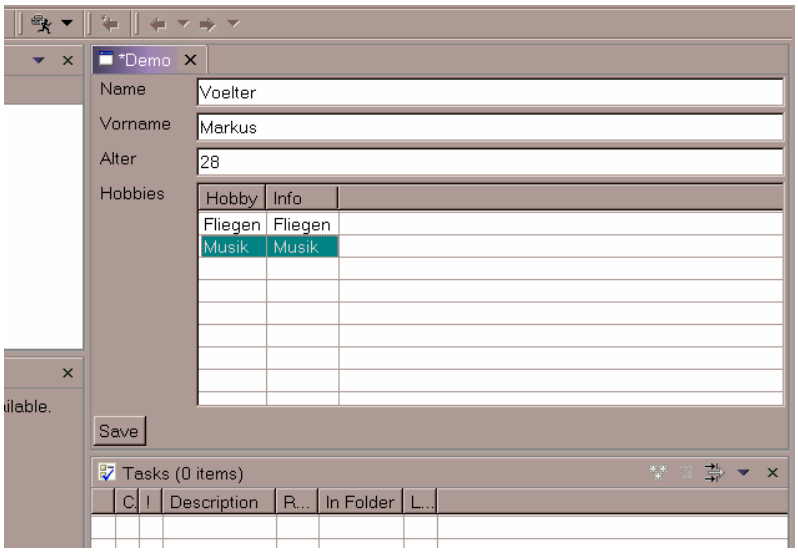
Warum noch ein Framework? Die Punkte auf die bei der Entwicklung des Frameworks besonders geachtet wurden, sind die Trennung der Daten, mitsamt deren Validierung, von ihrer späteren Darstellung. Dies geschah aus mehreren Gründen:

- Zum einen sollte sich der Anwendungsprogrammierer nicht um das SWT-spezifische Layout bei der Darstellung seiner Daten kümmern müssen, um eine möglichst einheitliche Oberfläche zu bekommen.
- Zum zweiten sollte die Darstellung und Eingabe von Daten sowie deren weitere Bearbeitung in einem Eclipse-Editor, einem Eclipse-View, einem reinen SWT-Wizard oder SWT-Dialog möglich sein ohne etwas an den Maskendefinitionen ändern zu müssen.
- Des weiteren war es wichtig, die Erstellung von Editoren oder Bäumen möglichst einfach zu halten. Diese sollte immer gleich erstellt werden und der Programmierer sollte sich nicht um Details einzelner Widgets für den Editor bzw. der Elemente für den Baum kümmern müssen.
- Und nicht zuletzt sollten die Eingabe und Validierung der Daten sowie die Abhängigkeiten der Felder untereinander auch ohne die Oberfläche zu sehen automatisch testbar sein, um Unit-Tests von Bearbeitungsmasken effizient zu ermöglichen.

Des weitern gab es die Anforderung, möglichst viel der in Eclipse standardmäßig vorhandenen Funktionalität zu nutzen, da das ganze eben in einem Eclipse-Client laufen sollte.

Editoren

Um den Umfang des Artikels nicht zu sprengen wollen wir uns im Folgenden im wesentlichen mit dem Editorframework beschäftigen. Ein Editor sieht folgendermaßen aus:



