Rich Clients mit Eclipse 3

Eclipse als Plattform für die Entwicklung von flexiblen Rich Client Anwendungen

Martin Lippert, mit Markus Völter

Viele assoziieren mit Eclipse eine umfangreiche integrierte Entwicklungsumgebung für Java, die sich durch Plugins flexibel erweitern lässt. Doch Eclipse ist weit mehr als nur eine erweiterbare IDE – weitaus mehr. Mit der Version 3.0 etabliert sich Eclipse als Plattform für die Entwicklung von Rich-Client-Anwendungen und zeigt eindrucksvoll, dass sich Java-basierte Client-Anwendungen nicht zu verstecken brauchen.

Rich Clients – also nicht browserbasierte Clientanwendungen – hatten in der Java Welt immer schon ein Schattendasein geführt. Dies lag an den folgenden Gründen: Zunächst war da das erste Java basierte GUI Toolkit, awt, welches schlicht daran gescheitert war, dass es zu „altbacken“ aussah und nicht ausreichend viele Widgets zur Verfügung stellte. Um dies zu korrigieren, und um außerdem die Plattformabhängigkeit weiter zu verringern wurde Swing entwickelt. Swing war schon eine ganze Ecke besser, es wurde aber den Ruf der Trägheit nicht los – zum Teil berechtigt. Da Swing außerdem nicht auf den Widgets des darunterliegenden Betriebssystems aufsetzt, war es eben nicht 100% Look-and-Feel kompatibel. Mit einigen wenigen Ausnahmen floppten daher die meisten Java basierten Client Anwendungen.

Mit Eclipse und der darunterliegenden GUI-Bibliothek SWT änderte sich dies radikal. SWT ist schnell und (durch Verwendung der nativen Widgets) Look-and-Feel kompatibel. Außerdem bot die Eclipse Workbench eine Fülle weiter reichender Features (von Abhängigkeitsmanagement durch Plugins über Fenstermanagement bis zu Deployment- und Updateunterstützung) eigentlich alles, was das Rich-Client Entwicklerherz begehrt – mit einer Ausnahme: Man sah der Workbench immer noch an, dass sie eigentlich eine IDE ist. Bestimmte Menüpunkte wurde man nicht los, die Titelzeile enthielt immer „Eclipse“, die Möglichkeiten des Branding waren sehr eingeschränkt.

Mit Eclipse 3.0 wurde die Rich Client Platform (RCP) eingeführt, die die angesprochenen Nachteile behebt. Dabei handelt es sich um eine Zusammenstellung von Basis-Plugins, die neben dem SWT und dem Plugin-Mechanismus die grundlegenden Konzepte einer Eclipse-basierten Anwendung realisieren. Damit ist es erstmals möglich, die gängigen UI-Konzepte von Eclipse (Workbench, Editors, Views, etc.) direkt für eigene Anwendungen zu nutzen. Eclipse kann damit hervorragend als eine Basis für Java-basierte Rich Clients genutzt werden. In diesem Artikel zeigen wir, wie sich Eclipse als Rich-Client-Platform einsetzen lässt und welche Komponenten dazu auf welche Art und Weise genutzt werden können[1].

Rich-Clients auf Basis von Plugins

Wie sämtliche Eclipe-basierten Funktionalitäten beruhen auch Eclipse-basierte Rich Client Anwendungen auf einer Menge von Plugins. Widmen wir uns also zunächst der reinen Plug-In-Technologie, wie sie von Eclipse zur Verfügung gestellt wird. Dieser Mechanismus kann unabhängig von der grafischen Workbench eingesetzt werden, um große Anwendungen aus einer Ansammlung von sauber modularisierten Komponenten, eben Plugins, zusammenzusetzen.

