

Eclipse Rich Client Platform und NetBeans Platform im Vergleich

Das Rad muss immer noch nicht neu erfunden werden!

>> KAI TÖDTER

Für Entwickler sind Kenntnisse über mehr als eine Rich-Client-Plattform unabdingbar geworden. Aufgrund des damit verbundenen Aufwands ist es ihnen jedoch kaum möglich, diese im Detail zu betrachten und miteinander zu vergleichen. Anknüpfend an den Artikel, der in der letzten Ausgabe des Eclipse Magazins zum selben Thema erschien, sollen hier weitere Unterschiede der Eclipse Rich Client Platform und der NetBeans Platform untersucht werden.

Im Artikel der letzten Ausgabe, den Sie auch auf der Heft-CD finden, wurde ein Überblick über die Eclipse Rich Client Platform (Eclipse RCP) und die NetBeans Platform gegeben. Es wurden die Softwarearchitektur, die typischen Projektstrukturen und die erste kleine „Hello World!“-Applikation gezeigt. Anhand der Demoapplikation MP3-Manager wurden einige Konzepte vorgestellt und mit dem Vergleich des Action-Konzepts auf erste Details eingegangen. In diesem Artikel werden viele weitere und interessante Themen bezüglich beider Plattformen untersucht. Des Weiteren soll intensiv auf Eclipse Extension Points, NetBeans Registries und Lazy Loading eingegangen werden. Zusätzlich soll erklärt werden, wie Views und ID3-Tag-Editoren sowohl mit NetBeans als auch mit Eclipse realisiert werden können, was auch einen Exkurs in das NetBeans-Node-System und den Eclipse Tree Viewer umfasst. Weitere Themen sind die Integration der Update-Funktionalität und die Einbindung des Hilfesystems. Ein weiterer, interessanter Punkt sind die Mechanismen beider Plattformen, interaktiv mit „lange dauernden Operationen“ umzugehen, also die Möglichkeit, asynchrone Operationen auszuführen, die z.B.

vom Endbenutzer abgebrochen werden können. Der letzte Abschnitt wird sich der Anpassbarkeit (Customization) des User Interfaces widmen. Dieser Aspekt ist für Produktfamilien, Branding und z.B. das Corporate Look & Feel bedeutsam. Weitere detaillierte Informationen über Eclipse RCP bzw. NetBeans finden Sie unter [1], [2], [5] und [6].

Extension Points, Registries und Lazy Loading

Eine häufige Anforderung an Applikationssoftware ist, dass das Domänenmodell der Applikation als eine Menge von Interfaces implementiert ist, welche die Implementierung verstecken. Dadurch wird die Erstellung von verschiedenen Implementierungen für das Domänenmodell und deren Austauschbarkeit stark vereinfacht. Im Falle des MP3-Managers

wollte ich die ID3-Tag-Information als Interface liefern ([3] „Das Datenmodell“) und dann die entsprechende Implementierung später (lazy) zur Laufzeit anbinden. In der Eclipse-Welt konnte ich dafür recht einfach den Extension-Point-Mechanismus verwenden. Ich erstellte einen eigenen Extension Point, der eine Klasse verlangt, die ein bestimmtes Interface implementiert. In unserem Fall spezifizierte ich ein sehr einfaches Interface *IMP3InfoProvider*, das eine einzige Getter-Methode besitzt, um ein *MP3Info* zurückzuliefern. Hier noch eine kurze Erklärung zu der verwendeten Namenskonvention: In der Eclipse-Welt ist es eine verbreitete Praxis, jedes Interface mit einem großem „I“ zu beginnen. So kann sofort am Namen erkannt werden, dass es sich um ein Interface handelt, ohne dafür die Semantik kennen zu müssen. Ich übernehme hier diesen Ansatz und lasse meine Interfaces mit einem großen „I“ beginnen. Hier nun das einfache Interface *IMP3InfoProvider*:

```
public interface IMP3InfoProvider {
    public IMP3Info getMP3Info();
}
```

In der Eclipse-basierten Applikation erstellte ich den Extension Point *com.siemens.ct.mp3m.model.IMP3InfoProvider.mp3info*, der eine ID und eine

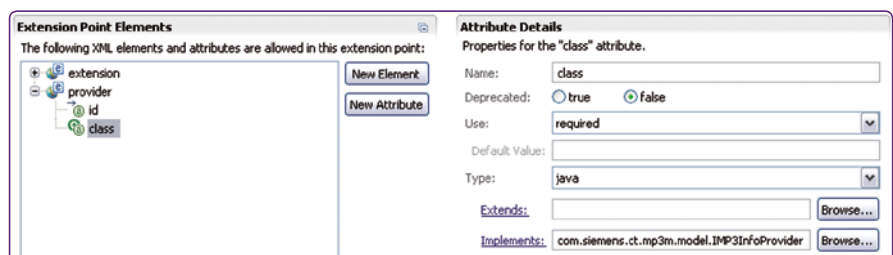


Abb. 1: Der Eclipse Extension Point Editor



Java-Klasse verlangt. Die Java-Klasse wiederum muss das Interface *com.siemens.ct.mp3m.model.IMP3InfoProvider* implementieren. Abbildung 1 zeigt den Eclipse Extension Point Editor.

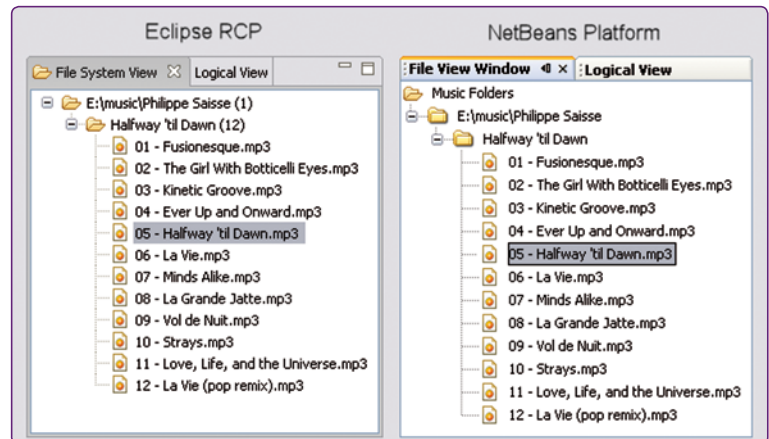
Ein Benutzer dieses Extension Points würde ihn erweitern und eine Implementierung des verlangten Interfaces liefern. Zur Laufzeit kann sich die RCP-Applikation alle bereitgestellten Extensions aus der so genannten *Extension Registry* holen. Danach muss die Applikation selbst entscheiden, was es mit den gefundenen Extensions machen will. Listing 1 zeigt einen Code-Schnipsel, der verdeutlicht, wie alle Extensions für unseren Extension Point in einer Schleife geholt werden. Das Code-Beispiel ist allerdings etwas vereinfacht, in einer realen Applikation würde die dynamische Erzeugung der Java-Klasse etwas sicherer ablaufen (z.B. mithilfe der Verwendung eines *ISafeRunnable*), um zu verhindern, dass unbekannte Extension-Lieferanten die Applikation korrumpieren können.