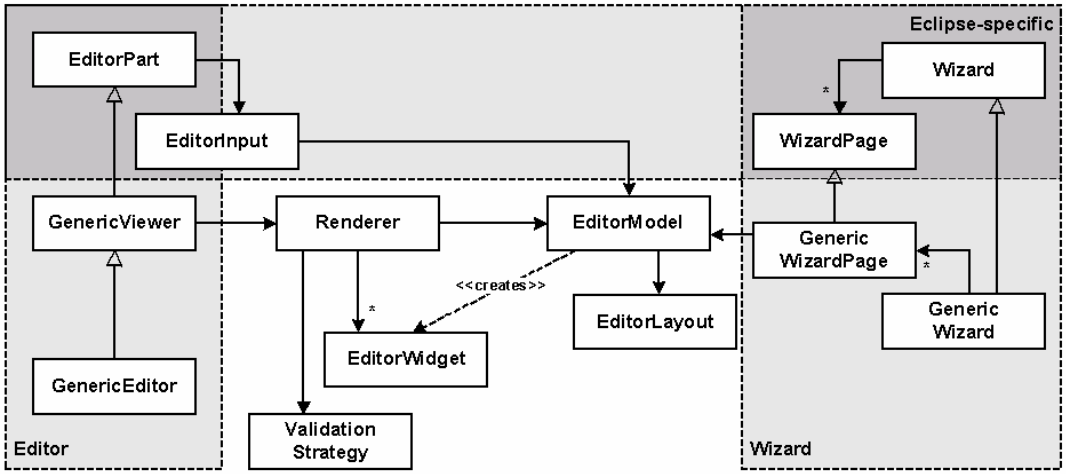
Dabei ist zunächst erkennbar, dass es sich um einen Editor im Sinne von Eclipse handelt. (Dies ist jedoch nicht zwingend notwendig, da der größte Teil des Frameworks auch ohne Eclipse – also rein mit SWT – anwendbar ist, bzw. „Editoren" auch in Views, Wizards oder Dialogen darstellbar sind) Die Labels zu den einzelnen Widgets stehen standardmäßig links, die Widgets selbst stehen rechts. Unten links steht, wenn vorhanden, der Speichern- bzw. Abbrechenknopf, rechts daneben evtl. zusätzliche Buttons. Einem Editor liegt immer ein Datenobjekt zu Grunde: in diesem Fall eine Instanz der Klasse *Person,* die Daten einer Person speichert (s.u.).

Um einen solchen Editor zu erstellen, wird eine Klasse die von *EditorModel* abgeleitet ist definiert. Ein Editormodel kümmert sich um die Kommunikation des Datenobjektes (hier: *Person*) mit den *EditorWidgets* (unabhängig von den SWT-Widgets) und die Validierung der eingetragenen Daten. Außerdem kann das Editormodell Aktionen anstoßen, die nach der Eingabe eines Wertes bzw. vor oder nach dem Drücken des Speichernknopfes oder anderer Knöpfe ausgeführt werden sollen. Diese Aktionen können sowohl das Editormodel selbst (Aktivieren oder Deaktivieren eines Knopfes oder eines Widgets), andere Editoren oder Views (per Senden eines Events) oder ganz andere GUI Elemente betreffen.

Das Editormodell, bzw. dessen Widgets wissen nichts über die spätere Anordnung im Editor. Dazu wird im Editormodell ein *EditorLayout* gesetzt, welches später vom *Renderer* dazu benutzt wird die Widgets anzuordnen (Standard ist dabei wie oben ersichtlich: Label links, Widget rechts daneben). Die Widgets, die vom *EditorModel* verwendet werden, sind nicht einfach nur SWT-Widgets, sondern Wrapper um solche. Diese bilden die verschiedenen SWT-Widgets auf eine einheitliche Schnittstelle ab, das Framework kann somit jedes Widget einheitlich ansprechen um Daten zu setzen, bzw. um diese auszulesen (sie sind also ein Adapter im GoF-Sinne [3]). Des weiteren macht diese Zwischenschicht

das Framework nahezu bibliothekenunabhängig, da nur diese Wrapper sowie einige Framework-Klassen geändert werden müssen wenn eine andere GUI-Bibliothek eingesetzt werden soll; alle Modellklassen (also die Klassen die am häufigsten vorkommen und die Anwendungslogik beinhalten) müssen nicht angepasst werden.

