1 Frameworkbasierte GUI Entwicklung

1.1 Kapitelüberblick

1.1.1 Themenüberblick

Dieses Kapitel soll einen Überblick über die Programmierung von GUIs für Eclipse-basierte Rich Clients geben. Dabei wird insbesondere auf frameworkbasierte GUI-Programmierung eingegangen. Dies bedeutet, dass ein kleines, anwendungsspezifisches Framework auf SWT/Eclipse aufgesetzt wird um die Programmierung gegenüber Eclipse weiter zu vereinfachen. Dabei ist nicht das Ziel, Eclipse komplett zu verstecken, sondern die im Rahmen der Anwendung regelmäßig benötigten Features komfortabler benutzbar zu machen.

In vielen (nicht-IDE-artigen) Rich Clients vor allem im Geschäftsumfeld findet man sehr oft Baumansichten sowie formularbasierte Eingabemasken. Für derartige Clients ist das hier vorgestellte Beispielframework gedacht.

1.1.2 Kapitelstruktur

Das Kapitel ist in sechs grobe Teile aufgegliedert. Teil 1 ist die Kapitelübersicht die Sie gerade lesen. Teil 2 zeigt zunächst ein kleines „Hello-World“ Beispiel um zu verdeutlichen, wo es im Rahmen dieses Kapitels hingeht. Teil drei wird dann einige allgemeine Ziele des Frameworks und frameworkbasierter GUI-Entwicklung im Allgemeinen erläutern, bevor Teil 4 eine ganze Reihe weiterer Features, Konzepte und Ansätze des Frameworks vorstellt. Teil 5 erläutert einige der Vorteile dieses Ansatzes. Dazu zählt unter anderem auch das effektive Testen von GUIs. Schließlich diskutiert der 6. Teil wie das Framework zur Eclipse-Philosophie passt.

1.1.3 Ziele des Kapitels

Das Kapitel verfolgt mehrere Ziele die hier kurz erläutert seien:

- Zunächst soll es Sie als Leser in die Lage versetzen mit dem hier vorgestellten Framework Anwendungen zu entwickeln.
- Es soll generell aufzeigen, wo die Vorteile frameworkbasierter GUI-Entwicklung liegen.
- Es soll aber auch generell den Ansatz frameworkbasierter GUI Entwicklung vorstellen, mit dem Ziel, Sie zu ermutigen, sich selbst für ihre Domänen eigene, ähnlich funktionierende Frameworks zu bauen bevor sie zu einem GUI-Builder greifen.
- Es soll weiterhin einige interessante Eclipse-Features zeigen – schließlich ist das hier ein Eclipse-Buch.
- Last but not Least werden einige der GoF Patterns[1] verwendet und damit ihr Nutzen illustriert.

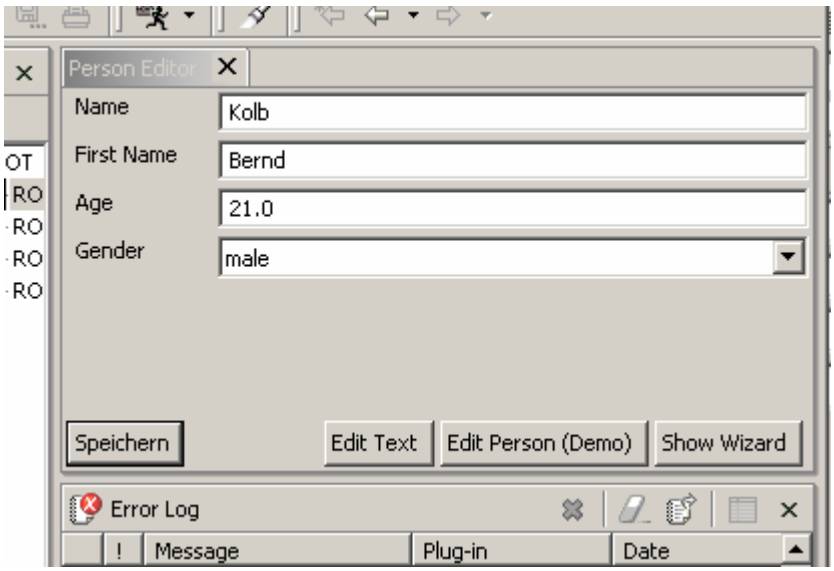
[1] GoF

1.2 Hello World

1.2.1 Der Editor – Grundlegendes und Implementierung

Ich möchte mit einem einfachen Hello World beginnen, ohne vorher auf viele der Konzepte des Frameworks einzugehen. Ich gehe dabei davon aus, dass zwischenzeitlich die wichtigsten Eclipse-Konzepte (Plugin, View, Editor, etc.) bekannt sind – ich werde dazu nichts erläutern.

Als Beispiel entwickeln wir einen kleinen Editor, der Personenobjekte bearbeiten kann. Dieser sieht (in Eclipse, als Eclipse-Editor) folgendermaßen aus:



Auf dieser Abbildung ist zunächst erkennbar, dass es sich um einen Editor im Sinne von Eclipse handelt. [2]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 die im Rahmen des Editors selbst definiert werden können. Einem Editor liegt immer ein Datenobjekt zu Grunde: in diesem Fall eine Instanz der Klasse *Person*, die Daten einer Person speichert (siehe unten).

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* (die Formularfelder, Details siehe unten) sowie die Validierung der eingetragenen Daten.

[2] Dies ist jedoch nicht zwingend notwendig, da der größte Teil des Frameworks auch ohne Eclipse – also rein mit SWT – anwendbar ist, bzw. die Formulareditoren des Frameworks auch in Views, Wizards oder Dialogen darstellbar sind)

Das Editormodell, bzw. dessen Widgets wissen nichts über die konkrete Anordnung (das Layout) im Editor. Um dies festzulegen, wird im Editormodell ein *EditorLayout g*esetzt, welches später vom *Renderer* dazu benutzt wird die Widgets anzuordnen (Standard ist dabei wie oben ersichtlich: Label links, Widget rechts daneben).

Wichtig ist dabei zu erwähnen, dass die Widgets die vom *EditorModel* verwendet werden nicht einfach nur SWT-Widgets sind, sondern Wrapper um solche. Dies dient dazu, die verschiedenen SWT-Widgets auf eine einheitliche Schnittstelle abzubilden, das Framework kann somit jedes Widget einheitlich ansprechen um Daten zu setzen, bzw. um diese auszulesen (sie sind also ein Adapter im GoF-Sinne).

1.2.2 Die Datenklasse

Nun konkret zur Implementierung des Personeneditors. Wir beginnen mit der Datenklasse, die später im Editormodell angezeigt und editiert werden soll. Diese ist prinzipiell einfach eine JavaBean mit Attributen und Getter/Setter Methoden.

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

1.2.3 Das Editormodell

Nun erstellen wir das *EditorModel*. Ein Editormodell muss wie erwähnt von der Klasse *EditorModel* erben und besitzt typischerweise ein Attribut der Klasse, die im Editor bearbeitet werden soll (hier: *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 legen wir die Anzahl der Felder des Editormodells fest. Dadurch 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 wie und wann die Felder validiert werden sollen – wir belassen das hier zunächst bei den Defaults 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";
}
```

Nun beginnt der eigentlich interessante Teil des Editormodells. Für jedes Feld kann ein Label gesetzt werden. Dies wird je nach Layout positioniert (oder auch 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 EditorWidgets an sich definiert. Die Widgets die hier angesprochen werden müssen alle von der Klasse *EditorWidget* erben und bestimmten Konventionen genügen – Details siehe später.