Das komplette Eclipse-SDK zeigt vorbildlich, wie ein großes System aus einer Vielzahl von kleinen Bestandteilen zusammengesetzt werden kann und trotzdem an der Oberfläche einen integrierten Eindruck vermittelt. Hinzu kommt, dass das System relativ einfach und flexibel von

[1] Wir gehen hier davon aus, dass der Leser mit den grundsätzlichen Eclipse-Mechanismen zur Plugin-Entwicklung vertraut ist.

unterschiedlichen Teams erweiterbar ist. Ähnliche Anforderungen werden an Clients moderner und großer Enterprise-Anwendungen gestellt:

- *Strukturierung des Systems:* Der Plug-In-Mechanismus von Eclipse bietet eine Möglichkeit, große Systeme in kleine Bestandteile mit genau definierten Interfaces und Abhängigkeiten zu zerlegen. Eine solche Strukturierung existiert in den meisten großen Enterprise-Anwendungen per se (beispielsweise mittels Layers, Produktbereichen oder ähnlichem). Sie wird aber in den seltensten Fällen konstruktiv umgesetzt. Mit dem Plug-In-Mechanismus haben wir die Möglichkeit, dem Gesamtsystem eine explizite und wohldefinierte Struktur zu geben. Das bedeutet, dass die Abhängigkeiten zwischen den einzelnen Teilen des Systems explizit definiert werden können. Zudem sichert die Runtime von Eclipse zu, dass die definierten Abhängigkeiten zur Laufzeit eingehalten werden.
- *Flexible Erweiterungsmöglichkeiten:* Der Plug-In-Mechanismus von Eclipse bietet mit dem Extension/Extension-Point-Mechanismus eine elegante und einfache Komponententechnologie. Sie erlaubt es, das System an vordefinierten Stellen zu erweitern, ohne das komplette System anpassen zu müssen. Dies kann sowohl auf Ebene des UI geschehen, als auch darunter.
- *Schrittweiser Aufbau:* Indem Plug-Ins selbst Extension-Points definieren können, besteht die Möglichkeit, eine an der Anwendung orientierte Komponentenarchitektur iterativ und inkrementell aufzubauen.

Plugin-Hello-World

Gängige Java-Applikationen besitzen einen Einstiegspunkt, über den diese gestartet werden. Normalerweise gibt es dazu eine `main`-Methode, die von der Java-VM angesprochen wird. Schreiben wir eine Anwendung allerdings auf Basis der Plug-In Runtime, wird beim Start des Systems anstelle der `main`-Methode die Eclipse Runtime von der VM gestartet . Wie bekommen wir die Runtime aber dazu, anstelle der normalen Eclipse-IDE-Workbench unsere Applikation zu starten?

Die Eclipse Runtime verwendet den Extension-Point/Extension-Mechanismus, um die Applikation zu starten. Dazu bietet die Eclipse Runtime den Extension-Point `org.eclipse.core.runtime.applications` an. Dieser ist im Plug-In *org.eclipse.core.runtime* definiert. Für diesen Extension Point ist eine eigene Extension im Rahmen eines zu definierenden Plug-Ins zu erstellen.

Der Extension-Point `applications` definiert für die Erweiterungen das Interface `IPlatformRunnable`. Neben der Definition der Extension in der *plugin.xml* müssen wir also eine Klasse erstellen, die dieses Interface implementiert. Um dies zu tun verwenden wir die entsprechenden Wizards, die uns der *plugin.xml*-Editor zur Verfügung stellt. Mit dem *plugin.xml*-Editor definieren wir zunächst die neue Extension des Extension-Points `applications`. Damit uns der Extension-Wizard diesen Extension-Point anbietet, muss unser Plug-In natürlich das *org.eclipse.core.runtime*-Plug-In importieren.

Auf der Extension-Seite des *plugin.xml*-Editors können wir der neuen Applikation eine ID und einen Namen geben. Wir sollten beide Felder ausfüllen. Das macht es uns später einfach, die Anwendung zu starten.