Ein Blick auf das Objektmodell und einige Erläuterungen dazu:



- Die grau hinterlegten Klassen sind Eclipse-spezifisch. Sie haben mit der Funktionalität des Framework nichts zu tun und dienen nur dazu Editoren, Views und Wizards innerhalb von Eclipse darstellen zu können.
- Die drei weiß hinterlegten Klassen *Renderer*, *EditorWidget* und *EditorLayout* sind SWT-spezifisch. Diese Klassen und ihre Unterklassen müssen bei einer Portierung auf einen andere Bibliothek angepasst werden.
- Eine Instanz von *ValidationStrategy* wird in einem *EditorModell* gesetzt und bestimmt zu welchem Zeitpunkt die Inhalte des Editors überprüft werden sollen. Dies kann z.B. nach jeder Eingabe oder erst vor dem Speichern geschehen.
- Editormodelle sind die zentralen Klassen die einen Editor aus Sicht des Frameworks beschreiben – Details siehe unten.
- Beim *GenericEditor* handelt es sich um einen Eclipseeditor, dem ein beliebiges *EditorModel* übergeben werden kann und sich dann mit Hilfe des *Renderers* die Eigentliche, angezeigte Oberfläche erstellt.
- Der *Renderer* ist verantwortlich für den Lebenszyklus eines Editormodels; vom erstellen, anordnen der Labels und Widgets, ändern, auf Fehler überprüfen, Maßnahmen einleiten bis zum Speichern und Schließen des Editors.

Ein Beispiel

Um das ganze etwas anschaulicher zu gestalten wollen wir uns jetzt einmal ein Beispiel ansehen. Wir erstellen den oben erwähnten Editor zum Bearbeiten von Personendaten. Dazu beginnen wir mit der Datenklasse, die später im Editormodell angezeigt und editiert werden soll:

```
package guifw.demo;
public class Person {
 private String name;
 private String firstName;
 private int age;
 private String gender;
 // public getter und setters…
}
```

Nun erstellen wir das *EditorModel*. Ein Editormodell muss wie erwähnt von *EditorModel* erben und besitzt als Attribut immer eine Instanz der Klasse, deren Daten das Editormodell bearbeitet, in diesem Fall *Person*. Diese wird typischerweise im Konstruktor übergeben.

```
package guifw.demo;
public class PersonEditorModel extends EditorModel {
    private Person person;
    public PersonEditorModel( Person p ) {
         person = p;
    }
```

Nun wird für jedes Feld eine Integerkonstante angelegt. Diese Konstanten müssen fortlaufend sein und bei Null beginnen. Mit den Werten dieser Konstanten greift später der *Renderer* auf ein *EditorWidget* des Editormodells zu.

```
    public static final int NAME = 0;
    public static final int FIRST = 1;
    public static final int AGE = 2;
    public static final int GENDER = 3;
```

Als nächstes müssen wir nun die Anzahl der Felder des Editormodells festlegen. Mittels dieser Methode erhält der *Renderer* das Wissen über die Anzahl der darzustellenden Widgets.

```
    public int getCount() {
        return 4;
    }
```

In Rahmen von *initialize()*wird die Konfiguration des Editors vorgenommen. Hier wird zum Beispiel festgelegt welche Knöpfe zu sehen sind, oder welche *ValidationStrategy* verwendet wird. Wir belassen es hier beim Standard und implementieren die Methode leer.

```
    public void initialize() {
    }
```

Wie der Name schon sagt wird in der folgenden Methode der Titel des Editormodells gesetzt. Dieser erscheint in Eclipse im Reiter des Editors oder der Views bzw. in Dialogtitel.

```
public String getTitle() {
      return "Person Editor";
}
```

Ab hier beginnt der eigentlich interessante Teil des Editormodells. Für jedes Feld kann ein Label gesetzt werden. Dies wird je nach Layout positioniert oder gar nicht dargestellt. Beim *DefaultEditorLayout* stehen die Labels immer links von den Editorwidgets. In dieser wie auch in den folgenden Methoden ist auch erkennbar wie die Integerkonstanten verwendet werden: Der Renderer ruft für jedes der Widgets die entsprechende Operation mit dem passenden Index auf.

```
public String getLabel(int index) {
      switch (index) {
            case NAME : return "Name";
            case FIRST: return "Vorname";
            case AGE: return "Alter";
            case GENDER: return "Geschlecht";
            default: return null;
      }
}
```