```
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(“Invalid Number”);
         }
   }
}
```

getContent() liefert den Wert des Models an das *EditorWidget*, *setContent()* macht das Gegenteil: es übernimmt den Inhalt eines Widgets zurück ins Datenobjekt. Letzteres 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());
    }
  }
```

1.2.4 Anzeigen des Editors

Damit wäre die Erstellung des Editormodells abgeschlossen. Wir müssen den Editor jetzt noch anzeigen, beispielsweise durch ein Ausführen eines Commands (Commands werden später erläutert) nach Drücken eines Knopfes. Dazu ist folgendes notwendig.

- Der *GenericEditor* (der Eclipse-Editor der formularbasierte Editoren darstellen kann) muss einmal in der *plugin.xml* unter einer zu wählenden ID registriert werden. Mit dieser ID kann dann der Editor geöffnet werden. Wir verwenden hier als ID *guifw.GenericEditor*

- Dann werden ein Datenobjekt und eine Modellinstanz angelegt, das Datenobjekt wird dem Editormodell übergeben.

```
Person p = new Person();
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 = //…
p.openEditor(new GenericEditorInput(em),
            ifw.GenericEditor" );
```

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

1.2.5 Diskussion

Damit ist die Programmierung des „Hello World“ Editors abgeschlossen. Wie Sie daran erkennen können haben sie keine Zeile SWT oder Eclipse Code geschrieben. Sie haben sich auch nicht um Layoutaspekte gekümmert. Genau dies ist das Ziel des Frameworks: Die Entwickler der fachlichen Oberflächen sollen von lästigen GUI Details entlastet werden. Dazu im nächsten Abschnitt mehr.

1.3 Allgemeines zur Frameworkbasierten GUI Entwicklung

1.3.1 Motivation und Hintergrund

Das Framework wurde im Laufe verschiedenster Projekte (weiter-)entwickelt. In allen Fällen wurden umfangreiche fachliche grafische Oberflächen benötigt. Im letzten Projekt im Rahmen dessen das Framework für Eclipse portiert wurde waren ca. 15 GUI Entwickler beteiligt. Diese hatten – Eclipse war damals recht neu – wenig bis gar keine Erfahrung mit SWT/Eclipse. Damit musste eine Möglichkeit geschaffen werden, wie diese Leute effizient GUIs entwickeln können. Dabei sollte eine möglichst flache Lernkurve erforderlich sein.

Weiterhin legte dieses Projekt sehr viel Wert auf (Unit-)Tests. Auch die Oberflächen sollten soweit möglich in die Tests integriert werden. Auch dafür musste entsprechende Unterstützung her.

1.3.2 Anforderungen an das Framework

Das Framework sollte die folgenden Anforderungen erfüllen:

- Die Anwendung sollte ein konsistentes Look-and-Feel besitzen. Dies wird durch die Trennung von Funktionalität sehr erleichtert.
- Es sollten verschiedene, fachlich motivierte Eingabewidgets konsistent im Rahmen der Anwendung verwendet werden. Damit war systematische Wiederverwendung ein wichtiges Thema. Das Framework sollte den Rahmen dazu bieten.
- Wie oben erwähnt, sollte das Framework so weit wie möglich SWT/Eclipse Details verstecken.
- Unit Tests sollten einfach „gescriptet“ werden können. Der Einsatz von Robots oder umfangreichen GUI Test-Tools (wie z.B. Winrunner) sollte vermieden werden.
- Neben den oben erwähnten fachlich motivierten Widgets sollte das Framework insbesondere Baumansichten sowie maskenbasierte Editoren unterstützen.
- Weiterhin sollte das Framework soweit wie möglich auf vorhandene Features von Eclipse aufsetzen; insbesondere sollten Editoren, Views, Wizards und Dialoge unterstützt werden.

1.3.3 Technische Implementierungsprinzipien

Basierend auf diesen Anforderungen, auf Erfahrung aus anderen Projekten sowie allgemeinen Best Practices zur GUI Entwicklung wurden die folgenden Prinzipien bzgl. des Frameworks aufgestellt:

- Daten und ihre Darstellung (sowie das Layout) sollen konsequent getrennt werden, in Anlehnung an das Model-View-Controller Pattern. Wenn Layout spezifiziert werden muss, dann in separaten Klassen.
- GUI Funktionalität (nicht das Layout oder die Ergonomie!) sollen per Unit Test testbar sein.
- Anwendungsspezifisches Verhalten soll in Commands gekapselt werden. Damit kann Funktionalität an verschiedenen Stellen im GUI wieder verwendet werden – alle Frameworkbestandteile erwarten Commands als Schnittstelle zur Anwendungslogik.
- Für alle Frameworkfunktionalitäten müssen *sinnvoll verwendbare* Defaults vorhanden sein (Verwendung von Strategy und Template Method Pattern).
- Einfache Features sollten einfach verwendbar sein. Für komplexere Alternativen werden entsprechende Erweiterungsmöglichkeiten (Hooks) vorgesehen.
- Nichts was es in Eclipse in passender Form bereits gibt soll nochmals neu entwickelt werden. Vereinfachende Wrapper oder Adapter um/für Eclipse-Konzepte sind erlaubt.
- Die gleichen Patterns, Namenskonventionen und Konzepte werden durch das gesamte Framework hindurch verwendet. Dadurch wurde es einfach verständlich und anwendbar.

1.3.4 Entwicklungsprozess

In den letzten Jahren haben Frameworks ja durchaus nicht immer nur eine positive Presse gehabt. Es gibt in der Tat viele generische Monsterframeworks (oder Eierlegende Wollmilchsauen) die so komplex und groß sind, dass man sie im Rahmen eines Projektes nicht effektiv einsetzen kann. Oft sind Frameworks auch von irgendwelchen Elfenbeinturm-Abteilungen entwickelt, die sich nicht wirklich um die Belange/Probleme der Projekte kümmern.

Diesen Problemen sollte im Rahmen der Entwicklung des hier vorgestellten Frameworks durch entspr. Prozesse und Vorgehensweisen entgegengewirkt werden. Ich lege ein derartiges Vorgehen jedem nahe, der Frameworks im Rahmen eines Projektes entwickelt.

- Es wurde initial nur eine minimale Basisfunktionalität entwickelt. Deren Umfang beruhte größtenteils auf Erfahrungen von früheren, ähnlich gelagerten Projekten.
- Weitere Features wurden nur implementiert, wenn es aus dem Projekt eine konkrete Anforderung dafür gab. Das Framework wurde inkrementell weiterentwickelt.
- Ein kleines von Kernentwicklern entwickelt das Framework um eine konsistente Implementierung zu gewährleisten.
- Das Framework wurde während der Weiterentwicklung ständig verwendet, insbesondere auch von Entwicklern, die das Framework nicht „erfunden“ haben.

Deren Feedback wurde konsequent aufgenommen und eingearbeitet. Bei Änderungen des Frameworks wurde die Anwendung refaktorisiert.

- Die Frameworkentwickler hatten direkt Kontakt zum restlichen Team (die Frameworkanwender). Teils haben sie selbst Anwendungen basierend auf dem Framework entwickelt.

Wenn diese Grundsätze verwendet werden, kann ein Framework letztendlich wirklich eine große Hilfe für die Anwendungsentwickler darstellen.