Normalerweise werden alle Extensions in einer Schleife behandelt, da man erst zur Laufzeit weiß, wie viele Extensions gerade zur Verfügung stehen. Das weitere Verfahren hängt dann von der domänenspezifischen Strategie ab. In der NetBeans-Welt verwendete ich den Service-Mechanismus und die globale Registry, um das gewünschte Verhalten zu erreichen. Das Interface *IMP3InfoProvider* ist exakt das gleiche, wie in der Eclipse-Implementierung. Um einen Service (eine Implementierung eines Java Interfaces) zu erzeugen, sind die folgenden Schritte notwendig:

- Erstellen eines Verzeichnisses META-INF/SERVICES
- Erstellen einer Datei in diesem Verzeichnis mit dem voll qualifizierten Namen des zu implementierenden Interfaces, in unserem Fall *com.siemens.ct.mp3m.model.IMP3InfoProvider*
- Schreiben des voll qualifizierten Namens der Implementierung in diese Datei. In unserem Fall ist dies *com.siemens.ct.nb.mp3m.model.ueberdosis.UeberdosisMP3InfoProvider*, da ich die Überdosis-Bibliothek verwendet habe, um die ID3-Tag-Funktionalität zu implementieren [4]

Dieser Mechanismus ist übrigens nicht NetBeans-spezifisch, sondern Teil der

Abb. 2:
Der MP3
TreeView
mit beiden
Plattformen
implementiert



Standard-JAR-Spezifikation. Dieser Mechanismus kann also in jedem Java-Programm verwendet werden. Im NetBeans-Fall muss diese Struktur natürlich mit dem NetBeans-Modul deployt werden. Zur Laufzeit kann die Klasse dann folgendermaßen dynamisch erzeugt werden

```
IMP3InfoProvider provider = (IMP3InfoProvider)Lookup
    .getDefault().lookup(IMP3InfoProvider.class);
```

In diesem Beispiel wird nur eine einzige Implementierung des Interfaces erfragt. Mit *Lookup.getDefault().lookupAll()* bekommt man alle zur Laufzeit verfügbaren Implementierungen des Interface. Der NetBeans-Global-Lookup-Ansatz besticht durch seine Einfachheit, denn es ist nicht mit viel Aufwand verbunden, Implementierungen in der globalen Registry zu registrieren und diese zur Laufzeit zu erfragen. NetBeans bietet noch weitere Möglichkeiten der Realisierung von Lazy Loading (z.B. in der *layer.xml*), auf die an dieser Stelle aber nicht weiter eingegangen werden soll.

Doch nun wieder zurück in die Eclipse-Welt. In diesem kleinen Beispiel wurde der Extension-Point-Mechanismus lediglich benutzt, um das Lazy Loading einer Java-Klasse zu realisieren. Natürlich ist der Extension-Point-Mechanismus ein generelles und sehr flexibles Konzept, das sehr viel mehr bietet als die einfache dynamische Klassenerzeugung. Andererseits kann man auch in der NetBeans-Welt mit der *layer.xml* weit mehr als dynamische Klassenerzeugung realisieren, was aber nicht Inhalt dieses Abschnitts sein sollte. Das Entscheidende ist, dass beide Plattformen Mechanismen bieten, welche die Realisierung von Lazy Loading ermöglichen,

um eine lose gekoppelte und skalierbare Softwarearchitektur zu erstellen.

Erstellen eines MP3-Datei-Views

Für meine Demoapplikation wollte ich einen TreeView implementieren, der eine Reihe von Musikverzeichnissen als Wurzelobjekte hat. Weiterhin sollten nur Verzeichnisse und MP3-Dateien angezeigt werden. Abbildung 2 zeigt die Implementierungen mit Eclipse und mit NetBeans.

Bei Eclipse gibt es verschiedene Möglichkeiten dies zu implementieren. Ein Ansatz wäre das *Common Navigator Framework*, das mit Eclipse 3.2 eingeführt wurde, wieder zu verwenden. Ich möchte das Beispiel aber möglichst einfach und verständlich halten, daher zeige ich die Implementierung mit einem JFace TreeViewer zusammen mit *ContentProvider* und *LabelProvider*. Der ContentProvider ist für das Mapping von Domänenobjekten auf Baumknoten verantwortlich, während ein LabelProvider sich um den sichtbaren Text und das Icon der Baumknoten kümmert. Um selbst diesen Mechanismus zu vereinfachen,

Listing 1

```
IConfigurationElement[] providers =
    Platform.getExtensionRegistry()
        .getConfigurationElementsFor("com.siemens.
            ct.mp3m.model", "mp3info");
for (IConfigurationElement provider : providers) {
    try {
        IMP3InfoProvider provider = (IMP3InfoProvider)
            provider.createExecutableExtension("class");
        // do something useful with the dynamically
        // created class...
    } catch (Throwable e) {
        LogUtil.logError("com.siemens.ct.
            mp3m.model", e);
    }
}
```



bietet Eclipse einen eleganten Adapteransatz: Anstelle von LabelProvidern und ContentProvidern, die zwischen verschiedenen Domänenobjekten des Modells unterscheiden müssen (z.B. Verzeichnisse und MP3-Dateien), habe ich eine AdapterFactory erstellt, die wiederum zwei Adapter registriert und bereitstellt, einen für Verzeichnisse und einen für MP3-Dateien. Die Adapter übernehmen dann das Mapping auf sichtbare Baumknoten. Um diesen Ansatz zu realisieren, habe ich ein kleines Datenmodell erstellt, das die physikalische Dateistruktur auf die eigenen Klassen *Mp3Directory* und *Mp3File* abbildet. Die Implementierung ist natürlich trivial und ich möchte an dieser Stelle nicht näher darauf eingehen. Listing 2 zeigt die Implementierung der AdapterFactory.

Nachdem die AdapterFactory implementiert wurde, konnte diese direkt in einem JFace TreeViewer verwendet werden. Nun musste noch die eigentliche View-Klasse *FileSystemView* erstellt werden, die eine Unterklasse von *ViewPart* darstellt. Listing 3 zeigt die Initialisierung des TreeViewers.

Weil der View möglichst lose an die Applikation gekoppelt werden soll, habe ich den Extension Point *org.eclipse.ui.views* erweitert und so den *FileSystemView* dynamisch ladbar gemacht. Damit der View auch in der Applikation angezeigt wird, wurde ein Platzhalter mit der entsprechenden ID in das Layout der initialen Perspektive der Applikation gepackt. So wird der View gleich beim ersten Start der Applikation angezeigt.

Bei NetBeans verwendete ich einen anderen, aber auch sehr eleganten Ansatz, um Dateien auf Datenobjekten abzubilden und diese dann auf entsprechende Baumknoten zu mappen. Das NetBeans-Node-System ist ein recht komplexes Thema, das in [6] ausführlich behandelt wird. Im MP3-Manager wollte ich einen neuen Dateityp mit der Endung *.mp3* einführen. Der „*New File Type...*“-Wizard der NetBeans Platform erzeugt dafür die komplette Infrastruktur und alle notwendigen Java-Klassen. Nachdem alle nötigen Informationen in den Wizard eingegeben worden waren, erzeugte dieser die folgenden Java-Klassen:

- *MP3FileDataLoader*
- *MP3FileDataLoaderBeanInfo*
- *MP3FileDataNode*
- *MP3FileDataObject*
- *MP3FileResolver.xml*
- *MP3FileTemplate.mp3*

Die meisten dieser generierten Dateien musste ich gar nicht anfassen und sie konnten direkt verwendet werden. Wie man einen Editor bei selektierter *MP3FileDataNode* öffnet, wird im nächsten Abschnitt erklärt. In diesem Abschnitt konzentriere ich mich auf den MP3 Tree Viewer, wie in Listing 4 gezeigt.

