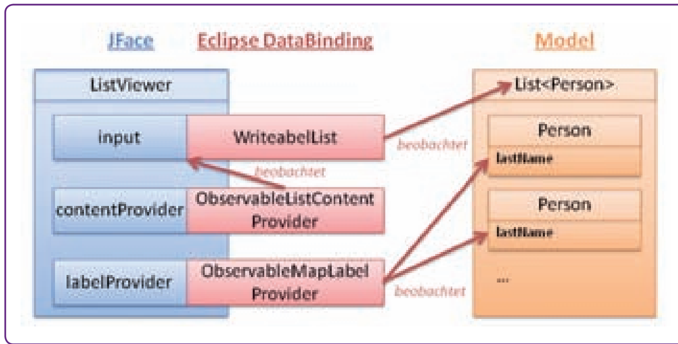

Abb. 1: ListViewer

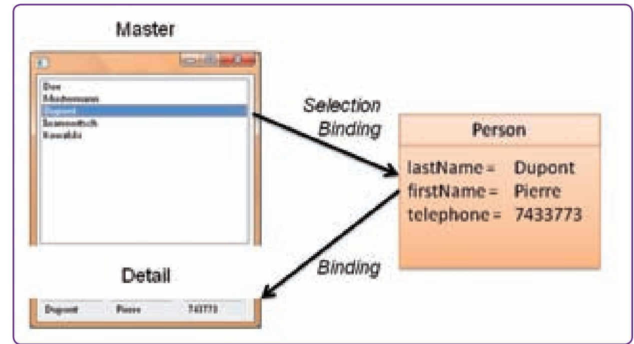

Abb. 2: Master-Detail

fachte Anbindung eines Datenmodells. Eclipse DataBinding erweitert dieses Konzept durch verschiedene Adaptoren für einen vereinheitlichten Zugriff auf das Datenmodell. Am einfachsten kann anhand des List Widgets die Verwendung von JFace-Viewern und Eclipse DataBinding veranschaulicht werden. In einem kleinen Beispiel sollen die Namen von Personen in einer Liste angezeigt werden. Als Modell liegt eine Liste mit Instanzen der Klasse *Person* vor:

```
public class Person {
 private String lastName;
 public Person(String lastName) {
  this.lastName = lastName;
 }

 public void setLastName(String lastName) {
  this.lastName = lastName;
 }

 public String getLastName() {
  return lastName;
 }

 @Override
 public String toString() {
  return getLastName();
 }

}
```

Für Listen bietet das Eclipse DataBinding bereits einen passenden ContentProvider: *ObservableListContentProvider*. Um ihn verwenden zu können, muss dem ListViewer von JFace eine Liste übergeben werden, die *IObservableCollection* implementiert. Wir entscheiden uns für eine *WritableList*, die eine Liste von Personen kapselt. Der Mehrwert der Kapselung besteht darin, dass eine *IObservableCollection* über das Observer Pattern überwacht werden kann und bei Änderungen Events feuert. Das Ganze sieht im Code dann wie folgt aus:

```
ListViewer viewer = new ListViewer(shell);
viewer.setContentProvider
            (new ObservableListContentProvider());
List<Person> persons = createPersonsList();
WritableList input = new WritableList
                        (persons, Person.class)
viewer.setInput(input);
```

Änderungen an der *WritableList* werden auf diese Weise sofort in dem UI reflektiert. Der *ObservableListContentProvider* überwacht die Liste und leitet Änderungen an das Widget weiter. Hierfür wird der Quellcode wie folgt erweitert. Dem GUI wird ein Button hinzugefügt, der das selektierte Element aus der Personenliste löscht:

```
Button removeButton = new Button(shell, SWT.PUSH);
removeButton.setText("Remove");
removeButton.addSelectionListener
                    (new SelectionAdapter() {
 @Override
 public void widgetSelected(SelectionEvent e) {
  if (viewer.getSelection().isEmpty()) {
   return;
  }
  Person p = (Person) ((IStructuredSelection) viewer
    .getSelection()).getFirstElement();
 input.remove(p);
 }
});
```

Nach dem Löschen des selektierten Elements wird dieses nicht mehr in der UI-Liste angezeigt: Die Daten wurden von dem Modell zum User Interface synchronisiert.