1.3.5 Vergleich mit anderen Ansätzen/Frameworks/Patterns

Bevor wir zur Eigentlichen Verwendung des Frameworks kommen, hier noch ein (recht knapper) Vergleich mit anderen Frameworks im Umfeld:

- Reines SWT: SWT basiert auf einem niedrigeren Abstraktionslevel. Es bietet lediglich primitive Eingabefelder und der Benutzer muss sich um Events, Layout, etc. kümmern. Diese Details wollten wir vor dem Benutzer verstecken.
- JFace: JFace befindet sich ungefähr auf dem demselben Abstraktionsniveau. Allerdings fehlen einige der wichtigen Features, beispielsweise formularbasierte Editoren. Im Bereich Bäume wäre JFace eine Alternative gewesen. Aus Konsistenzgründen wurde diese Funktionalität aber auch anders realisiert.
- MVC: Views im hiesigen Framework spiegeln Änderungen am Modell nicht automatisch wider. Es gibt keinen Publish-Subscribe Mechanismus. Das GUI wird als alleiniger Besitzer/Änderer der Daten angesehen. Wenn trotzdem eventbasierte Notifikationen nötig sind, muss dies über den Eventmanager manuell implementiert werden.
- Werkzeug & Material Framework: WAM eignet sich eher für “workbench-artige” Oberflächen wo Objekte modifiziert werden. Im Gegensatz dazu verwenden wir primär eine formularbasiere Metapher.

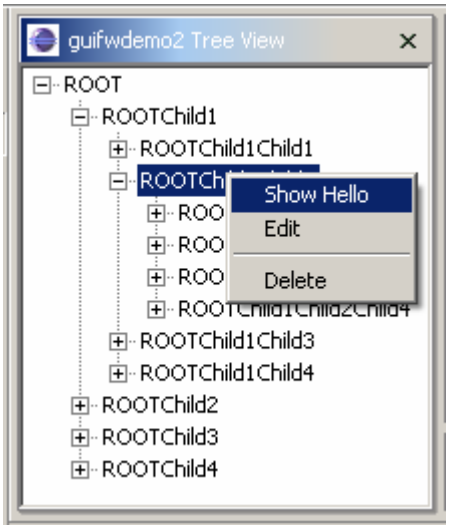
1.4 Frameworkfeatures und weitere Beispiele

1.4.1 Tree Views

Ein einfacher Tree View

Das Ziel dieses Abschnittes ist die Entwicklung eines einfachen Treeviews. Dieser berücksichtigt die oben genannten Ziele des Frameworks, insbesondere die Trennung von Darstellung und Daten. Die Datenstrukturen wissen zwar, dass sie eine Hierarchie darstellen, wissen aber nichts davon, dass sie in Bäumen dargestellt und modifiziert werden sollen.

Der folgende Screenshot zeigt worauf wir hinaus wollen. Der Baum stellt die Daten dar, erlaubt das Öffnen von Kontextmenüs und ist innerhalb eines Eclipse-Views dargestellt.



Datenklasse *DummyData*

Um einen Baum darstellen zu können, benötigen wir selbstverständlich eine hierarchische Datenstruktur. Dazu erstellen wir uns eine solche, die nie endet. Jedes Element hat wieder eine Reihe neuer Kinder. Das ist zwar in der Praxis sinnlos, macht aber die Implementierung des Beispielbaumes einfacher.

Die Klasse *DummyData* hat eine Reihe von Attributen (sie sind nachher für den Editor wichtig, im Rahmen des Baumes uninteressant), sowie die Liste der Kinder und eine Referenz auf das Elternelement.

```
public class DummyData {
   private String text = null;
   private String anotherText = null;
   private boolean bool = false;
   private String status = "offline";
   private List children = null;
   private DummyData parent = null;
   // public getters and setters for those...
}
```

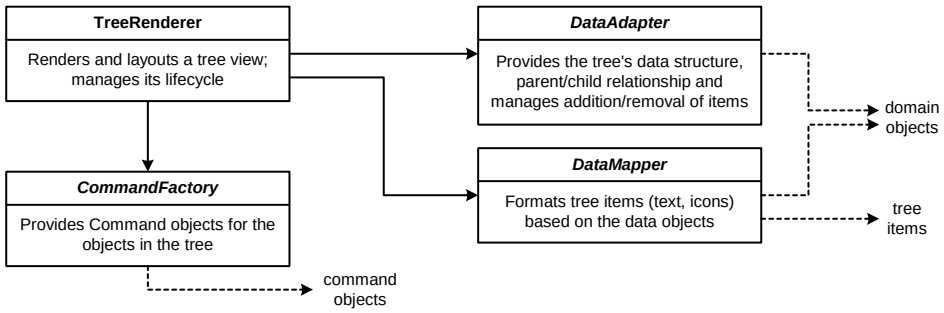
Damit das ganze in der Praxis funktioniert, muss sichergestellt werden, dass die Kinder erst bei Bedarf angefügt werden (lazy initialization). Sonst hängt sich die VM mit einem *StackOverflowError* auf.

```
public List getChildren() {
   if (children == null) {
      children = new ArrayList();
```

```
        addChild(new DummyDate(text + " Child1"));
        addChild(new DummyDate(text + " Child2"));
        addChild(new DummyDate(text + " Child3"));
        addChild(new DummyDate(text + " Child4"));
      }
      return children;
  }
```

Struktur des Frameworks

Um diese Dinge nun in einem Baum darstellen zu können, seien zunächst die Grundlagen des Treeframeworks erläutert. Es ist in der folgenden Abbildung zu sehen.



Ein *DataAdapter* bildet die hierarchische Datenstruktur auf die Anforderungen des Treeframeworks ab und kümmert sich um das Anlegen/Verschieben/Löschen von Kindern (Adapter Pattern). Der *DataMapper* definiert das Aussehen von Baumknoten für bestimmte Datenelemente (z.B. Text, Icon). Die *CommandFactory* definiert die Kontextmenüs für die verschiedenen Baumelemente. Der *TreeRenderer* ist eine Frameworkklasse und ist für die Darstellung des Baumes zuständig.

Man beachte dabei die klare Trennung der Aspekte: Der *DataAdapter* weiß nichts von der Darstellung der Baumknoten und enthält daher absolut keine Referenzen auf SWT oder Eclipse Klassen. Die Darstellungsoptionen (die von SWT *TreeItem* abhängen) finden sich alle im *DataMapper*. Die Frage, welche Operationen (Menüeinträge) für die Baumelemente vorhanden sind wird in der *CommandFactory* behandelt.

Der *TreeRenderer* benötigt um den Baum darstellen zu können je eine Instanz der drei Klassen – dadurch dass diese einzeln gesetzt (und damit auch zur Laufzeit getauscht) werden können, sind sehr flexible Bäume möglich.

Der Baum für *DummyDatas*

Der einfachst-mögliche *DataAdapter* für die *DummyData* Struktur sieht folgendermaßen aus:

```
public class ExampleDataAdapter extends DataAdapter {
   public List getChildren(Object parent) {
      DummyData d = (DummyData) parent;
      return d.getChildren();
   }

   public List getRoots() {
      return Util.makeList(new DummyData("ROOT"));
   }
  // some other operations that need not be considerd
  // now; they are implemeted "empty".
}
```

Einige Erläuterungen: *getRoots()* liefert die Wurzelelemente des Baumes zurück (hier: ein Element vom Typ *DummyData* mit dem Titel *ROOT*). *getChildren()* liefert die Kinder des als Parameter übergebenen Datenobjektes. In Falle von *DummyData* leitet die Operation die Anfrage einfach an das übergebene Objekt weiter (man kann aber z.B. auch in einer Datenbank nachschauen…).