Die Extension ist allerdings noch nicht komplett fertig. Der Extension-Point applications verlangt von einer entsprechenden Extension, dass sie ein Sub-XML-Element `application` mit wiederum einem Sub-XML-Element `run` besitzt. Wir verwenden dazu das Kontextmenü und erzeugen ein Unterelement `application`, welches ein Unterelement `run` bekommt. Für das `run`-Element können wir nun mit einem Klick auf das Attribut `class` uns eine passende Klasse für die Erweiterung `MyApplication` generieren:

```
package com.entwickler.eclipsebuch.testapplication;

import org.eclipse.core.runtime.IPlatformRunnable;
public class MyApplication implements IPlatformRunnable {
  public MyApplication() {
```

```
  }
  public Object run(Object args) throws Exception {
    return null;
  }
}
```

Die für das Plug-In erforderliche *plugin.xml*-Datei sieht dann folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="testapplication"
   name="testapplication"
   version="1.0.0">

   <runtime>
      <library name="testapplication.jar">
         <export name="*"/>
      </library>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime"/>
   </requires>

   <extension
         id="myapplication"
         name="myapplication"
         point="org.eclipse.core.runtime.applications">
      <application>
         <run
             class="com.entwickler.eclipsebuch.testapplication.MyApplication">
         </run>
      </application>
   </extension>

</plugin>
```

In der *plugin.xml*-Beschreibung ist eine neue Abhängigkeit zum *Runtime*-Plug-In eingefügt und die Extension definiert worden. Diese verweist auf die gerade neu generierte Klasse.

Um die Hello-World-Applikation mit ein wenig Leben zu füllen, müssen wir nun die `run`-Methode der Extension entsprechend anpassen:

```
/**
 * @see IPlatformRunnable#run
 */
public Object run(Object args) throws Exception {
  System.out.println("Hello, World");
  return EXIT_OK;
}
```

Nun müssen wir nur noch dafür sorgen, dass beim Start von Eclipse diese Hello-World Anwendung statt der Workbench gestartet wird. Dazu starten wir Eclipse entweder von der Kommandozeile aus mit folgendem Kommandozeilen-Paramter:

```
-application testapplication.myapplication
```

Oder wir wählen in der entsprechenden Runtime-Workbench-Launch-Konfiguration die zu startende Eclipse-Anwendung einfach aus der Auswahlliste aus. In beiden Fällen wird die Anwendung mit der ID der `application`-Extension angegeben, der die ID des Plug-Ins vorangestellt wird. In unserem Fall also: `testapplication` für das Plug-In und `myapplication` für die Extension.

Jetzt können wir in Eclipse eine Runtime-Workbench starten und sehen die „Hello-World“-Ausgabe in der Konsole. Bevor wir dem Beispiel eine echte grafische Oberfläche hinzufügen, betrachten wir die Plug-Ins, die zum Starten der Hello-World-Applikation benötigt werden. In der Standard-Konfiguration werden alle Plug-Ins der verwendeten Eclipse-Installation verwendet. Wenn wir unsere Business-Anwendung irgendwann einmal an den Kunden ausliefern, soll nicht die komplette Eclipse-IDE-Installation enthalten sein. Wir ändern also unsere Launch-Konfiguration und selektieren nur die folgenden Plug-Ins (neben dem Plug-In unserer Applikation):

- org.eclipse.core.runtime
- org.eclipse.osgi

Diese Plug-In-Zusammenstellung reicht aus, um die Eclipse Runtime der Version 3.0 zu starten und die gewünschte „Hello, World“-Ausgabe zu erzielen.

Eine Konsolenanwendung ist nicht das, was man sich üblicherweise unter einem Rich Client vorstellt. Wir wollen daher im Folgenden eine grafische Oberfläche implementieren.

Prinzipiell lässt sich auf Basis der Eclipse-Plattform, wie wir sie bis jetzt kennengelernt haben, jede Art von grafischer Oberfläche implementieren. Wir können sogar eine Swing-basierte Anwendung aus der run-Methode heraus starten. Allerdings wollen wir uns im nächsten Schritt weitere Features der Eclipse-RCP zu Nutze machen.

Eclipse Rich Client Platform

Während die reine Plug-In Runtime schon seit der ersten Version von Eclipse ohne die IDE-spezifischen Funktionalitäten verwendet werden kann, enthielten die UI-Anteile von Eclipse vor der Version 3 IDE-spezifische Merkmale. Man konnte zwar die Workbench von Eclipse (der eigentliche UI-Rahmen) durch deinstallieren möglichst vieler Plug-Ins soweit wie möglich entkernen, einzelne Eigenschaften der Workbench ließen sich damit allerdings noch nicht entfernen. Das Projekt-Menü war beispielsweise fest in der Workbench verankert. Allerdings möchte man die Anwender einer normalen Business-Anwendung nicht unbedingt mit einem Projects-Menu und einen „Rebuild All“-Eintrag beglücken.

Dies hat sich mit der Version 3.0 von Eclipse verändert. Die grundlegenden Plug-Ins sind komplett von IDE-spezifischen Konzepten befreit worden. Dazu gehört beispielsweise auch die generische Workbench mit ihren Konzepten wie Views und Editoren. Diese Plattform wird im Eclipse Projekt die Eclipse Rich Client Platform genannt.

Mit der Rich Client Platform ist es möglich, deutlich mehr Funktionen der Eclipse Platform für die Entwicklung genereller Business-Anwendungen wieder zu verwenden. Obwohl wir hier nicht alle Möglichkeiten durchspielen können, zeigen wir anhand eines kleinen Beispiels, wie man prinzipiell die Eclipse Rich Client Platform verwenden kann.

Hello, RCP-World

Wir starten mit dem kleinen Beispiel wieder bei der Applikation, die von der Eclipse Runtime gestartet wird. Dazu realisieren wir, wie oben bereits beschrieben, eine Klasse, die `IPlatformRunnable` implementiert und lassen das Plug-In den `applications`-Extension-Point der Runtime erweitern. Im nächsten Schritt implementieren wir die `run`-Methode, indem wir die generische Workbench erzeugen und starten.