Um den Viewer zu implementieren, wurde zunächst eine *TopComponent* gebraucht, die wiederum *ExplorerManager.Provider* implementiert. Im Konstruktor erzeuge und verwende ich einen *BeanTreeView*, der als Container für den *ExplorerManager* fungiert. Danach erzeuge ich den eigentlichen *ExplorerManager* und füttere ihn mit dem Wurzelknoten des Baumes. Die Mischung aus *ExplorerManager* und *BeanTreeView*, zusammen mit den durch den

Listing 2 ✖

```
public class AdapterFactory implements IAdapterFactory {
    private final String ID = "com.siemens.ct.mp3m.ui.views";

    private IWorkbenchAdapter directoryAdapter = new IWorkbenchAdapter() {
        public Object getParent(Object o) {
            return ((Mp3Directory) o).getDirectory();
        }
    };

    public String getLabel(Object o) {
        Mp3Directory directory = ((Mp3Directory) o);
        return directory.getName();
    }

    public ImageDescriptor getImageDescriptor(Object object) {
        return AbstractUIPlugin.imageDescriptorFromPlugin(ID, IImageKeys.FOLDER);
    }

    public Object[] getChildren(Object o) {
        return ((Mp3Directory) o).getMp3Files();
    }
};

private IWorkbenchAdapter entryAdapter = new IWorkbenchAdapter() {
    public Object getParent(Object o) {
        return ((Mp3File) o).getDirectory();
    }
};

public String getLabel(Object o) {
    Mp3File entry = ((Mp3File) o);
    return entry.getName();
}

public ImageDescriptor getImageDescriptor(Object object) {
    return AbstractUIPlugin.imageDescriptorFromPlugin(ID, IImageKeys.MP3);
}

public Object[] getChildren(Object o) {
    return new Object[0];
}
};

public Object getAdapter(Object adaptableObject, Class adapterType) {
    if (adapterType == IWorkbenchAdapter.class && adaptableObject instanceof Mp3Directory)
        return directoryAdapter;
    if (adapterType == IWorkbenchAdapter.class && adaptableObject instanceof Mp3File)
        return entryAdapter;
    return null;
}

public Class[] getAdapterList() {
    return new Class[] { IWorkbenchAdapter.class };
}
}
```

Listing 3 ✖

```
public void createPartControl(Composite parent) {
    treeViewer = new TreeViewer(parent, SWT.BORDER | SWT.MULTI | SWT.V_SCROLL);
    Platform.getAdapterManager().registerAdapters(adapterFactory, Mp3File.class);
    treeViewer.setLabelProvider(new WorkbenchLabelProvider());
    treeViewer.setContentProvider(new BaseWorkbenchContentProvider());
    ...
}
```



Wizard erzeugten Dateien, macht das Erstellen des MP3 TreeViewers recht einfach. In Listing 4 habe ich einiges von der generellen Initialisierung weglassen und mich auf die für den Viewer wichtigen Teile konzentriert. Die erste domänenspezifische Klasse, die in Listing 4 verwendet wird, ist *Mp3MusicFolderChildren*. Listing 5 zeigt die Implementierung dieser Klasse.

In der Methode *addNotify()* wird eine Liste von Musikverzeichnissen (*musicFolders*) erzeugt. Wenn diese Liste ungleich null und nicht leer ist, wird sie als die Schlüssel (*keys*) dieses Knotens gesetzt und als erstes Level im Baum erscheinen. Der trickreiche Teil ist nun die Anbindung der Unterbäume mit den MP3-Verzeichnissen und MP3-Dateien an die Musikverzeichnisse. Es sollten ja nicht alle Dateien im Baum dargestellt werden, sondern nur MP3-Dateien. Eine Möglichkeit, um dies zu erreichen, ist die Verwendung einer Unterklasse von *FilterNode*, die wiederum intern eine Unterklasse von *FilterNode.children* verwendet. Listing 6 zeigt die entsprechenden Klassen *MP3FilterNodeProxy* und *ProxyChildren*.

Listing 4

```
final class FileViewTopComponent extends TopComponent
    implements ExplorerManager.
        Provider, MusicFoldersChangeListener {

    private transient BeanTreeView beanTreeView;
    private transient ExplorerManager explorerManager;

    private FileViewTopComponent() {
        initComponents();
        beanTreeView = new BeanTreeView();
        add(beanTreeView, BorderLayout.CENTER);

        explorerManager = new ExplorerManager();
        AbstractNode root = new AbstractNode(
            new Mp3MusicFolderChildren());
        root.setIconBaseWithExtension(ICON_PATH);
        explorerManager.setRootContext(root);
        explorerManager.getRootContext()
            .setDisplayName("Music Folders");

        ActionMap map = this.getActionMap();
        associateLookup(ExplorerUtils.createLookup(
            explorerManager, map));
    }

    public ExplorerManager getExplorerManager() {
        return explorerManager;
    }

    // ...
```

Diese Klasse testet, ob der zu erzeugende Knoten ein Verzeichnis ist oder nicht. Wenn ja, wird einfach die *createNodes()*-Implementierung der Superklasse zurückgeliefert. Sollte der Knoten kein Verzeichnis sein, muss der Knoten ein Cookie (dazu später mehr) für die Klasse *DataObject* besitzen. Wenn das entsprechende Dateiobjekt den richtigen MIME-Typ hat (wie im Wizard angegeben), wird der Knoten geklont und zurückgeliefert. Ansonsten liefere ich ein leeres Node-Array. Mit diesem Mechanismus wurden nur Verzeichnisse und MP3-Dateien gefiltert – genau das, was erreicht werden sollte.

Doch nun einige Erläuterungen zu Cookies in NetBeans: Der Begriff Cookie wird vielen von Ihnen aus der Webbrowser-Welt ein Begriff sein. Bei NetBeans haben Cookies aber eine ganz andere Bedeutung: Da oft nicht von vornherein bekannt ist, welche Eigenschaften bestimmte NetBeans-Objekte haben (z.B. Nodes), können diesen Ob-

jekten dynamisch zur Laufzeit bestimmte Cookies hinzugefügt werden. Jedes Cookie repräsentiert eine bestimmte Eigenschaft. Soll zum Beispiel ein Objekt speicherbar sein, wird ein *SaveCookie* verwendet. Ich werde ein *SaveCookie* auch im nächsten Abschnitt beim MP3-Tag-Editor verwenden, dann wird die Verwendung von Cookies etwas klarer.