Damit ist der *DataAdapter* fertig. Wir benötigen nun einen *DataMapper*. Dieser implementiert zunächst nur die Operation *formatTreeItemInternal()*. Das übergebene *TreeItem* ist das Objekt, welches der *TreeRenderer* zur Darstellung des übergebenen Datenobjektes verwendet. Die hier gezeigte Beispielimplementierung setzt als Text des Baumknotens einfach den Titel des Datenobjektes. Mit *TreeIcon.setIcon()* könnte auch ein Icon gesetzt werden.

```
public class ExampleDataMapper extends DataMapper {
   public void FormatTreeItemInternal(Treeltem item,
                     Object dataObject, boolean status) {
      item.setText(ob.getText(i));
   }
}
```

Nachdem wir in diesem ersten Schritt noch keine Einträge im Kontextmenü benötigen, war dies bereits alles. Es muss jetzt nur noch der Baum in einem Eclipse-View dargestellt werden. Dazu erstellt man einen ganz normalen View; implementiert werden muss dann – wie immer – die *createControl()* Operation (den Eintrag in der *plugin.xml* nicht vergessen!)

```
public void createPartControl(Composite parent) {
   TreeRenderer r = new TreeRenderer(new ExampleDataAdapter(),
          new ExampleDataMapper(), null); /* no commands */
   r.getTree(parent, false, false, false);
```

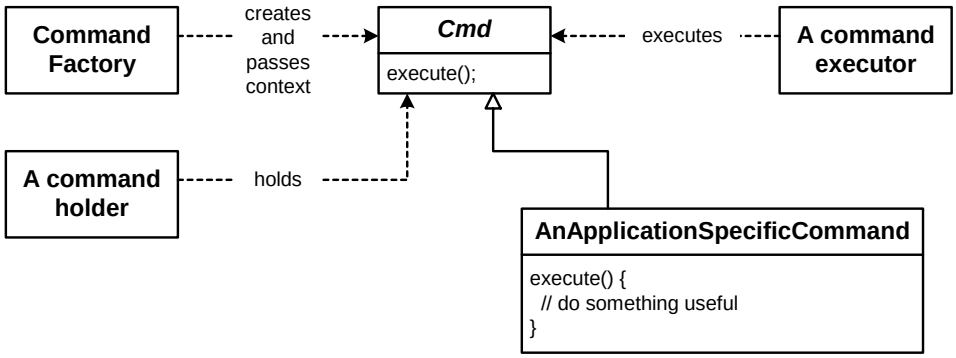
```
    //just ignore all the false's for now;
    //they are concerned with multiple selections and DnD
  }
```

In dieser Operation wird lediglich ein neuer *TreeRenderer* angelegt. Dieser bekommt eine Instanz des oben implementierten *DataAdapters* sowie des *DataMappers* mit. Dann wird *getTree()* aufgerufen und das *parent*-Composite übergeben. Das ist alles, der Baum wird nun funktionsfähig dargestellt, nachdem der entsprechende View in Eclipse geöffnet worden ist.

Der Tree mit Kontextmenüs und Kommandos

Um Kontextmenüs zu ermöglichen (und um generell Verhalten in das Framework einzuklinken) möchte ich zunächst das Konzept der Kommandos einführen (siehe GoF Buch).

Ein Kommando (oder *Command*) kapselt ein Stück Verhalten innerhalb eines Objektes. Dabei wird der Ausführungskontext des Verhaltens mitgeführt – das Verhalten wird in dem Kontext ausgeführt, der beim Anlegen des Kommandos gültig war.



Im Rahmen des hier vorgestellten Frameworks müssen Kommandos von der abstrakten Klasse *Cmd* erben. Diese bietet primär eine abstrakte Methode *execute()*, die konkrete Kommandos überschreiben müssen. *Cmd* besitzt außerdem Funktionalität um einen Titel, einen Hilfetext und ein Icon zu verwalten. Diese drei Parameter können/müssen im Konstruktor übergeben werden.

Zu beachten ist, dass Eclipse mit der *Action* Klasse selbst das *Command* Pattern implementiert. Aus verschiedenen Gründen besitzt das hier vorgestellte Framework eine eigene Implementierung – Adapter in beide Richtungen sind vorhanden.

Das folgende Beispiel zeigt ein Kommando, welches bei Ausführung *Hello World* auf dem Bildschirm ausgibt.

```
public class HelloWorldCmd extends Cmd {
   public HelloWorldCmd() {
      super("Show Hello", "");
```

```
    }

    public void execute() {
      Util.reportInformation("Hello World");
    }
}
```

Derartige Kommandos können nun für Knöpfe, Menüeinträge, etc. verwendet werden.

Kommandos im Baum

Um Kommandos im Baum (z.B. im Rahmen eines Kontextmenüs) verwenden zu können, muss eine *CommandFactory* implementiert werden. Die folgende Implementierung fügt obiges *HelloWorldCmd* zu den Kontextmenüs aller Baumknoten hinzu.

```
public class ExampleCommandFactory extends CommandFactory {
   public List getCommands(Object dataObject) {
      return Util.makeList(new
          DummyDataCommands.HelloWorldCmd());
   }
}
```

Diese Factory muss natürlich bei der Erstellung des Baumes im View mit übergeben werden:

```
public void createPartControl(Composite parent) {
   TreeRenderer r = new TreeRenderer(new ExampleDataAdapter(),
         new ExampleDataMapper(), new ExampleCommandFactory());
   r.getTree(parent, false, false, false);
}
```

Das Kommando oben ist nicht besonders nützlich. Nicht nur, weil es ein sinnloses „Hello World“ ausgibt, sondern insbesondere auch deshalb, weil das Kommando *keinen* Kontext transportiert. Es ist zustandslos. Das folgende Beispiel gibt immer noch „Hello World“ aus, kennt nun aber das Baumelement auf welchem es aufgerufen wurde (und gibt dessen Titel mit aus).

```
public class ShowHelloCmd extends Cmd {
   private DummyData data;
   public ShowHelloCmd(DummyData d) {
      super("Show Hello", "");
      data = d;
   }
```

```
    public void execute() {
      Util.reportInformation("Hello:   " + data.getText());
    }
  }
```

Bei Erzeugung einer Instanz (in der Factory) muss natürlich das aktuelle Baumelement (genauer: das darunterliegende Datenobjekt) mitgegeben werden. Praktischerweise ist dieses Objekt der Parameter der *getCommands()* Operation.

```
public class ExampleCommandFactory extends CommandFactory {
  public List getCommands(Object dataObject) {
    return Util.makeList(
      DummyDataCommands.showHelloCmd((DummyData) dataObject));
  }
}
```

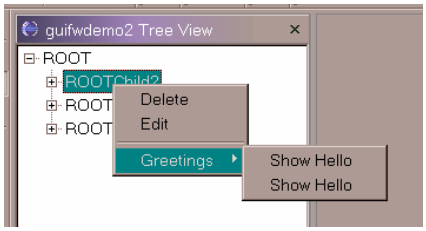
Damit können nun beliebige Aktionen auf dem gerade selektierten Baumelement bzw. seinem Datenobjekt ausgeführt werden.

Untermenüs

Es sei noch darauf hingewiesen, dass man auch Untermenüs verwenden kann. Dazu werden die Klassen *SubmenuStartCommand* und *SubmenuEndCommand* verwendet:

```
public List getCommands(Object dataObject) {
  List l = new ArrayList();
  l.add(new DummyDataCommands.DeleteCmd((DummyData)
       dataObject));
  l.add(new DummyDataCommands.OpenEditorCmd((DummyData)
       dataObject));
  l.add(new SeparatorCommand());
  l.add(new SubmenustartCommand("Greetings"));
  l.add(new DummyDataCommands.HelloWorldCmd());
  l.add(new DummyDataCommands.ShowHelloCmd((DummyData)
       dataObject));
  l.add(new SubmenuEndCommand());
  return l;
}
```

Der Baum sieht unter Verwendung der oben gezeigten Factory folgendermassen aus:



Alternative Baum-Implementierung

Wichtig bei dem oben gezeigten Ansatz ist, dass *DummyData* kein spezielles Interface implementieren musste. Das Treeframework kann beliebige Objekte darstellen, *ohne* ihre Klassedefinitionen anpassen zu müssen. Das eignet sich vor allem dann sehr gut, wenn man eben irgendwie geartete Datenobjekte *unter anderem auch* im Baum darstellen will (oder man gar keine Objekte hat: Man kann auch Datenbankstrukturen im Baum darstellen. Als „Datenobjekt“ würde dann nur z.B. der Primärschlüssel-String verwendet).