In *createWidget()* werden nun die Eingabewidgets definiert. Die Widgets die hier angesprochen werden müssen alle von der Klasse *EditorWidget* erben und bestimmten Konventionen genügen – Details würden den Rahmen dieses Artikels sprengen.

```
public EditorWidget createWidget(int index) {
  switch (index) {
     case NAME : return new TextWidget();
     case FIRST: return new TextWidget();
     case AGE: return new TextWidget();
     case GENDER: return new
            ComboBoxWidget(new String[]{"male", "female"});
     default: return null;
  }
}
```

checkContent() wird verwendet um, die Inhalte der Widgets zu überprüfen. Ist ein Inhalt nicht korrekt wird eine *FieldNotValid* Exception geworfen. Deren Message wird dann im Editor als Tooltip über dem betroffenen Label angezeigt – die Fehlermeldung.

```
public void checkContent(int index, Object content)
       throws FieldNotValid {
  switch (index) {
    case AGE:
      try{
        int age = Integer.parseInt(content);
        if( age < 0)
```

```
            throw new FieldNotValid("Age is negative!");
          } catch (NumberFormatException nfe){
            throw new FieldNotValid("Age is not a number!");
          }
        }
    }
```

getContent() liefert den Wert des Models an das *EditorWidget*, *setContent()* macht das Gegenteil: es übernimmt den Inhalt eines Widgets zurück ins Datenobjekt. Dies geschieht z.B. nach Drücken des Speichernknopfes.

```
public Object getContent(int index) {
  switch (index) {
      case NAME : return person.getName();
      case FIRST: return person.getFirstName();
      case AGE: return String.valueOf( person.getAge() );
      case GENDER: return person.getGender();
      default: return null;
  }
}

public void setContent(int index, EditorWidget widget) {
  switch (index) {
    case NAME : person.setName( widget.getContentAsString() );
    case FIRST: person.setFirstName(widget.
                                getContentAsString());
    case AGE: person.setAge( widget.getContentAsInt() );
    case GENDER:person.setGender(widget.
                    getContentAsString());
  }
}
```

Damit wäre die Erstellung des Editormodells fertig; wir müssen den Editor jetzt noch anzeigen, beispielsweise durch ein Command[1] nach Drücken eines Knopfes. Dazu ist folgendes notwendig:

- Der *GenericEditor* muss einmal (unabhängig vom Editormodell) in der *plugin.xml* unter einer zu wählenden ID registriert werden. Mit dieser ID kann dann der Editor geöffnet werden. Wir verwenden hier *guifw.GenericEditor*
- Dann wird ein Datenobjekt und eine Modellinstanz angelegt, das Datenobjekt wird dem Editormodell übergeben.

  ```
  Person p = new Person();
  ```

[1] Das Framework verwendet das Command-Pattern [3] verwendet wird, um Aktionen zu kapseln, die dann als Menüeintrag, Knopf oder mit anderen Mitteln dargestellt werden können. Auch dieses Vorgehen erleichtert die Testbarkeit, da sowohl die Kommandos separat getestet werden können, als auch das (konfigurierbare) Vorkommen von Aktionen in der Oberfläche getestet werden kann.

```
p.setName("Kolb");
p.setFirstName("Bernd");
p.setAge(21);
p.setGender("male");

PersonEditorModel em = new PersonEditorModel(p);
```

- Zu guter letzt wird der Eclipse-Editor geöffnet.

```
IWorkbenchPage wbp = //…
wbp.openEditor(new GenericEditorInput(em),
    "guifw.GenericEditor" );
```

Soll statt des Editors ein View, ein Wizard oder ein Dialog angezeigt werden muss der letzte Schritt entsprechend angepaßt werden.