```
public class RCPApplication implements IPlatformRunnable {

  public Object run(Object args) throws Exception {
    WorkbenchAdvisor workbenchAdvisor = new RcpWorkbenchAdvisor();
    Display display = PlatformUI.createDisplay();
    int returnCode = PlatformUI.createAndRunWorkbench(display,
                                          workbenchAdvisor);
    if (returnCode == PlatformUI.RETURN_RESTART) {
      return EXIT_RESTART;
    } else {
      return EXIT_OK;
    }
  }
}
```

Wir konzentrieren uns auf den hervorgehobenen Teil des Codes. Die restlichen Implementationen dienen hier nur einer kompletten Darstellung.

Die Workbench selbst kann erzeugt und gestartet werden, indem aus der PlatformUI-Klasse die entsprechende Methode aufgerufen wird. Diese erwartet zwei Parameter:

- *Ein Display:* Stellt das Fenster für die SWT-Oberfläche zur Verfügung und kann einfach über PlatformUI erzeugt werden.

- *Einen Workbench-Advisor:* Dieser muss selbst implementiert werden. Er dient dazu, die Workbench zu konfigurieren. Der spezielle Workbench-Advisor muss die abstrakte Klasse WorkbenchAdvisor erweitern.

```
public class RcpWorkbenchAdvisor extends WorkbenchAdvisor {

  public String getInitialWindowPerspectiveId() {
    return "rcp.perspective1";
  }