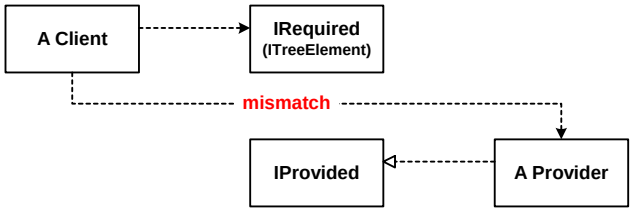
Allerdings kann der *DataAdapter* bei vielen verschiedenen Typen von Objekten im Baum unübersichtlich werden, da er sich je nach Typ anders verhalten muss. *instanceof* Kaskaden sind die lästige Folge:

```
public List getChildren(Object parent) {
   if (parent instanceof Project) {
      return ((Project) parent).getSubprojects();
   }
   if (parent instanceof Subproject) {
      return ((Subproject) parent).getItems();
   }
   // ...
   return new ArrayList();
}
```

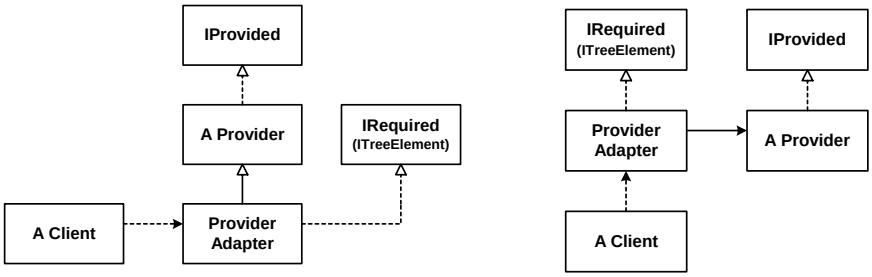
Um dies zu vermeiden, lässt sich die Implementierung auch „umdrehen“; als Folge werden der *DataAdapter* und der *DataMapper* einfach und generisch, jedoch müssen die Baumelemente nun ein bestimmtes Interface implementieren.

Das Adapter Pattern

Basis dieses Ansatzes ist ein weiteres mal das *Adapter* Pattern. Das Problem ist, dass ein Client ein Interface erwartet, welches der Provider nicht bietet.



Wir im GoF Buch erläutert, gibt es zwei Möglichkeiten, dieses Problem aufzulösen: den Klassenadapter (links in der folgenden Abbildung) und den Objektadapter (rechts).



In unserem Fall werden wir einen Klassenadapter verwenden. Wir bauen einen generischen *DataAdapter*, *DataMapper* und *CommandFactory*. Diese arbeitet mit Objekten, die das folgende Interface implementieren.

```
public interface IReadOnlyTreeElement {
   public List getChildren();
   public String getText();
   public Image getIcon();
   public List getCommands();
   public Cmd getDoubleClickCommand();
}
```

Dieses Interface fasst die von vorher bekannten Operationen *für ein Objekt* zusammen. Der generische *DataAdapter/DataMapper/CommandFactory* kann also immer das Objekt selbst fragen, wenn die Kinder oder andere Informationen benötigt werden.

Da wir dieses Feature zunächst nur für *Readonly*-Bäume implementieren, heißen die Klassen *DefaultReadOnlyAdapter*, *DefaultReadOnlyMapper*, und *DefaultCommandFactory*. Diese delegieren die Aufrufe nach Downcast auf das Interface einfach an die Datenobjeke weiter.

Außerdem stellen wir eine Factoryklasse zur Verfügung, die es erleichtert, eine *TreeRenderer* Instanz mit den entsprechenden Adaptern zu erstellen:

```
public class TreeFactory {
   public static TreeRenderer createRenderer(List roots) {
      DataAdapter adapter = new DefaultReadOnlyAdapter(roots);
      DataMapper mapper = new DefaultReadOnlyMapper();
      CommandFactory cmdf = new DefaultCommandFactory();
      TreeRenderer r = new TreeRenderer(adapter, mapper, cmdf);
      return r;
   }
}
```

Um nun die *DummyData* Objekte mit diesem Ansatz verwenden zu können, müssen diese natürlich das oben gezeigte Interface implementieren. Da wir *DummyData* nicht verändern wollen, erstellen wir einfach eine Unterklasse, die das Interface implementiert.

```
public class DummyDataTreeElement extends DummyData
             implements IReadOnlyTreeElement {

   public DummyDataTreeElement(String t) {
      super(t);
   }
   public List getCommands() {
      // as before in the Command Factory,
      // passing this as data object
   }

   public Cmd getDoubleClickCommand() {
      return null;
   }
   public Image getIcon() {
      return null;
   }

   public List getChildren() {
      if (children == null) {
         children = new ArrayList();
         addChild(new DummyDataTreeElement(text + "Childl"));
         addChild(new DummyDataTreeElement(text + "Child2"));
         addChild(new DummyDataTreeElement(text + "Child3"));
         addChild(new DummyDataTreeElement(text + "Child4"));
```

```
      }
      return children;
    }
  }
```

Der Nachteil dieses Ansatzes ist natürlich der, dass jetzt die *DummyData* Klasse angefasst (bzw. eine Unterklasse erstellt) werden muss … Dieses Problem könnte durch Verwendung von Objektadaptern vermieden werden.

Verändern der Baumstruktur

Readonly-Bäume sind nützlich, aber in den meisten Fällen ist es nötig, Bäume verändern zu können indem Kinder angefügt, verschoben oder gelöscht werden. Dazu gibt es im GUI meist zwei Möglichkeiten: Cut&Paste und Drag&Drop. Im Endeffekt führen diese Operationen (oder „Gesten“) aber zu den gleichen atomaren Operationen auf dem Baum.

Deshalb trennt das Treeframework die Implementation der atomaren Operation von den GUI-Gesten die diese anstoßen.

Des Weiteren ist zu beachten, dass Änderungen an der Baum*struktur* natürlich in der visuellen Repräsentation des Baumes sichtbar sein müssen – Datenmodell und View müssen synchronisiert werden, indem der Baum informiert wird wenn sich die Struktur ändert.

Als erstes Beispiel wollen wir uns das löschen eines Baumknotens anschauen. Dazu muss es auf der *DummyData* Klasse eine *delete()* Operation geben, welche den aktuellen Baumknoten löscht, ihn also aus der Liste der Kinder seines Parent-Objektes austrägt.

Da, wie bereits erwähnt, diese Dinge mit dem Baum synchronisiert werden müssen, muss dieses Lösch-Verhalten in den *DataAdapter* eingebaut werden. Dazu überschreiben wir die Operation *removeChildInternal():*

```
public class ExampleDataAdapter extends DataAdapter {
  protected void removeChildInternal(Object child)
            throws TreeException {
    DummyData d = (DummyData) child;
    d.delete();
  }
  // rest as before
}
```

Um einen Knoten nun wirklich zu löschen (z.B. aus einem Kommando heraus), muss die Operation *removeChild()* auf dem *DataAdapter* aufgerufen werden (nicht das oben implementierte *removeChildInternal()*). *removeChild()* ruft zunächst ihrerseits *removeChildInternal()* auf, und benachrichtigt dann denn Baum. Würde direkt *removeChildInternal()* aufgerufen werden, würde die Baumaktualisierung nicht funktionieren.