Eine Person wird in der Liste als String gerendert, der den Nachnamen repräsentiert. Der standardmäßige LabelProvider des ListViewer ermittelt den angezeigten Wert über die *toString*-Methode des Modellobjekts. In der Klasse *Person* liefert *toString* den Nachnamen.

Um eine Änderung innerhalb eines Personenobjekts automatisch in dem List Widget zu reflektieren, ist ein weiterer Schritt notwendig. Das Modellobjekt muss eine Schnittstelle anbieten, die es erlaubt, dessen Properties zu überwachen. Nur auf diese Weise besteht die Möglichkeit, Änderungen direkt weiterzugeben. Anschließend kann sich ein spezieller LabelProvider über diesen Mechanismus an dem Modellobjekt registrieren und die Wertänderungen an das User Interface weitergeben. Die Vorgehensweise ist der Überwachung der *WritableList* sehr ähnlich. Exemplarisch erweitern wir nun den Code wie folgt: In unserem Beispiel wird zunächst ein Button hinzugefügt, der den Nachnamen komplett in Groß- bzw. Kleinbuchstaben umwandelt:

```
Button changeButton = new Button(shell, SWT.PUSH);
changeButton.setText("Change to upper/lower case");
changeButton.addSelectionListener
                    (new SelectionAdapter() {
```

Wo gibt´s mehr zu Eclipse DataBinding?

In dieser Ausgabe des Eclipse Magazins:

„EMF und SWT: Datenmodell und UI effizient verheiratet!" von Matthias Heinrich und Henrik Lochmann.

Auf der Heft-CD und auf JAXenter.de:

„Daten und ihre Bindungen" von Ludwig Mittermeier: www.jaxenter.de/artikel/1353, „Eclipse Forms im Härtetest" von Marco van Meegen: www.jaxenter.de/artikel/1816.

Anderswo: „Eclipse DataBinding + Validation + Decoration" von Kai Tödter: http://eclipse.dzone.com/articles/eclipse-databinding-validation, „Eclipse DataBinding with Eclipse RCP Applications – Tutorial" von Lars Vogel: http://www.vogella.de/articles/EclipseDataBinding/article.html

```
  @Override
  public void widgetSelected(SelectionEvent e) {
   if (viewer.getSelection().isEmpty()) {
    return;
   }
   Person p = (Person) ((IStructuredSelection) viewer
      .getSelection()).getFirstElement();
   if (p.getLastName().toUpperCase().equals
                                   (p.getLastName())) {
    p.setLastName(p.getLastName().toLowerCase());
   } else {
    p.setLastName(p.getLastName().toUpperCase());
   }
  }
});
```

Im nächsten Schritt muss die Klasse *Person* angepasst werden. Diese Klasse muss um PropertyChangeSupport erweitert werden. Außerdem wird ein LabelProvider benötig, der sich an einem Objekt als Observer registriert und Events weiterleitet. Eclipse DataBinding stellt hierfür die Klasse *ObservableMapLabelProvider* zur Verfügung. Der Konstruktor der Klasse erwartet als Parameter ein *IObservableMap*. In dieser Map steht, welches Attribut welcher Elemente bezüglich Änderungen beobachtet werden soll. In unserem Beispiel wird das Attribut *lastName* der Person überwacht. Wir erstellen die *IObservableMap* mithilfe der von *BeansObservables* angebotenen Methode *observeMap*:

```
ListViewer viewer = new ListViewer(shell);
ObservableListContentProvider contentProvider =
                      new ObservableListContentProvider()
viewer.setContentProvider(contentProvider);
IObservableMap observableMap =
                             BeansObservables.observeMap(
    contentProvider.getKnownElements(),
    Person.class, "lastName");
viewer.setLabelProvider(new ObservableMapLabelProvider
                                        (observableMap));
List<Person> persons = createPersonsList();
WritableList input = new WritableList(persons,
                                            Person.class)
viewer.setInput(input);
```