  public void preWindowOpen
                    (IWorkbenchWindowConfigurer configurer) {
    super.preWindowOpen(configurer);
    configurer.setShowCoolBar(false);
    configurer.setShowShortcutBar(false);
    configurer.setShowStatusLine(false);
  }
}
```

Der Workbench-Advisor muss die abstrakte Methode `getInitialWindowPerspectiveId` aus der Klasse `WorkbenchAdvisor` implementieren. Hier wird die Id derjenigen Perspective zurückgeliefert, die zu Beginn angezeigt werden soll. In diesem Beispiel haben wir eine eigene Perspective definiert, die wir gleich genauer betrachten werden.

Zusätzlich kann der Workbench-Advisor noch eine Reihe weiterer Operationen aus der Oberklasse erweitern. In diesem Beispiel haben wir die Methode *preWindowOpen* implementiert und schalten in ihr alle von uns nicht gewünschten Eigenschaften der Workbench (wie beispielsweise die Cool-Bar, die Status-Zeile und die Shortcut-Bar) aus.

Als letzten fehlenden Schritt müssen wir noch die oben erwähnte eigene Perspective implementieren. Wir können dies sehr schnell und einfach erledigen, indem wir uns die *plugin.xml*-Datei anschauen und auf dem Extension-Tab eine neue Perspective-Extension definieren und uns die dazu passende Klasse generieren lassen. Heraus entsteht eine leere Implementation einer Perspective.

```
public class RcpPerspective implements IPerspectiveFactory {

  public void createInitialLayout(IPageLayout layout)  {
  }
}
```

Wichtig ist, dass in der *plugin.xml*-Datei die Perspective mit der gleichen ID versehen wird, mit der wir die Perspective in Code der Klasse *RCPWorkbenchAdvisor* identifiziert haben.

Die *plugin.xml*-Datei sieht jetzt folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="rcp"
   name="rcp"
   version="1.0.0">

   <runtime>
      <library name="rcp.jar">
         <export name="*"/>
      </library>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.ui"/>
   </requires>

   <extension
         id="rcpappl"
         name="rcpappl"
         point="org.eclipse.core.runtime.applications">
      <application>
         <run
            class="com.entwickler.eclipsebuch.rcp.RCPApplication">
         </run>
      </application>
   </extension>
   <extension
         id="rcpper"
         name="rcpper"
         point="org.eclipse.ui.perspectives">
```

```
      <perspective
            name="rcp.perspective1"
            class="com.entwickler.eclipsebuch.rcp.RcpPerspective"
            id="rcp.perspective1">
      </perspective>
   </extension>

</plugin>
```

Jetzt kann das Rich Client Beispiel starten. Dazu definieren wir eine neue Launch-Konfiguration im Run-Menü von Eclipse. Als Applikation wählen wir aus der Choicebox wieder unsere Anwendung aus. Zusätzlich lassen wir uns die Liste der benötigen Plugins von Eclipse selbst per Knopfdruck aufwählen.

Starten wir die neu definierte Launch-Konfiguration, sehen wir eine leere Workbench, in die nun eigene anwendungsspezifische Views, Editoren und ähnliches eingebracht werden können.

Menüs

Im nächsten Schritt erweitern wir das Beispiel um eine Menüleiste mit einigen Menü-Einträgen. Dazu erweitern wir zunächst den eigenen Workbench-Advisor und überschreiben die Operation `fillActionBars`. In der Operation bekommen wir als Parameter das *WorkbenchWindow* sowie einen *ActionBarConfigurer*. Mit letzterem können wir eigene Menüs definieren, indem wir uns den *MenuManager* geben lassen. Anschließend können wir dem *MenuManager* mitteilen, welche selbstdefinierten oder vordefinierten Menüs wir nutzen möchten. Der Code dafür sieht recht schlicht aus:

```
public class RcpWorkbenchAdvisor extends WorkbenchAdvisor {

  ...

  public void fillActionBars(IWorkbenchWindow window,
      IActionBarConfigurer configurer, int flags) {
    IMenuManager manager = configurer.getMenuManager();
    manager.add(createFileMenu(window));
  }

  private MenuManager createFileMenu(IWorkbenchWindow window) {
    MenuManager menu = new MenuManager("File");
    menu.add(ActionFactory.QUIT.create(window));
    return menu;
  }
```

In der Methode `createFileMenu` erzeugen wir zunächst ein *MenuManager* für das Menü mit dem Namen „File“. Anschließend fügen wir einen vordefinierten Menüeintrag hinzu. Als Beispiel nutzen wir hier den *Exit*-Eintrag, der klassischerweise am Ende des File-Menüs steht und dazu dient, die gesamte Applikation zu beenden. Da der *Exit*-Menüeintrag vordefiniert ist, erzeugen wir uns einfach ein Exemplar dieses Eintrages und fügen ihn unserem Menü hinzu.