Das Löschen eines Baumelements kann nun zum Beispiel aus einem Kommando heraus passieren – die *execute()*-Operation ist im folgenden Abschnitt gezeigt.

```
public void execute() {
   adapter.removeChild(currentDataObject);
}
```

Die Frage die noch bleibt, ist: Wie kommt das Kommando an den korrekten *DataAdapter* den es benötigt, um *removeChild()* darauf aufzurufen. Man könnte sich den Adapter beispielsweise manuell setzen (in der *CommandFactory*). Eine Alternative besteht allerdings darin, dass Kommando das Interface *TreeCmd* implementieren zu lassen, oder einfach von der Klasse *DefaultTreeCmd* (statt *Cmd*) zu erben. Das Framework setzt dann den Adapter automatisch; durch den Aufruf von *getAdapter()* innerhalb des Commands kann auf ihn zugegriffen werden.

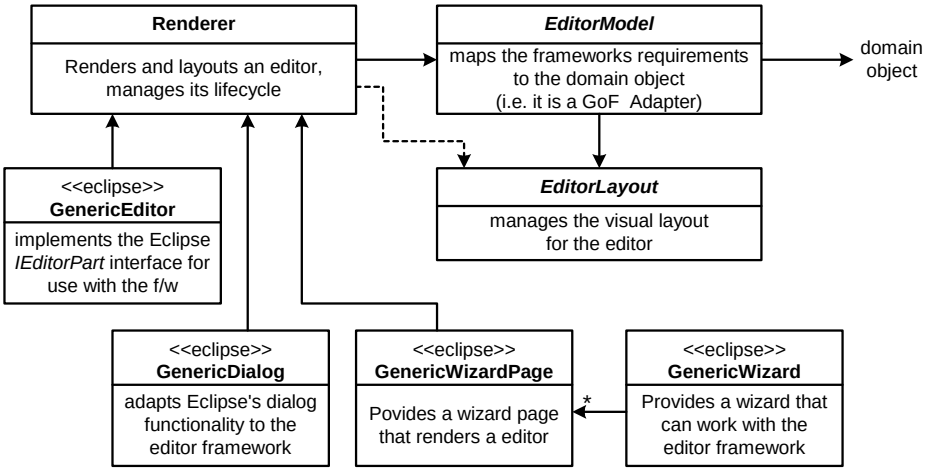
```
public class DeleteCmd extends DefaultTreeCmd {
   private DummyData currentDataObject;

   public DeleteCmd(DummyData d) {
      super("Delete", "");
      currentDataObject = d;
   }

   public void execute() {
      getAdapter().removeChild(currentDataObject);
   }
}
```

1.4.2 Editoren

Wie bereits im einleitenden Beispiel gesehen, arbeiten Editoren nach demselben Prinzip wie der Baum: Trennung von Darstellung und Datenmodell. Um genauer zu verstehen was passiert, hier zunächst die Klassenstuktur des Editorframeworks.

Struktur des Editorframeworks

Der *Renderer* ist – wie der *TreeRenderer* beim Baum – zuständig für die Darstellung und Ausführung des Editors. Er verwendet ein *EditorModel* welches dem Editor Zugriff auf die Daten erlaubt. Das *EditorModel* hat eine Referenz zu einem *EditorLayout*, welches sich um das Layouting des Editors kümmert (man muss nicht unbedingt etwas angeben; es gibt einen sinnvollen Default). Die *EditorModels* verwenden so genannte *EditorWidgets* um die Eingaben zu handeln – siehe unten.



Editoren können sowohl in Eclipse Editoren als auch in Eclipse Views, Dialogen und Wizards dargestellt werden, daher die verschiedenen *Generic…* Klassen.

Wie ein Editor prinzipiell funktioniert, wurde ja bereits in der Einleitung erläutert. Wir wollen hier also nur noch auf einige weitere Features eingehen.

EditorWidgets

Zunächst ein Blick auf *EditorWidgets*. *EditorWidgets* sind prinzipiell Wrapper um SWT Widgets (wie Textfelder, Comboboxen, Checkboxen, etc.). Im Gegensatz zu den SWT Widgets haben *EditorWidgets* alle eine einheitliche Schnittstelle. Diese umfasst prinzipiell das Setzen und Lesen des Inhalts, das Anzeigen einer Fehlermeldung (für den Fall dass der Inhalt eines Widgets ungültig ist), sowie die Aktivierung/Deaktivierung. Aufgrund der Tatsache, dass sie alle dieselbe Schnittstelle besitzen, kann das Editorframework die Widgets einheitlich behandeln. Eigene, anwendungsspezifische EditorWidgets können leicht hinzugefügt werden.

Ein Blick auf das *TextWidget*, das einfachste *EditorWidget* im Framework, erläutert die Funktionsweise. Zunächst einmal enthält das *TextWidget* das SWT *Text* Objekt, welches das eigentliche Widget darstellt.

```
public class TextWidget extends EditorWidget {
    private Text text = null;
```

Weiterhin besitzt jedes dieser Widgets eine Operation *createControl()* welche das eigentliche SWT Widget in das betreffende *parent* Composite einfügt.

```
    public Control createControl(Composite parent,
                                 Object layoutData) {
        final EditorWidget theWidget = this;
        text = new Text(parent,SWT.BORDER);
```

```
            text.setLayoutData(layoutData);
            // add a couple of listeners that
            // notify the editor of important events
            return text;
        }
```

Widgets besitzen alle eine Operation *setContent()*, die vom Framework verwendet wird um den Inhalt eines Widgets (den das EditorModel liefert) zu setzen. Dargestellt wird hier ein String; das originale *Object* wird mitgespeichert. *getContent()* liefert den Inhalt des *EditorWidgets*.

```
        public void setContent(Object o) {
            setRawContent(o);
            if (o != null) text.setText(o.toString());
        }

        public Object getContent() {
            return text.getText();
        }
```

setErrorInfo() wird vom Framework aufgerufen, wenn eine Fehlermeldung angezeigt werden muss. Meist wird ein Tooltip mit dem Text der Fehlermeldung angezeigt, und das Widget bekommt einen gelben Hintergrund. *defaultHandleErrorInfo()* erledigt dies.

```
        public void setErrorInfo(WidgetErrorInfo e) {
            defaultHandleErrorInfo( e, text );
        }
    }
```

Andere Widgets bieten dieselben öffentlichen Operationen, implementieren diese aber anders, je nach verwendetem SWT Widget.

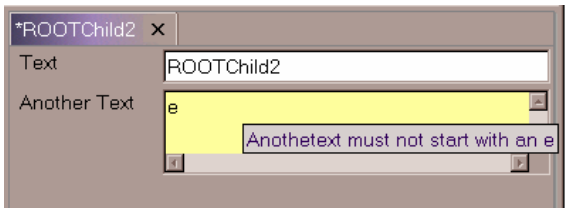
Feldinhalt im EditorModel überprüfen

Um den Feldinhalt zu überprüfen gibt es zwei Möglichkeiten: Zum einen kann die Operation *checkContent()* im *EditorModel* implementiert werden. Zum anderen können *CheckableWidgets* verwendet werden. Zunächst zur ersten Alternative.

Das Framework ruft für jedes Feld die Operation *checkContent()* auf. Ist der Feldinhalt ungültig, so muss dort eine *FieldContentNotValidException* geworfen werden.

```
public class SimpleEditorModel extends EditorModel {
   private final int ANOTHERTEXT = 0;
   public void checkContent(int index, EditorWidget widget)
               throws FieldContentNotValidException {
      switch (index) {
         case ANOTHERTEXT: {
            if (widget.getContentAsString().startsWith("e"))
                throw new FieldContentNotValidException(
                 "Anothertext must not start with an e");
            }
      }
   }
   // rest as before
}
```

Wie im folgenden Screenshot ersichtlich, wird das betreffende, fehlerhafte Feld dann gelb hinterlegt und die Fehlermeldung (die *Message* der Exception) wird als Tooltip über dem Widget angezeigt.