Nun ist die Implementierung vollständig und der *ExplorerManager* liefert einen Baum, der alle Musikverzeichnisse des Datenmodells als Baumknoten im ersten Level darstellt. Die weiteren Level enthalten dann Verzeichnisse und MP3-Dateien. Ein Grund für die vorgestellte Implementierung war natürlich, dass ich Erfahrungen mit dem NetBeans-Node-System sammeln wollte. Jedoch bietet die NetBeans Plattform noch andere (möglicherweise bessere) Ansätze, um

Listing 5

```
public class Mp3MusicFolderChildren extends
    Children.Keys<String> {

    String[] folders;

    protected Node[] createNodes(String musicFolder) {
        try {
            FileObject root = FileUtil.toFileObject(
                new File(musicFolder));
            if (root != null) {
                DataObject rootDataObject =
                    DataObject.find(root);
                if (rootDataObject != null) {
                    Node node = new Mp3FileNodeProxy(
                        rootDataObject.getNodeDelegate());
                    node.setDisplayName(musicFolder);
                    return new Node[]{node};
                }
            }
        } catch (DataObjectNotFoundException ex) {
            ex.printStackTrace();
        }
        return new Node[]{};
    }

    protected void addNotify() {
        List<String> musicFolders = MusicFolders
            .getMusicFolders();
        if (musicFolders != null && !musicFolders.isEmpty()) {
            folders = musicFolders.toArray(new String[]{});
            setKeys(folders);
        }
    }
}
```

Listing 6

```
public class Mp3FileNodeProxy extends FilterNode {
    public Mp3FileNodeProxy(Node original) {
        super(original, new ProxyChildren(original));
    }
}

class ProxyChildren extends FilterNode.Children {

    public ProxyChildren(Node owner) {
        super(owner);
    }

    protected Node copyNode(Node original) {
        return new Mp3FileNodeProxy(original);
    }

    protected Node[] createNodes(Node key) {
        Node n = (Node)key;
        boolean isDirectory = n.getCookie(
            DataFolder.class) != null;

        if (isDirectory) {
            return super.createNodes((Node) key);
        } else {
            FileObject fo = ((DataObject)n.getCookie(
                DataObject.class)).getPrimaryFile();

            if (Mp3FileDataLoader.REQUIRED_MIME
                .equals(fo.getMIMEType())) {
                return new Node[] {
                    n.cloneNode()
                };
            } else {
                return new Node[0];
            }
        }
    }
}
```

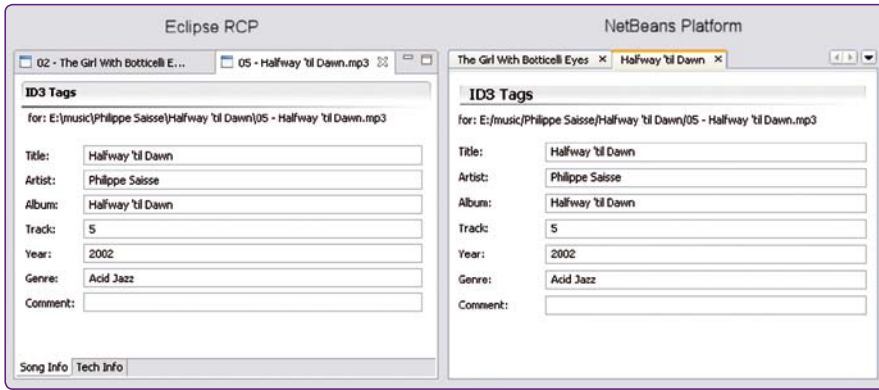


Abb. 3: Der ID3-Tag-Editor implementiert mit beiden Plattformen

das gleiche zu erreichen. Ein Ansatz, der wahrscheinlich sogar noch weniger Code verwenden würde, wäre der Gebrauch von *DataShadows*. Ein *DataShadow* könnte als Link vom NetBeans-File-System zu einem physikalischen Verzeichnis auf der Festplatte dienen, was die *MusicFolder*-Klasse im Datenmodell [3] überflüssig machen würde.

Erstellen eines ID3-Tag-Editors

Nachdem ich nun die MP3-Dateien browsen konnte, wäre der nächste inter-

essante Schritt, die entsprechenden ID3-Tags zu editieren. In der Eclipse-basierten Implementierung erweiterte ich den Extension Point *org.eclipse.ui.editors* und implementierte eine entsprechende Java-Klasse. In unserem Beispiel erweiterte ich den *FormsEditor*, der schon die Infrastruktur für einen formularbasierten Multi-Page-Editor zur Verfügung stellt. Ein Multi-Page-Editor hat verschiedene Seiten, die unten durch Tabs selektiert werden können.

In Abbildung 3 sehen Sie bei der Eclipse-Implementierung unten die zwei Tabs für die Seiten „Song Info“ und „Tech Info“. Listing 7 zeigt das Eclipse-RCP-basierte Code-Skelett des Editors.

Jeder *FormEditor* benötigt einen *IEditorInput*, der die domänenspezifischen Daten für den Editor kapselt. In unserem Beispiel wird ein *PathEditorInput* verwendet, der einen Pfad im Dateisystem kapselt. Nun müssen nur noch die eigentlichen Seitenklassen implementiert werden. Für diesen Job erweiterte ich einfach die Klasse *org.eclipse.ui.forms.editor.FormPage*. Eine Formularseite ist zuständig für:

- das visuelle Layout der Seite,
- das Mapping zwischen Domänenobjekten und UI-Objekten, wie z.B. Textfelder,
- das „dirty“-Markieren der Seite (also die Anzeige, dass sich auf der Seite etwas geändert hat),
- das eigentliche Abspeichern in das Domänenmodell.

Die Implementierung von *FormPages* ist immer ähnlich und nicht besonders kompliziert. Um das Seiten-Layout zu gestalten, hätte ich den Eclipse Visual Editor [7] benutzen können. Dieser ist eine weiteres Eclipse-Open-Source-Projekt, aber

nicht Teil der Standard-Eclipse-SDKs. (Für Eclipse gibt es eine ganze Reihe von GUI Builder, sowohl für Swing wie auch für SWT. Einige unterstützen sogar das *GroupLayout*, welches vom NetBeans GUI Builder Matisse verwendet wird. Die meisten dieser Eclipse GUI Builder sind allerdings kommerzielle Produkte und nicht frei erhältlich.) Aber da das Formular einfach aufgebaut ist, habe ich es manuell mit dem Eclipse Forms API implementiert. Nachdem der Editor implementiert ist, wäre der letzte Schritt, den Editor zu öffnen, wenn eine MP3-Datei selektiert ist (oder Doppelklick). In Eclipse kann das elegant realisiert werden, indem der Editor für eine Dateieindung registriert wird, in unserem Fall „mp3“. Dann könnten alle Views die Eclipse Extension Registry zur Laufzeit fragen, welche Editoren für „mp3“ gerade zur Verfügung stehen und den entsprechenden Editor benutzen. Dieser Ansatz ist flexibel und skaliert sehr gut, da die Editoren einfach ausgetauscht werden können (selbst zur Laufzeit), ohne den Java-Code in den aufrufenden Viewern oder Actions anfassen zu müssen. Eine nette Sache, welche die Eclipse RCP standardmäßig zur Verfügung stellt, ist die Anbindung der Editoren an die generische Save Action, die ich sowohl in der Tool-Bar wie auch im File-Menü verwendet habe. Ich musste überhaupt keinen eigenen Code schreiben: Sobald der Editor als „dirty“ markiert ist, wird automatisch sowohl der Menüeintrag als auch der Toolbar Button enabled und an die Save-Methode des gerade offenen Editors angebunden.