Als Menüeinträge erwartet die Eclipse-Plattform generell so genannte Actions. Eine Action ist jenseits der Menüs ein grundlegendes Konzept in der Eclipse-Plattform. Es wird auch verwendet, um beispielsweise Knöpfe in Toolbars zu realisieren. Für Actions gibt es entsprechende Interfaces und Oberklassen.

Diese wollen wir im nächsten Schritt nutzen. Um dem File-Menü einen selbstdefinierten Eintrag hinzuzufügen, müssen wir zunächst einmal eine Action implementieren. Dazu erben wir im einfachsten Fall von der Oberklasse *Action* und überschreiben die *run*- und die *getText*-Methode.

```
public class SayHelloAction extends Action {

  public String getText() {
    return "MyAction";
  }

  public void run() {
      MessageDialog.openInformation(
                            PlatformUI.getWorkbench().
                            getActiveWorkbenchWindow().
                            getShell(),
                            "Title", "This is my message!");
```

```
  }
}
```

Die *getText*-Methode gibt den Namen der Action zurück. Dies ist auch der Name, der später im Menü angezeigt wird. Die *run*-Methode implementiert die eigentliche Aktion, die ausgeführt werden soll. In diesem Beispiel nutzen wir einen einfachen Informationsdialog.

Als nächstes müssen wir die Action noch in das *File*-Menü eintragen. Dazu gehen wir zurück in die entsprechende Methode in unserem *WorkbenchAdvisor* und ergänzen den Eintrag in der Methode, die das File-Menü erzeugt.

```
public class RcpWorkbenchAdvisor extends WorkbenchAdvisor {

  ...

  public void fillActionBars(IWorkbenchWindow window,
      IActionBarConfigurer configurer, int flags) {
    IMenuManager manager = configurer.getMenuManager();
    manager.add(createFileMenu(window));
  }

  private MenuManager createFileMenu(IWorkbenchWindow window) {
    MenuManager menu = new MenuManager("File");
    menu.add(new SayHelloAction());
    menu.add(new Separator());
    menu.add(ActionFactory.QUIT.create(window));
    return menu;
  }
```

Es können auch vordefinierte Menü-Einträge von so genannten *ContributionItems* wiederverwendet werden. Wir wollen beispielsweise ein Menü für Arbeitskontexte definieren, indem wir die Perspective-Auswahl wieder verwenden wollen. Dazu implementieren wir ein Menü, welches als Untermenü ein vorgefertigtes *ContributionItem* für die Auswahl einer Perspective nutzt.

```
  public void fillActionBars(IWorkbenchWindow window,
        IActionBarConfigurer configurer, int flags) {
    IMenuManager manager = configurer.getMenuManager();
    manager.add(createFileMenu(window));
    manager.add(createWorkContextsMenu(window));
  }

  private MenuManager createWorkContextsMenu(
      IWorkbenchWindow window) {
    MenuManager menu = new MenuManager("WorkContext");
    MenuManager workContextMenu =
      new MenuManager("Switch to Work Context");
    workContextMenu.add(
      ContributionItemFactory.PERSPECTIVES_SHORTLIST.
      create(window));
    menu.add(workContextMenu);
    return menu;
  }
```

Das bis jetzt vorgestellte Grundgerüst kann nahezu beliebig ausgebaut werden. Als wesentliche Elemente einer RCP-Anwendung kommen dann natürlich Views und Editors hinzu. Deren Konstruktion unterscheidet sich allerdings nicht von der Konstruktion von Views und Editors für das Eclipse-SDK.