Testen von Editormodellen

Durch die strikte Trennung von Editormodel und SWT-spezifischen Dingen kann man seine Editormodelle sehr gut und einfach mit JUnit [4] testen. Dabei hilft der *EditorTester*. Im folgenden ein Beispiel eines JUnit Tests, der den Personeneditor testet. Zunächst wird im *setUp()* eine neue Person angelegt, dann das entsprechende Editormodell, und zuletzt der Tester.

```
public class SkelettEditorTest extends TestCase {
    private EditorTester tester = null;
    private EditorModel editorModel;
    private Person person;

    protected void setUp() throws Exception {
      person = new Person();
      p.setFirstName(“Bernd”);
      editormodel = new PersonEditorModel(person);
      tester = new EditorTester(em);
    }
```

Der *EditorTester* simuliert den Eclipse-Editor und hält intern einen *Renderer* der, wie üblich, die Darstellung des Editors und dessen Lifecycle übernimmt (bzw. hier simuliert). Nun beginnt der eigentliche Test.

Wir initialisieren dazu zunächst den Editor. Dieses erzeugt all die Widgets und kopiert u.a. auch die Attribute von der Person in die Editorfelder.

```
tester.setupEditor();
```

Dann verifizieren wir, ob das Eingabefeld für den Vorname auch wirklich diesen Vorname enthält – diese wurde dem Personenobjekt im Konstruktor gesetzt.

```
assertEquals( tester.getWidget( em.FIRST ).
                     getContent(), "Bernd" );
```

Im letzten Schritt setzen wir nun den Wert des Widgets und rufen danach *save()* auf dem Tester auf. Jetzt muss geprüft werden ob der Vorname der Person zu dem geändert wurde was wir ins Widget geschrieben haben.

```
        tester.getWidget( em.FIRST ).setContent("XYZ");
        tester.save();
        assertEquals( p.getFirstName(), "XYZ" );
      }
    }
```

Das ist alles. Es läßt sich mit diesem Verfahren – unter der Annahme dass die Widgets funktionieren – die komplette Fachlichkeit des Editormodells testen, auch Abhängigkeiten zwischen Feldern oder die korrekte Validierung der Daten.

Events

Um im Framework zwischen verschieden Komponenten zu kommunizieren, kann sich eine Komponente für einen oder mehrere bestimmte Typen von Events entscheiden, oder sich für eine bestimmte Eventquelle anmelden. Beim Versenden des Events wird ein Datenobjekt mitgegeben. So können z.B. unterschiedliche Editormodelle miteinander kommunizieren. Der Mechanismus beruht nicht direkt auf dem Observer-Pattern [3], sondern setzt einen zentralen Eventhandler ein. Damit spart man sich daß alle potentiellen Subjekte (im GoF Sinne [3]) eine Observer-Verwaltung besitzen müssen. Des weiteren können Event priorisiert und einfach geloggt werden.

Bäume

Beim 2. Teil des Frameworks handelt es sich um ein kleines Framework zur Baumbearbeitung. Es unterstützt einen eigenen, umfangreicheren Drag&Drop-Mechanismus als der von SWT. Auch hier wurde darauf geachtet, dass es eine strikte Trennung der hierarchischen Daten und dem eigentlichen *Tree* erhalten bleibt. Details hierzu müssen aus Platzgründen leider unerwähnt bleiben.

Zusammenfassung

Das hier (knapp) beschriebene Framework wird in einem grösseren Projekt (>20 Entwickler) für einen großen Automobilhersteller erfolgreich eingesetzt. Die Ziele – Testbarkeit, einheitliche GUIs sowie deren effiziente Entwicklung im Team – wurden voll erreicht. Wir hoffen mit diesem Beitrag einige Denkanstösse bzgl. GUI Entwicklung und Testen von GUIs ohne aufwendige Robots gegeben zu haben. Das Framework für SWT/Eclipse kann für nicht kommerzielle Zwecke kostenlos unter www.guifw.kolbware.de herunter geladen werden. Es kann jedoch keinerlei Haftung für die korrekte Funktionsfähigkeit übernommen werden.

Referenzen

[1] www.eclipse.org

[2] www.eclipse.org/articles

[3] Gamma, Helm, Johnson, Vlissides; *Design Patterns*; Addison-Wesley 1994

[4] www.junit.org/