Die Methode *observeMap* nimmt nicht entsprechend unseres Modells eine Liste entgegen, sondern ein *IObservableSet*. Glücklicherweise bekommt der Java-Entwickler dieses Set vom *ObservableListContentProvider* geliefert. Wenn wir nun unser Beispiel starten, sehen wir, dass die Änderung der Groß- bzw. Kleinschreibung sofort in der UI-Liste angezeigt wird.

Master-Detail

Häufig findet man in UIs das folgende Szenario: Im UI gibt es eine Liste, Tabelle oder einen Baum, wo ein Element selektiert werden kann. Entsprechend der Selektion sollen detailliertere Informationen angezeigt und möglicherweise bearbeitet werden. Beispielsweise kann es sich bei dem selektierten Element um eine Person handeln, und die editierbaren Details sind neben dem Namen auch die Telefonnummer oder das Geburtsdatum. Für solche Szenarien hat sich der Begriff „Master-Detail" durchgesetzt.

Wir wollen unser Beispiel erweitern, um die Unterstützung von Eclipse DataBinding bezüglich Master-Details zu veranschaulichen. Die Klasse *Person* erhält zwei neue Properties: *firstName* und *telephoneNumber*. Damit die Details im GUI angezeigt werden können, fügen wir drei Textfelder hinzu:

```
Text lastName = new Text(shell, SWT.BORDER);
lastName.setEditable(false);
Text firstName = new Text(shell, SWT.BORDER);
firstName.setEditable(false);
Text telephoneNumber = new Text(shell, SWT.BORDER);
telephoneNumber.setEditable(false);
```

Jetzt müssen die Details mit der Auswahl in der Liste verbunden werden. Drei Schritte sind notwendig. Zuerst muss die Selektion in der Liste beobachtet werden (*ViewersObservables.observeSingleSelection*), dann das Property (Detail) in der Selektion (*BeansObservables.observeDetailValue*), und zuletzt muss das Detail mit dem Textfeld gebunden werden. Der zweite und der dritte Schritt wiederholen sich für die beiden anderen Properties:

```
DataBindingContext dbc = new DataBindingContext();
// 1. Observe changes in selection.
IObservableValue selection = ViewersObservables
.observeSingleSelection(viewer);
// 2. Observe the property of the current selection.
IObservableValue detailObservable = BeansObservables
 .observeDetailValue(Realm.getDefault(), selection,
                                "lastName", String.class);
// 3. Bind the Text widget to the detail
dbc.bindValue(SWTObservables.observeText(
 lastName, SWT.None), detailObservable, null, null);
```

Wird in der Liste nun eine Person ausgewählt, werden in den Textfeldern die dazugehörigen Details angezeigt.

Anzeige

TableViewer

Im Beispiel soll anstelle einer Liste eine Tabelle verwendet werden. Der *ListViewer* wird einfach durch den *TableViewer* ersetzt. Auf den ersten Blick ist auch der Unterschied im GUI minimal. Wollen wir aber mehr als eine Spalte anzeigen, benötigen wir für den LabelProvider mehr als nur eine Map. Der *ObservableMapLabelProvider* kann auch ein Array von Maps entgegennehmen, und mit *BeansObservables.observeMaps* kann man dieses Array sehr einfach erstellen lassen. Das sieht im Quellcode so aus:

```
IObservableMap[] observableMaps = BeansObservables.
                                        observeMaps(
 peopleViewerContentProvider.getKnownElements(),
 Person.class, new String[] { "lastName", "firstName",
                                "telephoneNumber" });
viewer.setLabelProvider(new ObservableMapLabelProvide
                                  r(observableMaps));
```