Strukturierung von Anwendungen

Nachdem wir nun gesehen haben, wie sich Anwendungen auf Basis des Plug-In-Mechanismus von Eclipse entwickeln lassen, untersuchen wir im Folgenden die Auswirkungen dieser Technologie auf Enterprise-Anwendungen. Schließlich stellt sich die Frage, wie Enterprise-Anwendungen aus Plug-Ins zusammengesetzt werden könnten.

Wir nehmen uns wieder das Eclipse-System selbst als Vorbild. Dort erkennen wir beispielsweise, dass die Grundkonzepte und Grundbegriffe, aus denen die Anwendung zusammengesetzt werden soll, in eigenen Plug-Ins definiert sind. So wird beispielsweise festgelegt, dass die Oberfläche einer Eclipse-Anwendung im Wesentlichen aus Views und Editoren besteht, die in Perspektiven angeordnet werden können. Die IDE-spezifischen Teile

von Eclipse basieren auf *Resources*, die jeweils für den konkreten Einsatzkontext spezialisiert werden.

Das Übertragen dieser Kernidee auf Enterprise-Anwendungen führt zu der Überlegung, aus welchen grundsätzlichen Bestandteilen das System zusammengesetzt werden soll. Damit definieren wir die Grundbegriffe der Architektur der Enterprise-Anwendung. Beispielsweise könnten wir für ein Bankensystem grundlegende Begriffe wie Bankprodukt, Person oder Konto identifizieren. Definieren wir entsprechende Spezialisierungen wie Bauspardarlehen, Kreditkunde oder Girokonto, können wir auch dazu passende UIs und aufgabenbezogene Werkzeuge finden, während für das allgemeine Konzept einer Person ggf. ein allgemeiner Personensucher im System realisiert werden kann.

Das Verteilen der skizzierten Konzepte auf verschiedene Plug-Ins führt zum folgenden Aufbau des Systems:

- *Produkt-Plug-In:* Definiert die Grundkonzepte für Bankprodukte und bietet einen Produktordner an, mit dem alle verfügbaren Produkte nach verschiedenen Kategorien sortiert angeschaut und durchsucht werden können.
- *Personen-Plug-In:* Definiert die Grundkonzepte für eine Person im Rahmen einer Bank und liefert fertige Implementierungen für einen Personensucher.
- *Konten-Plug-In:* Definiert die Grundkonzepte für ein Konto in der Bank.
- *Arbeitsplatz-Plug-In:* Definiert den Rahmen für einen grafischen Arbeitsplatz, an dem der Mitarbeiter in der Bank mit den verschiedenen Werkzeugen (oder Views/Editoren) arbeiten kann.
- *Bauspar-Plug-In:* Liefert Erweiterungen für die Person, definiert eine Reihe spezialisierter Produkte und erlaubt, Produkte zu einer Bausparfinanzierung zusammenzusetzen.
- …

Das Beispiel erhebt keinen Anspruch auf fachliche Integrität und könnte in einem wirklichen Bankenprojekt natürlich völlig anders aussehen. Wir wollen mit dem Beispiel nur die grundsätzlichen Möglichkeiten der Plug-In-Technologie illustrieren. In dem skizzierten Fall könnten beispielsweise auch Verbundpartner mit ihren Produkten sehr einfach in das System integriert werden. An der Oberfläche würde sich das System stets wie aus einem Guss präsentieren.

Class-Loading: Tipps, Tricks und mögliche Fallen

Der Aufbau von Business-Anwendungen aus Plug-Ins auf Basis der Eclipse Runtime erzwingt, dass die Anwendung bestimmte Regeln und Rahmenbedingungen einhält. Die Regeln werden durch die Eclipse Runtime definiert und müssen eingehalten werden. Die wichtigsten Regeln werden wir im Folgenden auflisten und eine Reihe typischer Probleme und dazu passender Lösungen vorstellen.