Nun zur NetBeans-Implementierung. Bei NetBeans fing ich mit dem Editor genau so an wie mit dem Viewer: Ich verwendete den „New Window...“-Wizard, um eine Unterklasse von *TopComponent* zu erstellen, die ich *Id3EditorTopComponent* nannte. Der Wizard hat dann das Code-Skelett für die Java-Klasse sowie etwas Infrastruktur generiert. Für das Layout des Formulars würde ich normalerweise das JGoodies Forms Layout [8] verwenden, doch wollte ich hier den NetBeans GUI Editor Matisse ausprobieren. Matisse ist integraler Bestandteil der NetBeans IDE und ein wirklich großartiges Tool. Mit Matisse ist es selbst für Software-Ingenieure (die bis auf ein paar Ausnahmen oft fürchterliche GUI-Designer sind ...☺) möglich, ansprechende, gut benutzbare

Listing 7 ✨

```
public class Id3Editor extends FormEditor {
    @Override
    protected void addPages() {
        //Hier werden alle Editor-Seiten hinzugefügt
    }

    @Override
    public void doSave(IProgressMonitor monitor) {
        //Hier würde die "save"-Funktionalität
        //implementiert werden
    }

    @Override
    public void doSaveAs() {
        //Hier würde die "save as"-Funktionalität
        //implementiert werden
    }

    @Override
    public boolean isSaveAsAllowed() {
        return false;
    }

    protected void setInput(IEditorInput input) {
        //Diese Methode wird zur Laufzeit aufgerufen,
        //um den Editor Input zu setzen
    }

    @Override
    public boolean isDirty() {
        //zeigt an, ob der Editor "dirty" ist,
        //also abgespeichert werden kann
    }
}
```



Formulare und andere UI-Komponenten zu erstellen. Zwei Anforderungen sollten auf jeden Fall auch in der NetBeans-Implementierung realisiert werden:

- Eine möglichst lose Kopplung des Tree-Views und des ID3-Tag-Editors. Der View sollte nichts über eine spezifische Implementierung des Editors wissen.
- Eine Verbindung des Save-Menu-Items und des Save-Toolbar-Buttons zum gerade offenen Editor.

Die lose Kopplung löste ich mit genau dem gleichen Mechanismus, den ich schon im Abschnitt oben zu „Extension Points, Registries und Lazy Loading“ beschrieben habe. Zunächst wurde das Interface `com.siemens.ct.nb.mp3m.model.actions.IOpenEditorAction` definiert:

```
public interface IOpenEditorAction {
    public void openEditor(IMP3Info mp3Info);
}
```

Dann wurde die vom Wizard generierte Action modifiziert und der Code für das Öffnen des Editors in die `openEditor`-Methode eingefügt. Listing 8 zeigt die `Id3EditorAction`.

Schließlich wurde noch ein Popup-Menü für das Öffnen des Editors hin-

Listing 8

```
public class Id3EditorAction extends
    AbstractAction implements IOpenEditorAction {

    HashMap<String, Id3EditorTopComponent> editors =
        new HashMap<String, Id3EditorTopComponent>();
    public Id3EditorAction() {
        super(NbBundle.getMessage(Id3EditorAction
            .class, "CTL_Id3EditorAction"));
    }

    public void actionPerformed(ActionEvent evt) {
        // Do nothing so far
    }

    public void openEditor(IMP3Info mp3Info) {
        Id3EditorTopComponent editor =
            editors.get(mp3Info.getMP3File());
        if(editor == null) {
            editor = new Id3EditorTopComponent(mp3Info);
            editors.put(mp3Info.getMP3File(), editor);
        }
        editor.setMP3Info(mp3Info);
        editor.open();
        editor.requestActive();
    }
}
```

zugefügt, das man bei jeder Selektion einer MP3-Datei mit der rechten Maustaste erhält. Listing 9 zeigt die `Mp3FileDataNode`, mit der ich das Popup-Menü realisierte.

Die zwei wichtigsten Dinge bei der Implementierung sind die dynamische Instanziierung der `IOpenEditorAction` und der Aufruf von `openEditor()` in der Methode `actionPerformed()` der `OpenEditorAction`.

Jetzt zur zweiten Anforderung, die realisiert werden soll: das Anbinden des gerade offenen Editors sowohl an das Save-Menu-Item wie auch an den Save-Toolbar-Button. Hierfür müsste nur ein Node-Objekt mit einem `SaveCookie` benutzt werden. Immer, wenn eine `TopComponent` aktiviert wird, setzt sie eine Menge von aktivierten Nodes. In unserem Fall musste ein abstraktes Node-Objekt erzeugt werden, das in die Menge der aktiven Nodes eingefügt wird und im Falle eines „dirty“-Editors ein entsprechendes `SaveCookie` zurückliefert. Listing 10 zeigt die Implementierung.

Wenn wir nun die Implementierungen des ID3-Tag-Editors vergleichen, sehen wir, dass sich die wichtigsten Anforde-

rungen mit beiden Plattformen gut realisieren lassen. Natürlich hat jede Plattform hierbei ihren eigenen Stil. Das Konzept eines Multi-Page-Editors wird im Eclipse SDK häufig verwendet und ist damit auch unter Eclipse RCP leicht wieder zu verwenden. In NetBeans gibt es standardmäßig kein solches Konzept. Das ist aber nicht weiter tragisch, da mehrere Editor-Seiten bei Bedarf recht leicht über `JTabbedPanes` realisiert werden könnten.

Integration von Update-Funktionalität

Für die meisten Rich-Client-Applikationen ist die integrierte Update-Funktionalität eine Basisanforderung. Sowohl Eclipse wie auch NetBeans beinhalten ein ausgefeiltes Update-System. Ich möchte an dieser Stelle nicht in die Details gehen, aber einige allgemeine Dinge bezüglich Update ansprechen. Wenn ein Update-Prozess für einen domänenspezifischen Rich Client entworfen wird, würde ich empfehlen, erstmal über die Anforderungen und Kenntnisse der zukünftigen Benutzer nachzudenken. Eclipse bietet einen Mechanismus, um Features upzudaten. Das gibt es schon für installierte