Damit alle Spalten sichtbar sind, müssen sie auch erzeugt werden:

```
TableViewerColumn columnLastName =
            new TableViewerColumn(viewer, SWT.NONE);
columnLastName.getColumn().setText("Last Name");
columnLastName.getColumn().setWidth(80);
TableViewerColumn columnFirstName = new
TableViewerColumn(viewer, SWT.NONE);
columnFirstName.getColumn().setText("First Name");
columnFirstName.getColumn().setWidth(80);
TableViewerColumn columnTelephone =
            new TableViewerColumn(viewer, SWT.NONE);
columnTelephone.getColumn().setText
                              ("Telephone Number");
columnTelephone.getColumn().setWidth(100);
```

Dank Eclipse DataBinding bleibt die komplette Funktionalität des Beispielprogramms auch mit der Tabelle erhalten. Etwas aufwändiger wird es, wenn die Zellen der Tabellen editierbar werden sollen. Jede Spalte muss einer Klasse (*EditingSupport*) übergeben werden, die den Editor liefert. Eclipse DataBinding stellt die abstrakte Klasse *ObservableValueEditingSupport* bereit, um Zelleditor und DataBinding zusammenzuführen. Dem Java-Entwickler obliegt es, den Konstruktor und drei Methoden zu implementieren. Dem Konstruktor muss der Viewer für die Spalten und der aktuelle Kontext des DataBindings übergeben werden. Im Beispiel übergeben wird noch den Namen der Property. Im Konstruktor wird der Editor für die Zelle erzeugt. Für die Strings unserer Modellklasse bietet sich ein Editor an, der als Control ein Textfeld liefert (*TextCellEditor)*. Die Methode *doCreateCellEditorObservable* sorgt dafür, dass Änderungen im Textfeld beobachtet und gemeldet werden, und *doCreateElementObservable* tut dies für die Property unseres Modells. *getCellEditor* liefert immer den Editor, den wir bereits im Konstruktor erstellt haben. Somit sieht unsere Implementierung wie in Listing 1 aus.

Den folgenden Editor benötigen wir jetzt für jede Spalte:

```
columnLastName.setEditingSupport(new ViewerEditor
                Support(viewer, dbc, "lastName"));
```

TreeViewer

Erst mit Eclipse 3.4 bietet das Eclipse DataBinding eine Unterstützung für Bäume im UI. Die Vorgehensweise beim Anbinden eines Modells ist der für Tabellen sehr ähnlich. Der Hauptunterschied ergibt sich aus der Struktur des Datenmodells. Als Modell liegt nicht, wie in einer Tabelle, eine Liste vor, sondern ein Baum. Um

Listing 1

```
private static class ViewerEditorSupport extends
  ObservableValueEditingSupport {

 private CellEditor cellEditor;
 private String property;

 public ViewerEditorSupport(ColumnViewer
                 viewer, DataBindingContext dbc,
   String property) {
  super(viewer, dbc);
  cellEditor = new TextCellEditor((Composite)
                      viewer.getControl());
  this.property = property;
 }

 @Override
 protected IObservableValue doCreateCellEditor
                              Observable(
   CellEditor cellEditor) {
  return SWTObservables.observeText
                    (cellEditor.getControl(),
    SWT.FocusOut);
 }

 @Override
 protected IObservableValue doCreateElement
                 Observable(Object element,
   ViewerCell cell) {
  return BeansObservables.observeValue(element,
                                  property);
 }

 @Override
 protected CellEditor getCellEditor(Object element) {
  return cellEditor;
 }

}
```