Zusätzlich zu der Definition der Überprüfung mittels *checkContent()* muss auch noch definiert werden, wann die Überprüfung passieren soll. Dazu gibt es drei Möglichkeiten: bevor der Editor gespeichert wird, nach Verlassen eines Feldes oder nach jeder Veränderung des Inhalts. Die *ValidationStrategy* dient dazu, dies festzulegen. Diese kann im Rahmen der *initialize()* Operation des *EditorModels* gesetzt werden.

```
public void initialize() {
   // whenever a widget changed
   setValidationStrategy(ValidationStrategy.WIDGET_CHANGED);
   // leaving a widget
   setValidationStrategy(ValidationStrategy.WIDGET_LOSE_FOCUS );
   // just before saving
   setValidationStrategy(ValidationStrategy.EDITOR_SAVE);
}
```

Dies ist übrigens eine Anwendung des Strategy Musters.

Checkable Widgets

Das Abprüfen von Inhalt im *EditorModel* (wie oben gezeigt) sollte immer dann verwendet werden, wenn die Überprüfung stark anwendungsabhängig ist. Gilt es jedoch allgemeinere Dinge abzuprüfen (ganze Zahlen, Postleitzahlenformate, etc), so ist dieses Vorgehen lästig, und eine effizientere Möglichkeit ist nötig. Dazu gibt's die *CheckableWidgets*. Dies sind *EditorWidgets*, die zusätzlich das Interface *CheckableWidget* implementieren. Das folgende ist ein Beispiel für ein *IntegerTextWidget* – ein *TextWidget* welches nur ganze Zahlen als Eingabe erlaubt.

```
public class IntegerTextWidget extends TextWidget
             implements CheckableWidget {
   public void checkContent()
            throws FieldContentNotValidException {
      String s = getContentAsString();
      try {
         Integer.parseInt(s);
      } catch (NumberFormatException e) {
         throw new FieldContentNotValidException(e.getMessage());
      }
   }
}
```

Natürlich ließe sich dieser Checkmechanismus auch leicht mittels einer Strategy parametrierbar machen.

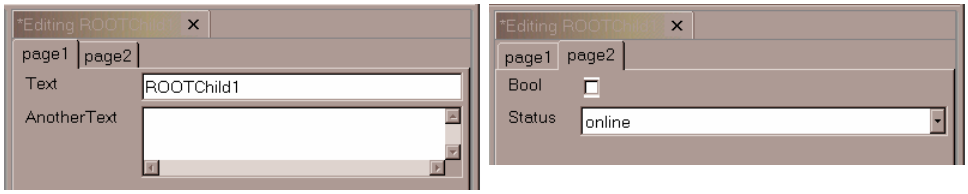
Ein weiteres Beispiel wäre das folgende Widget, welches *Floats* akzeptiert. Man beachte: Wenn man Komma statt Punkt eingibt, wird die automatisch korrigiert.

```
public class FloatTextWidget extends TextWidget
                             implements CheckableWidget {
   public void checkContent()
               throws FieldContentNotValidException {
      String s = getContentAsString();
      // check if there is a comma instead of a dot....
      if (s.indexOf(",") > 0) {
         s = s.replace(',', '.');
         // set the content
         setContent(s);
      }
      //check that it is a valid number
      try {
         Float.parseFloat(s);
      } catch (NumberFormatException e) {
```

```
            throw new FieldContentNotValidException(e.getMessage());
         }
      }
   }
```

Multiple Panes

Die Widgets eines *EditorModels* können auf mehrere Tabs einer TabbedPane verteilt werden, wie die folgende Abbildung zeigt.



Dies ist sehr einfach zu erreichen. Man muss lediglich den Titel der Seite auf die ein Widget platziert werden soll in *getPaneTitle()* zurückgeben.

```
public class SimpleEditorModel extends EditorModel {
   public String getPaneTitle(int index) {
      switch (index) {
         case TEXT :
         case AHOTHERTEXT :
            return "page1;";
         default :
            return "page 2 ";
      }
   }
   // rest as before
}
```

Man beachte dass dieses Feature vor allem deshalb so einfach zu implementieren ist, weil das *EditorModel* nichts vom konkreten Layout seiner *EditorWidgets* weiß.

Layouts

Das Layout des Editors kann unabhängig von seinem Inhalt variiert werden. Im Rahmen der *initialize()* Operation kann einfach ein anderes *EditorLayout* Objekt gesetzt werden.

```
public void initialize() {
   setLayout(new DefaultEditcrLaycut(true);
}
```

Das Framework kommt mit einigen verschiedenen Layouts – eigene können selbstverständlich auch implementiert werden. *EditorLayouts* kapselt im Prinzip einfach die SWT Layouts und liefern für die Labels und Widgets im Editor einfach das *LayoutData*.

```
public abstract class EditorLayout {
   public abstract Layout getLayout();
   public abstract Object getLabelLayoutData(int i);
   public abstract Object getWidgetLayoutData(int i, int max,
         boolean hasLabel, SizeHint hint, boolean grabAlways);
}
```

getLayout() liefert das zu verwendende SWT Layout zurück.

Abhängigkeiten zwischen den Widgets

Widgets können – vor allem auch abhängig von dem Zustand des zugrunde liegenden Datenobjektes aktiviert oder deaktiviert werden. Beispielsweise könnten wir das Altersfeld deaktivieren wenn das Geschlecht der Person weiblich ist.

```
public class PersonEditorModel extends EditorModel {
   public boolean isEnabled(int index) {
      switch (index) {
         case AGE :
            return getWidget(SEX).getContentAsEtring().
                                 equals("male");
         default : return true;
      }
   }
}
```

Auch kann der Inhalt von Widgets verändert werden, abhängig von Inhalt anderer Felder. Zu beachten ist dabei die Verwendung von *ensureContent()* statt *setContent()* um ewige Rekursion zu vermeiden.

```
public class PersonEditorModel extends EditorModel {
   public void updateDependencies() {
      if (getWidget(FIRST).getContentAsString().
                           equals("Markus")) {
         getWidget(NAME).ensureContent("Voelter");
      }
   }
}
```

1.5 Warum das alles? Vorteile.

Wir haben jetzt recht ausführlich verschiedene Features des Frameworks gezeigt. Warum sollte man nun mittels derartiger Frameworks Oberflächen entwickeln? Dieser Abschnitt soll auf einige „angenehmen Nebeneffekte“ der frameworkbasierten GUI Entwicklung eingehen.

1.5.1 Testen von GUIs

Im Rahmen des Testens von GUIs lassen sich drei verschiedene Aspekte unterscheiden:

- **Ergonomie:** Sitzen Knöpfe an der richtigen Stelle, sind Layouts angemessen, sind Eingabefelder groß genug?
- **„Logik“ der Oberfläche:** Werden Daten aus den Editoren in die Datenobjekte korrekt übernommen? Werden die Daten korrekt validiert? Werden die Abhängigkeiten unter den Feldern (Aktivierung/Deaktivierung) korrekt gehandhabt?
- **Grundfunktionalität:** Speichern funktioniert nach Drücken des Knopfes, Widgets funktionieren, etc.

Das Framework erleichtert das Testen der verschiedenen Aspekte in unterschiedlichem Maße.

Testen der Ergonomie

Das Testen der Ergonomie wird nicht unterstützt – es lässt sich auch schwer automatisieren. Durch die immer gleiche Anordnung von GUI Elementen und die Wiederverwendbarkeit von Layouts unabhängig von den Editordaten lässt sich der Aufwand allerdings deutlich reduzieren.