Listing 9

```
public class Mp3FileDataNode extends DataNode {
    private static final String IMAGE_ICON_BASE =
        "com/siemens/ct/nb/mp3m/model/mp3file/mp3.gif";
    private Mp3FileDataObject mp3FileDataObject;

    public Mp3FileDataNode(Mp3FileDataObject obj) {
        super(obj, Children.LEAF);
        setIconBaseWithExtension(IMAGE_ICON_BASE);
        mp3FileDataObject = obj;
    }

    public Action[] getActions(boolean popup) {
        return new Action[] { new OpenEditorAction() };
    }

    public Action getPreferredAction() {
        return new OpenEditorAction();
    }

    private class OpenEditorAction extends
        AbstractAction implements Presenter.Popup {
        private IOpenEditorAction action;
        public OpenEditorAction() {
            action = (IOpenEditorAction)Lookup.getDefault()
                .lookup(IOpenEditorAction.class);
        }

        public String getName() {
            return "Open Editor";
        }

        public String getDisplayName() {
            return "Open Editor";
        }

        public JMenuItem getPopupPresenter() {
            JMenuItem item = new JMenuItem(getDisplayName());
            item.addActionListener(this);
            return item;
        }

        public void actionPerformed(ActionEvent e) {
            if(action == null) {
                System.out.println("Cannot open editor,
                    no Implementation of IOpenEditorAction found.");
                return;
            }
            String path = mp3FileDataObject.getPrimaryFile()
                .getPath();
            IMP3Info info = MP3Infos.getMP3Info(path);
            if(info == null) {
                MP3Infos.addMP3File(path);
                info = MP3Infos.getMP3Info(path);
            }
            action.openEditor(info);
        }
    }
}
```



Features, der Endbenutzer kann jedoch auch neue Update-Sites hinzufügen (aus dem Internet oder lokal), wo nach neuen Features gesucht wird. Nachdem die neuen oder diejenigen Features gefunden wurden, die aktualisiert werden sollen, werden Signierungsinformationen und Lizenzen angezeigt. Vertraut der Endbenutzer dem Anbieter und akzeptiert er die Lizenz, kann er das Update oder das neue Feature installieren. Das NetBeans Update Center bietet einen sehr ähnlichen Mechanismus, basierend of NetBeans-Modulen. Aus meiner eigenen Erfahrung mit Rich-Client-Kundenprojekten würde ich aber sagen, dass diese Funktionalität die meisten Endbenutzer (die keine Software-Ingenieure sind) überfordert und für die meisten Rich-Client-App-

plikationen in dieser komplexen Form ungeeignet ist. Vielleicht eignet sich ein „unsichtbares“ Update im Hintergrund oder ein einfaches Update mit einem einzigen Mausklick besser für die Applikation. Hier ein Konzept, das Sie bei beiden Plattformen realisieren können:

- Zunächst könnte zum Test die Update-Funktionalität der entsprechenden IDE in die Applikation integriert werden. So kann man sich mit den eingebauten Basismechanismem vertraut machen und testen, ob die Update-Prinzipien funktionieren. Dies ist mit beiden Plattformen relativ einfach machbar.
- Dann sollte entschieden werden, welche Art von Granularität und Komplexität in der Applikation angeboten werden soll.
- Zuletzt sollte versucht werden, einige der feiner granulierten APIs wieder zu verwenden, um die gewünschte Update-Funktionalität zu verwirklichen.

Der Code-Schnipsel in Listing 11 zeigt, wie Sie in Eclipse einfach die Funktionalität „Update nur von installierten Features“ implementieren können.

BusiIndicator.showWhile() ist ein nettes kleines Beispiel für die standard-

mäßig implementierte gute Usability. Wenn dieser Code abläuft, wird der Mauszeiger automatisch auf einen „Busy Cursor“ gesetzt, um anzuzeigen, dass etwas im Hintergrund passiert. Sicherlich ist dies eines der einfachsten Beispiele für integriertes Benutzer-Feedback. Jetzt möchte ich einen genaueren Blick auf den Umgang beider Plattformen mit lange andauernden Operationen und interaktivem Benutzer-Feedback werfen.

Längere Operationen und Benutzerinteraktionen

Eclipse bietet ein Jobs API und ein Progress Monitor API an, welche die gleichzeitige Ausführung von potenziell länger dauernden Operationen unterstützen, sowohl mit Fortschrittsanzeige als auch Benutzerinteraktion. Listing 12 zeigt ein Beispiel, wie zehn Work Items im Hintergrund abgearbeitet werden können.

Als Standard wird ein abbrechbarer Fortschrittsdialog geöffnet. Ein netter Nebeneffekt ist die eingebaute Usability, wenn man *PlatformUI.getWorkbench().getProgressService().busyCursorWhile(runnable)* benutzt. Zuerst wird der Mauszeiger „busy“ geschaltet, aber kein Fortschrittsdialog angezeigt, bevor der Timeout für länger dauernde Operationen überschritten ist. Dann erst erscheint ein Fortschrittsdialog, der automatisch geschlossen wird, wenn die Operation zu Ende gelaufen ist oder vom Benutzer abgebrochen wurde. Dieses Verhalten ist sehr sinnvoll, wenn die Operation kontextabhängig kurz ist oder länger dauert, denn Endbenutzer wären vielleicht irritiert, wenn bei einer Operation ganz kurz ein Fortschrittsdialog aufpoppt, der aber wegen der Kürze der Operation nicht lesbar und sinnvoll wäre. Einen guten Onlineartikel über das Jobs API finden Sie unter [9]. In der NetBeans-basierten Applikation verwendete ich das Progress API, wie in Listing 13 gezeigt.

Als Default wird der Fortschritt in der Status-Bar der Applikation angezeigt und die Operation kann über einen kleinen Button abgebrochen werden. Auch NetBeans stellt also ein API für das Job-Management zur Verfügung: das Task API. Eine Task kann folgendermaßen angestoßen werden:

```
Task task = new Task(new MyJob());
RequestProcessor.getDefault().post(task);
```

Listing 10

```
class MyNode extends AbstractNode {
    private SaveCookie saveCookie;

    public MyNode(IMP3Info info) {
        super(Children.LEAF);
        saveCookie = new MySaveCookie(info);
    }

    public Node.Cookie getCookie(Class class1) {
        if(class1 == SaveCookie.class && isDirty()) {
            return saveCookie;
        }
        return super.getCookie(class1);
    }
}

class MySaveCookie implements SaveCookie {
    private IMP3Info info;

    public MySaveCookie(IMP3Info info) {
        this.info = info;
    }

    public void save() throws IOException {
        // Hier passiert das eigentliche Abspeichern
    }
}
```

Listing 11

```
BusyIndicator.showWhile(window.getShell()
    .getDisplay(), new Runnable() {
    public void run() {
        UpdateJob job = new UpdateJob(Messages
            .getString("UpdateAction.searchForUpdates"),
            false, false);

        job.setUser(true);
        job.setPriority(Job.INTERACTIVE);
        UpdateManagerUI.openInstaller(
            window.getShell(), job);
    }
});
```

Listing 12

```
IRunnableWithProgress runnable =
    new IRunnableWithProgress() {
    public void run(IPressMonitor
        progressMonitor) throws InterruptedException {
        int workitemCount = 10;
        progressMonitor.beginTask("Task Name",
            workItemCount);

        for (int n=0; n<workItemCount; n++) {
            if(!progressMonitor.isCanceled()) {
                // Abarbeitung eines WorkItems
            } else {
                // hier wegen des Abbruchs aufräumen
            }
            break;
        }
        progressMonitor.worked(1);
    }
    progressMonitor.done();
};

try {
    PlatformUI.getWorkbench().getProgressService()
        .busyCursorWhile(runnable);
} catch (Exception e) {
    // Exceptions immer sinnvoll abfangen!
}
```