Listing 2

```
IObservableValue firstNameOV_ui = SWTObservables.observeText(firstNameText, SWT.FocusOut);
firstNameOV_model = BeansObservables.observeValue(model, ”firstName“);
context.bindValue(firstNameOV_ui, firstNameOV_model, null, null);
IObservableValue lastNameOV_ui = SWTObservables.observeText(lastNameText, SWT.FocusOut);
lastNameOV_model = BeansObservables.observeValue(model, “lastName“);
context.bindValue(lastNameOV_ui, lastNameOV_model, null, null);
IObservableValue computed_ui = SWTObservables.observeText(computedNameText, SWT.Focu-
sOut);
IObservableValue computed_model = new FullNameObservable();
context.bindValue(computed_ui, computed_model, null, null);

....

private class FullNameObservable extends ComputedValue {

@Override
protected Object calculate() {
 String firstName = (String) firstNameOV_model.getValue();
 String lastName = (String) lastNameOV_model.getValue();
 StringBuilder builder = new StringBuilder();
 builder.append(lastName);
 if (lastName.length() > 0) {
  builder.append(”, ”);
 }
 builder.append(firstName);
 return builder.toString();
}
}
```

die für einen Baum übliche Parent-Child-Beziehung zu modellieren, erweitern wir die Klasse *Person* um die Property *children*, die eine Liste von Personen (*List<Person>*) hält. Der *TableViewer* wird durch *TreeViewer* ersetzt. Dem *TreeViewer* wird keine *WriteableList*, sondern das Wurzelelement des *TreeModels* als Input übergeben. Den *LabelProvider* können wir so belassen, aber da im Baum nur die erste Property angezeigt wird, verwenden wir wieder nur eine *IObservableMap*, und zwar die für *lastName*. Die größte Änderung betrifft den ContentProvider. Für einen Baum wird *ObservableListTreeContentProvider* eingesetzt. Dem Konstruktor wird zuerst eine Factory übergeben, die dafür sorgt, dass die Children jedes Knotens überwacht werden. Da die Children in unserem Beispiel in einer Liste abgespeichert sind, übergeben wir eine Factory, die eine solche beobachtet. Die wichtigsten Änderungen auf einen Blick sind:

```
IObservableFactory listFactory =
        BeansObservables.listFactory(Realm.getDefault(),
  "children", Person.class);

ObservableListTreeContentProvider peopleViewerContent
        Provider = new ObservableListTreeContentProvider(
  listFactory, null);
viewer.setContentProvider(peopleViewerContentProvider);
viewer.setInput(root);
```

Den Button REMOVE entfernen wir aus dem GUI, da unser Modell das Löschen von *Children* nicht unterstützt.

ComputedValue

Eine weitere hilfreiche Klasse von Eclipse DataBinding ist *ComputedValue*. Diese Komponente ist eine spezielle Ausprägung von *IObservableValue*. Der Wert eines *ComputedValues* leitet sich aus den Werten anderer *IObservableValues* ab. Die Ermittlung des Werts wird in der Methode *calculate* implementiert. Von hier aus können beliebige andere *IObservables* nach deren Werten gefragt werden, um die Parametermenge der Berechnung zu füllen. *Calculate* wird immer als Konsequenz von *getValue* aufgerufen. Diese Funktionalität lässt sich auf einfache Weise selbst implementieren. Wo liegt der Mehrwert des *ComputedValue*? Um diese Frage zu beantworten, ist es wichtig zu verstehen, zu welchen Zeitpunkten ein *ComputedValue* berechnet werden soll. Betrachten wir das Beispiel aus Listing 2: *ComputedValue* berechnet sich über das *IObservable* für den Nachnamen und das *IObservable* für den Vornamen. *getValue* liefert die entsprechenden Werte konkateniert zurück. *ComputedValue* ist gegen ein Textfeld (*computedNameText*) gebunden.