Testen der Grundfunktionalität

Die Grundfunktionalität des Frameworks an sich muss ganz normal getestet werden. Für die Widgets gilt das gleiche. Dadurch dass Widgets im Rahmen eines Projektes ja aber systematisch wieder verwendet werden, hält sich der Testaufwand in Grenzen.

Testen der GUI Logik

Bei richtiger Verwendung des Frameworks wird ein Grossteil der Entwicklungsarbeit in genau diesen Aspekt fließen. Daher bietet das Framework hier Unterstützung beim Test an. Der Test wird mittels JUnit durchgeführt.

Zunächst wird ein TestCase definiert, in dem ein *Person*-Objekt angelegt wird deren Name *Bernd* sein sol. Dann wird ein passendes *EditorModel* sowie ein sog. *EditorTester* angelegt. Der *EditorTester* ist eine Unterklasse von *Renderer* und besitzt einige nützliche Zusatzfunktionen die das Testen erleichtern.

```
public class SimpleTest extends TestCase {
   private EditorTester tester = null;
   private PersonEditorModel em;
   private Person person;
   protected void setUp() throws Exception {
      person = new Person();
      person.setFirstlJame("Bernd ");
      em = new PersonEditorModel(person);
      tester = new EditorTester(em);
   }
}
```

Man kann nun beispielsweise validieren, dass sich der Inhalt der Widgets nach der Speicherung auch wirklich im Domänenobjekt (hier: der Person) widerspiegelt.

```
public void testSomething() {
   tester.setupEditor();
   assertEquals(tester.getWidget(PersanEditorModel.FIRST).
                      getContent(), "Bernd");
   tester.getWidget(PersonEditorModel.FIRST).setContent("XYZ");
   tester.save();
   assertEquals(person.getFirstName(), "XYZ");
}
```

1.5.2 Berechtigungen

Dadurch dass alle GUI-Darstellungen durch das Framework passieren ist es recht einfach, das GUI dynamisch anzupassen. Ein sehr schönes Beispiel dafür sind Berechtigungen. Angenommen, verschiedene Benutzergruppen dürfen nur bestimmte Aspekte der GUI benutzen, zum Beispiel:

- Bestimme Eingabefelder dürfen nicht gelesen oder geschrieben werden.
- Bestimme Eingabefelder dürfen nur gelesen werden.
- Bestimmte Kommandos dürfen nicht ausgeführt werden
- Bestimmte Äste von Bäumen dürfen nicht ausgeklappt werden.

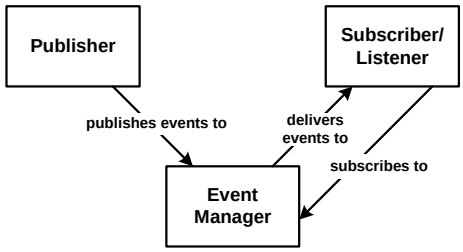
Entsprechende Funktionalität lässt sich sehr leicht einführen (das Framework enthält dies auch), da nur an sehr wenigen Stellen etwas getan werden muss:

- Kommandos werden nur an einigen wenigen Stellen (Button-Renderer, Kontextmenü-Renderer, …) ausgeführt.
- Editorfelder werden immer unter Kontrolle des *Renderers* dargestellt und bearbeitet
- Bäume (speziell das Aufklappen) passiert grundsätzlich unter der Kontrolle des *TreeRenderers*.

Die genaue Funktionsweise des Berechtigungsmanagement kann ihr aufgrund von Platzbeschränkungen nicht erläutert werden.

1.5.3 Events

Wie praktisch jedes GUI kommt auch das hier vorgestellte Framework nicht komplett ohne Events aus. Üblicherweise wird zur Implementierung von Events ja das Observer Pattern verwendet. Dies hat allerdings den Nachteil, dass alle Subjekte ihre Observer verwalten müssen. Um dies zu vereinfachen, verwendet das hier vorgestellte Framework einen zentralen *EventManager*.



Events haben einen Typ, eine Quelle und eine Nutzlast (ein beliebiges Objekt, welches einfach mit transportiert wird). Events können von jedem beliebigen Objekt (der Quelle) erzeugt und publiziert werden. Empfänger müssen das Interface *EventReceiver* implementieren.

Der Erzeuger eines Events erzeugt dies und publiziert es zum *EventManager*. Dieser transportiert das Event an alle *EventReceiver* die sich vorher dort für das Event angemeldet haben. Diese Anmeldung kann basierend auf dem Eventtyp oder basierend auf der Quelle passieren.

Als Beispiel soll die Aktualisierung des Baumes dienen – nachdem der Titel eines *DummyData* Objektes im Editor geändert wurde.

Dazu definieren wir zunächst einen entsprechenden Eventtyp.

```
public class SimpleEditorModel extends EditorModel {

   public static EventType DUMMYCHANGED =
        new EventType("DummyData changed!");
   // as before.
}
```

Um das Event zu erzeugen, können wir im Editor die Operation *afterSave()* überschreiben. Diese wird aufgerufen, wann immer der Speicher-Button gedrückt und die Daten erfolgreich ins Datenobjekt durchgeschrieben wurden. In dieser Operation, wird nun ein Event vom oben definierten Typ erzeugt.

```
public void afterSave() throws SaveVetoException {
   EventManager.publish(new Event(DUMMYCHANGED,
                        getObject(), null));
}
```

Der zweite Parameter (der hier mit *null* definiert ist) ist die Nutzlast des Objektes. Als Quelle des Events wird hier (mittels *getObject()* im EditorModel) das aktuell editierte *DummyData* Objekt mitgegeben. Dies geschieht deshalb, weil sich der *EventManagerEventAdapter* mittels dieses Datenobjektes für alle Events die von dem Objekt abstammen anmeldet und dann, bei passendem Eventtyp, den Baum aktualisiert (*changeTreeItem()*).

```
public class EventManagerEventAdapter
             extends EventAdapter
             implements EventReceiver {

   public EventManagerEvent Adapter(EventType eventType) {
      this.eventTypePassedInConstructor = eventType;
   }

   public void registerDataObject(Object dataObject) {
      EventManager.subscribeBySrc(dataObject, this);
   }

   public void receiveEvent(Event e) {
      if (eventTypePassedInConstructor != null) {
         if (e.isType(eventTypePassedInConstructor)) {
            adapter.changeTreeItem(e.getSrc);
         } else if (eventGroup != null) {
            if (e.isType(eventGroup)) {
                  adapter.changeTreeItem(e.getSrc());
            }
         }
      }
   }
}
```

1.6 Das Framework und die Eclipse Philosophie

Zentrale Philosophie bei Eclipse ist, dass alle Plugins die entwickelt werden selbst wieder erweiterbar sind, indem Extension Points definiert werden. Das hier vorgestellte Framework unterstützt diese Philosophie nicht explizit, allerdings

- bietet das Framework eine Bibliothek zum Bauen von Oberflächen. Die Erweiterbarkeit sollte auf Anwendungsebene liegen (also in den Anwendungen, die das hier vorgestellte Framework benuzten),
- lassen sich jederzeit Extension Points definieren und auf Anwendungsebene verwenden.

Das Framework verbietet dies nicht.

1.7 Zusammenfassung

Ich hoffe das aktuelle Kapitel konnte ein wenig Lust machen auf frameworkbasierte GUI Entwicklung. Wenn man einmal ein solches Framework zur Verfügung hat, wird das Entwickeln von GUIs deutlich effizienter. Das Beispielframework besteht aus ca. 160 Klassen, ist also durchaus überschaubar. Die Entwicklung eines für die betreffende Art von GUIs passenden Frameworks im Rahmen eines Projektes rechnet sich durchaus.