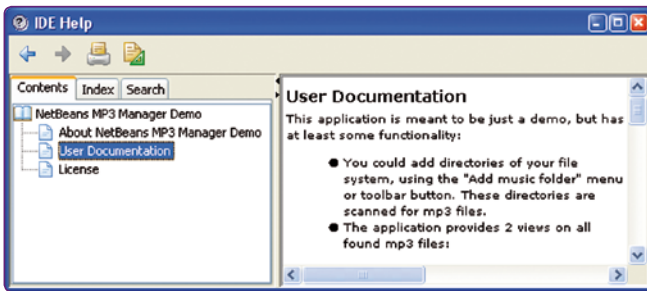


Abb. 4: Die Hilfe des NetBeans-basierten MP3-Managers

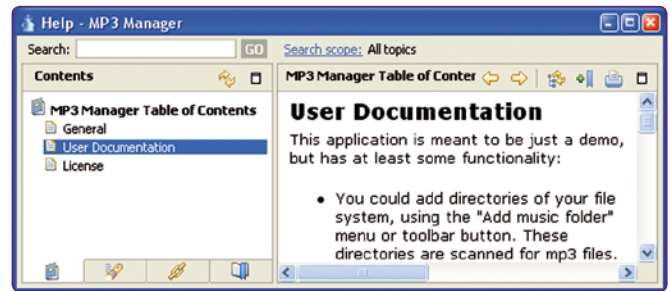


Abb. 5: Die Hilfe des Eclipse-basierten MP3-Managers

Wie die Beispiele zeigen, bieten sowohl NetBeans als auch Eclipse-Mechanismen, den Umgang mit länger dauernden Operationen inklusive Benutzerinteraktion. Natürlich sind die Implementierungen, wie immer, recht unterschiedlich. Bei komplexeren Anforderungen sollten die Möglichkeiten der jeweiligen Plattform im Detail untersucht werden. Für meine kleine MP3-Managerapplikation haben beide Plattformen ihre Arbeit sehr gut erledigt.

Integration des Hilfesystems

Beide Plattformen bieten die Möglichkeit, das eigene Hilfesystem auch in eigene Applikationen zu integrieren. Während die NetBeans-Hilfe auf JavaHelp [10] basiert, baut das Eclipse-Hilfesystem in der Version 3.2 auf Tomcat [11] und ab Version 3.3 auf Jetty [12] auf. Als Such- und Index-Engine wird in beiden Eclipse-Versionen Lucene verwendet [13]. Bei beiden Plattformen ist es sehr einfach, das entsprechende Hilfesystem in eigene Applikationen einzubinden. Beide Systeme verlangen XML-Beschreibungen für die Inhaltsverzeichnisse und HTML für die Seiten. Ein Unterschied der Hilfesysteme ist, dass JavaHelp nur statischen Inhalt darstellen kann, während das Eclipse-Hilfesystem aufgrund des voll integrierten Webservers auch dynamischen

Inhalt unterstützt. Der Preis, der dafür gezahlt werden muss, ist die erhebliche Größe des Eclipse-Hilfesystems im Vergleich zu JavaHelp. Letztendlich bieten aber beide Plattformen ein ausgeklügeltes Hilfesystem. Abbildung 4 zeigt die JavaHelp-basierte und Abbildung 5 die Eclipse-basierte Hilfe des MP3-Managers.

Anpassung des Look & Feels

Beim User-Interface-Design einer Rich-Client-Applikation ist es manchmal wünschenswert, das Look & Feel anzupassen. Das kann z.B. sinnvoll sein, wenn Branding-spezifische visuelle Änderungen in die Applikation eingebaut werden sollen wie z.B. ein Firmen-Logo. Ein anderes Beispiel wäre eine Produktfamilie von Rich Clients, wobei bestimmte visuelle Merkmale bei allen Familienmitgliedern gleich sein, sich aber vom Standard Look & Feel unterscheiden sollen. Da NetBeans Swing-basiert ist, kann hier natürlich der „Pluggable Look & Feel“-Mechanismus von Swing voll genutzt werden. Abbildung 6 zeigt beispielhaft den NetBeans-basierten MP3-Manager mit dem Napkin Look & Feel [14]. Man sieht hier sehr schön, dass fast alles angepasst werden kann. Wenngleich Sie Ihre Applikation vermutlich nicht mit dem Napkin Look & Feel ausliefern würden (selbst wenn es

witzig aussieht), könnten Sie Ihrem Chef vielleicht eine Beta-Version Ihrer Software mit dem Napkin Look & Feel zeigen, um die Reaktion „Hey, Ihre Applikation sieht fantastisch aus, morgen liefern wir sie aus!“ zu vermeiden. Ich habe das Napkin Look & Feel hier nur gezeigt, um zu verdeutlichen, wie dramatisch die visuelle Erscheinung einer Applikation mit Swing Look & Feels verändert werden kann.

In der Eclipse-Welt ist die Anpassung des Look & Feels eine andere Sache. Da Eclipse SWT-basiert ist, funktionieren hier Swing-basierte Look & Feels natürlich nicht. Eclipse RCP bietet ein so genanntes Presentation API, mithilfe dessen einiges mit den UI-Elementen gemacht werden kann, die Editoren und Views umgeben. Abbildung 7 zeigt den Eclipse-basierten MP3-Manager in einem neuen

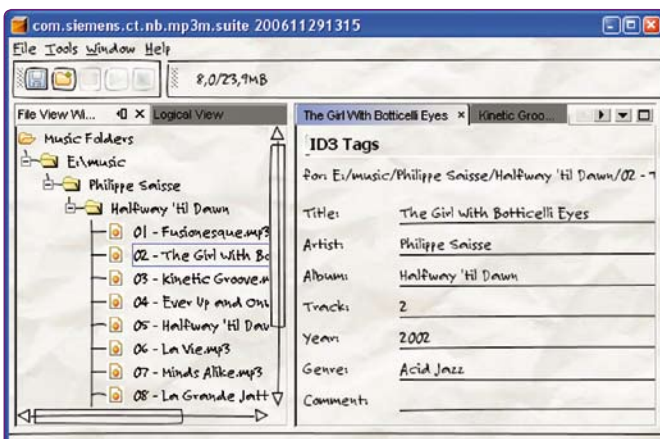


Abb. 6: Der NetBeans-basierte MP3-Manager im Napkin Look & Feel

Listing 13

```
class MyJob implements Runnable, Cancellable {
    private boolean isCancelled;

    public boolean cancel() {
        isCancelled = true;
        return true;
    }

    public void run() {
        // do some initialization
        ProgressHandle handle =
            ProgressHandleFactory.createHandle(NbBundle
                .getMessage(ModelInitializer.class,
                    "Task Name"), this);
        int progress = 1;
        handle.start(musicFolders.size());
        for (int n=0; n<workItemCount; n++) {
            if (!isCancelled) {
                // Abarbeitung eines Work Items
                handle.progress(progress++);
            } else {
                // Hier wegen des Abbruchs aufräumen
                break;
            }
        }
        handle.finish();
    }
}
```