ComputedValue soll immer dann aktualisiert werden, wenn sich entweder der Vorname oder der Nachname ändert. Und das Überraschende ist, dass unser Beispiel genau diese Funktionalität schon anbietet. Wie wird *ComputedValue* darüber informiert, dass sich eine seiner Abhängigkeiten ändert, ohne dass wir einen entsprechenden Observer an diesen registriert haben? An dieser Stelle kommt der *ObservableTracker* ins Spiel. Mithilfe dieser Komponente ist es möglich, beliebigen Code auszuführen und dabei alle *IObservableValues* zu sammeln, deren *get*-Methode aufgerufen wurde. Als Hintergrundinformation sei noch darauf hingewiesen, dass die Klasse *AbstractObservableValue* im Code der *getValue*-Methode sich bei dem *Observ-ableTracker* automatisch

Anzeige

registriert. Genau diese Funktionalität macht sich der *ComputedValue* zu Nutze. Er ruft nach der Instanziierung die eigene *calculate*-Methode über den *ObservableTracker* auf und erhält implizit die Liste seiner Abhängigkeiten in Form von *IObservables*. Bei jedem dieser *IObservableValues* registriert er sich als Observer und kann auf Wertänderungen reagieren.

ComputedList

ComputedList ist eine weitere Komponente des Eclipse DataBindings, deren Werte sich aus anderen *IObservableValues* ergeben. Die Methode *calculate* liefert eine Liste von Werten zurück, deren Inhalt entsprechend der Abhängigkeiten ermittelt wird. Um die Auswirkungen zu veranschaulichen, kodieren wir folgendes kleines Szenario: Wir haben eine Combobox, in der die Namen von Personen angezeigt werden. Über zwei Checkboxen wird bestimmt, ob weibliche und/oder männliche Personen erscheinen sollen. Das UI besteht somit aus drei Komponenten:

```
men = new Button(shell, SWT.CHECK);
men.setText("Men");
women = new Button(shell, SWT.CHECK);
women.setText("Women");
Combo combo = new Combo(shell, SWT.DROP_DOWN |
                                    SWT.READ_ONLY);
viewer = new ComboViewer(combo);
```

Zur Berechnung des Inhalts der Combobox benötigen wir zwei *IObservableValue* für die beiden Checkboxen. Somit kann in der Methode *calculate* der Klasse *ComputedList* die Liste erzeugt werden (Listing 3). Wie bereits bei *ComputedValue* funktioniert dies ohne weitere Observer, da auch *ComputedList* vom *ObservableTracker* über die Änderungen informiert wird.

Listing 3

```
final IObservableValue womenObservable =
          SWTObservables.observeSelection(women);
final IObservableValue menObservable =
            SWTObservables.observeSelection(men);
viewer.setContentProvider(new
                 ObservableListContentProvider());
IObservableList filteredList = new ComputedList() {
 @Override
 protected List<Person> calculate() {
  List<Person> result = new ArrayList<Person>();
  for (Person person : persons) {
   if (((Boolean) womenObservable.getValue())
     && person.getGender() == Gender.FEMALE) {
    result.add(person);
   } else if (((Boolean) menObservable.getValue())
     && person.getGender() == Gender.MALE) {
    result.add(person);
   }
  }
  return result;
 }
};
viewer.setInput(filteredList);
```

DelayedObservableValue

Eine interessante Komponente ist das *DelayedObservableValue*. Diese Komponente wird im Bereich von Textfeldvalidierungen eingesetzt. *DelayedObservableValue* kapselt ein *ISWTObservableValue* und gibt dessen Events erst verzögert weiter. Nachfolgende gleichartige Events innerhalb der Verzögerungszeit überschreiben den Event. Auf diese Weise wird z. B. bei einer Bindung an ein Textfeld verhindert, dass nach jedem Tastendruck eine Validierung stattfindet. Erst wenn die Tastatur über die Dauer der Verzögerungszeit ruht, werden observierende Objekte informiert. Eine Ausnahme ist das Verlassen des SWT Widget. Dieser Fall wird entsprechend des Ablaufs der Verzögerung behandelt. Um von dem Mehrwert von *DelayedObservableValue* zu profitieren, ist es also wichtig, im Fall einer Textfeldüberwachung das zu kapselnde Observable mit dem Flag *SWT.MODIFY* zu erzeugen.

Realms

Innerhalb von Eclipse DataBinding spielen Realms – verborgen vor dem Entwickler – eine wichtige Rolle. Realms repräsentieren einen beliebigen Kontext, der zu einem Zeitpunkt aktiv oder inaktiv sein kann. Im Bereich von SWT entspricht der Kontext gewöhnlich dem User Interface Thread. Das zugehörige Realm ist *DisplayRealm*. Bevor eine Aktion innerhalb eines Observables ausgeführt wird, gibt es einen Aufruf von *checkRealm()*. Hierdurch wird sichergestellt, dass das aktuelle Realm gültig ist. In dem von SWT verwendeten *DisplayRealm* wird an diesem Punkt überprüft, ob der Code auf dem aktuellen User Interface Thread ausgeführt wird. Ist dies nicht der Fall, gibt es eine Exception. Auf diese Weise wird einer Inkonsistenz des User Interface vorgebeugt. Neben der reinen Überprüfung, ob ein Realm aktuell gültig ist, besteht auch die Möglichkeit, die Ausführung von Code innerhalb eines Realms zu erzwingen. Dies geschieht über *Realm.asyncExec (final Runnable runnable)*. Der SWT-*DisplayRealm* implementiert die Methode so, dass das Runnable auf dem User Interface Thread ausgeführt wird. Durch diese Abstraktion von dem Widget Toolkit steht auch der Verwendung von Realms innerhalb von Swing oder anderen Toolkits nichts im Weg. Ein Realm kann einer Instanz von *DataBindingContext* übergeben werden.

Fazit

Eclipse DataBinding ermöglicht eine einheitliche Synchronisation von Datenmodellen mit UI Widgets und wird in verschiedenen Projekten innerhalb von Eclipse standardmäßig eingesetzt. Mithilfe der zahlreichen Convenience APIs wird auch das Binden von komplexen UI Controls zum Kinderspiel. Die Wartbarkeit des Codes wird durch die Nutzung von Eclipse DataBinding gesteigert. Außerdem stellen verschiedene Projekte Erweiterungen bereit, die den Umgang mit Eclipse DataBinding noch weiter vereinfachen und aufzeigen, wie elegant eine Integration in andere UI Toolkits (u. a. Swing, GWT) möglich ist. Besonders Riena [3], EMF [4] (Eclipse Modeling Framework) und UFace [5] sind

Heiko Barth ist Diplom-Informatiker und seit 2006 als IT-Consultant, Coach und Entwickler bei der compeople AG in Frankfurt tätig. Den Schwerpunkt seiner Tätigkeit bildet die Entwicklung von Smart-Client-Architekturen und -Anwendungen. Darüber hinaus beschäftigt er sich mit Eclipse Equinox und der Rich Client Platform. Heiko Barth ist aktiver Commiter des Eclipse-Riena-Projekts.

Thorsten Schenkel ist als IT-Consultant und Softwareentwickler bei der compeople AG in Frankfurt tätig. Er beschäftigt sich seit 1999 intensiv mit Client-Server-Anwendungen sowie grafischen Benutzeroberflächen im Java-Umfeld, und in den letzten Jahren verstärkt mit Anwendungen auf Basis der Eclipse RCP. Seit Ende 2007 ist er aktiver Committer des Eclipse-Riena-Projekts.

>>Links & Literatur

[1] Eclipse-DataBinding-Wiki-Seite: http://wiki.eclipse.org/index.php/JFace_Data_Binding

[2] CVS-Repository: org.eclipse.jface.examples.databinding/src/org/eclipse/jface/examples/databinding/snippets

[3] Riena: http://www.eclipse.org/riena/

[4] EMF (Eclipse Modeling Framework): http://www.eclipse.org/modeling/emf/

[5] UFace: http://code.google.com/p/uface/