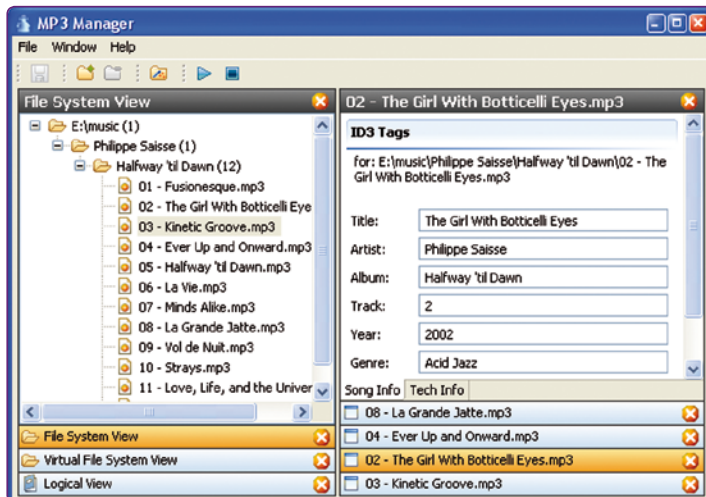


Abb. 7: Der Eclipse-basierte MP3-Manager mit eigener „Presentation“

sammeln, bevor Sie sich für eine Plattform entscheiden. Sowohl Eclipse RCP wie auch die NetBeans Plattform sind sehr ausgereift, bieten eine Menge und helfen Ihnen, bessere Java Rich Clients zu bauen. Aber eins ist sicher: Erfinden Sie nicht das Rad neu, wenn Sie einen Rich Client entwickeln wollen, sondern entscheiden Sie sich für eine existierende Rich-Client-Plattform als Grundlage, wie z.B. Eclipse RCP oder die NetBeans Plattform.



Kai Tödter arbeitet im Fachzentrum Architektur in der Corporate-Technology-Abteilung der Siemens AG. Er beschäftigt sich seit vielen Jahren mit Java-Technologie. Kai war bis ins Jahr 2002 für Siemens in den USA tätig und hat Siemens im Executive Committee des Java Community Process vertreten. Seit mehr als zwei Jahren beschäftigt sich Kai mit verschiedenen Rich-Client-Plattformen. Kontakt: kai.toedter@siemens.com.

Look, den ich mit dem Presentation API realisiert habe. SWT bietet noch eine Reihe von anderen Anpassungsmöglichkeiten, wie z.B. nicht rechteckige Fenster oder „Custom Widgets“. Im Gegensatz zu Swing sind bei SWT die Möglichkeiten der UI-Anpassung eingeschränkt, trotzdem kann zumindest einfach erreicht werden, dass die eigene Applikation nicht so wie das Eclipse SDK aussieht.

Deployment

Beide IDEs stellen Mechanismen bereit, um die Rich-Client-Applikation als allein laufende Applikationen auf das lokale Dateisystem zu bringen. Dann können die Applikationen entweder gezippt werden oder es wird noch ein Installationsprogramm (*Installer*) eingebunden. Neben vielen kommerziellen Installern existieren auch einige sehr gute Open Source Installer. Ein Beispiel wäre das „Nullsoft Scriptable Install System“ [15], oder Sie verwenden Java Web Start als Deployment-Mechanismus. Das ist bei NetBeans bereits eingebaut und bei Eclipse mit etwas Handarbeit oft auch möglich, nämlich dann, wenn alle Plug-ins in signierten JAR-Dateien auslieferbar sind. Man kann sogar den Java-Web-Start-Mechanismus zusammen mit den plattforminternen Update-Möglichkeiten nutzen. Ein Use Case wäre hier, nur die Basisapplikation über Web Start zu installieren und ab dann nur den applikationsinternen Update-Mechanismus zu nutzen, zum Beispiel, um neue Features zu installieren.

Fazit

In diesem und dem vorherigen Artikel wurden viele wichtige Merkmale der Eclipse Rich Client Platform und der

NetBeans Plattform verglichen. Es wurden dafür häufige Anforderungen an einen Rich Client in einer kleinen Demoapplikation auf beiden Plattformen implementiert. Ziel war es jedoch nicht, eine Plus/Minus-Liste zu erstellen und zu beschreiben, welche Dinge besser oder schlechter mit der entsprechenden Plattform realisiert werden können. Dazu müssten die entsprechenden Features noch viel detaillierter beleuchtet werden, als ich es in den Artikeln gemacht habe. Meine Absicht war vielmehr zu zeigen, dass häufig auftretende Use Cases sowohl mit der einen als auch mit der anderen Plattform implementiert werden können. Beide Plattformen bieten eine riesige Menge von wieder verwendbarer Funktionalität und beide können dabei helfen, qualitativ sehr hochwertige Rich-Client-Applikationen zu erstellen. Natürlich musste ich mich auf bestimmte Themen beschränken, und was Sie in meinen beiden Artikeln gelesen haben, ist wirklich nur die Spitze des Eisbergs. Beide Plattformen bieten noch deutlich mehr, aber ich hoffe, dass Sie nun eine Idee davon haben, was die fundamentalen Unterschiede beider Plattformen sind und dass Sie auch festgestellt haben, wie ähnlich die Ansätze beider Plattformen sein können. Wenn Sie mehr über Eclipse RCP erfahren wollen, kann ich Ihnen das Eclipse-Buch von Jeff McAffer und Jean-Michel Lemieux empfehlen [5]. Für die NetBeans Plattform gibt es seit kurzem auch ein gutes Buch von Tim Boudreau, Jaroslav Tulach und Geertjan Wielenga [6].

Wie bereits erwähnt, halte ich es für äußerst wichtig, zuerst möglichst viele Anforderungen für Ihre Applikation zu

>> Links & Literatur

- [1] Wiki über Eclipse RCP: wiki.eclipse.org/index.php/Rich_Client_Platform
- [2] Homepage der NetBeans Plattform: platform.netbeans.org
- [3] Kai Tödter: Das Rad muss nicht neu erfunden werden. Eclipse Rich Client Platform und NetBeans Plattform im Vergleich, *Eclipse Magazin* Vol 12
- [4] Java mp3info: sourceforge.net/projects/mp3info
- [5] Jeff McAffer, Jean-Michel Lemieux: Eclipse Rich Client Platform. Designing, Coding, and Packaging Java Applications, Addison-Wesley 2005
- [6] Tim Boudreau, Jaroslav Tulach, Geertjan Wielenga: Rich Client Programming. Plugging into the NetBeans Platform, Prentice Hall International 2007
- [7] Eclipse Visual Editor Project: www.eclipse.org/vep
- [8] JGoodies Forms: www.jgoodies.com/freeware/forms/
- [9] Michael Valenta: On the Job. The Eclipse Jobs API, www.eclipse.org/articles/Article-Concurrency/jobs-api.html
- [10] JavaHelp System: java.sun.com/products/javahelp/
- [11] Apache Tomcat: tomcat.apache.org
- [12] Jetty Wer-Server: www.mortbay.org
- [13] Lucene-Suchmaschine: lucene.apache.org/java/docs/index.html
- [14] Napkin Look & Feel: napkinlaf.sourceforge.net
- [15] Nullsoft Scriptable Install System: nsis.sourceforge.net/Main_Page