Eclipse erzeugt für jedes Plug-In einen eigenen Classloader. Damit erreicht die Eclipse Runtime, dass die Namensräume verschiedener Plug-Ins klar voneinander getrennt und selbst mehrere Versionen unterschiedlicher Bibliotheken von unterschiedlichen Plug-Ins verwendet werden können, ohne dass sich diese Libraries gegenseitig stören. Die Eclipse Runtime wird daher auch manchmal als ein Class-Loader-Framework bezeichnet.

Die speziellen Classloader haben normalerweise für die Plug-Ins keine besonderen Auswirkungen. Allerdings müssen die Plug-Ins darauf ausgelegt sein, anstelle von den Standard-ClassLoadern des JDK von einem speziellen Classloader geladen zu werden. Zugriffe auf die Methode *ClassLoader.getSystemClassLoader* sollten daher innerhalb eines Plug-Ins nicht stattfinden (da der *SystemClassLoader* nicht den Code der einzelnen Plug-Ins laden kann).

Für selbstentwickelte Plug-Ins ist es normalerweise auch nicht nötig, auf einen Classloader explizit zuzugreifen. Müssen spezielle Klassen dediziert geladen werden, kann dies in Eclipse 3.0 auch über die Bundles geschehen. Ein Problem kann aber dann wieder auftreten, wenn Third-Party-Libraries eingesetzt werden. Verwendet eine Third-Party-Library direkt den *SystemClassLoader* und wird in ein Plugin verpackt, kann der *SystemClassLoader* nicht mehr

die Klassen der Library zugreifen. Ein expliziter Aufruf der Library auf den *SystemClassLoader*, um eine eigene Klasse zu laden, wird fehlschlagen. Ob ein solcher Fall auftritt, hängt natürlich stark davon ab, wie die Library implementiert ist und wozu sie den *SystemClassLoader* nutzt. Wenn man Zugriff auf den Quellcode der Library hat, lässt sich das Problem in der Regel sehr einfach beheben, indem der Aufruf des SystemClassLoaders durch einen Aufruf des eigenen Plug-In-Classloaders ersetzt wird. Anstelle von

```
ClassLoader.getSystemClassLoader()
```

schreiben wir

```
this.getClass().getClassLoader()
```

Damit bekommen wir den Classloader, mit dem die Klasse von this geladen worden ist, also den Classloader des Plugins, welches die Klasse von this beherbergt.

Unangenehmer wird das spezielle Classloading von Eclipse, wenn die externen Libraries Klassen per Java-Reflection erzeugen müssen. Dies ist beispielsweise in vielen Persistenz-Frameworks, wie Hibernate oder JDO, der Fall, aber auch schon bei der einfachen Serialisierung von Java. In solchen Fällen kann man sich entweder mit dem Setzen des „richtigen“ Classloaders am Thread (Thread.setContextClassLoader) behelfen (funktioniert beispielsweise bei einigen JDO-Implementationen), oder die Library bietet (hoffentlich) einen entsprechenden Hook an, um Klassen zu finden (wie bei der Serialisierung von Java).

Fazit

Eclipse hat mit der Version 3.0 einen sehr großen Sprung gemacht. Wie IBM bereits mit der Lotus-Client-Workplace-Technologie demonstriert, lassen sich auf Basis der Eclipse-Rich-Client-Plattform mächtige Anwendungen realisieren. In eigenen Projekten nutzen wir neben der kompletten RCP auch gerne einmal nur den Plugin-Mechanismus, um große Anwendungssysteme zu strukturieren und gelangen so zu stabilen Systemarchitekturen, die sich flexibel erweitern lassen und gleichzeitig gut handhabbar sind. Aus unserer Sicht ist die Eclipse-Technologie nicht mehr wegzudenken.

Autor

Martin Lippert ist langjähriger Consultant und Coach in den Bereichen agile Methoden, Software-Architekturen und Java. Er berät Entwicklungsorganisationen bei der Einführung und Anwendung agiler Entwicklungsmethoden und hilft Teams, neue Wege in der Softwareentwicklung zu bestreiten. Neben großen und komplexen Refactorings gehört die Entwicklung großer plugin-basierter Anwendungssysteme auf Basis der Eclipse-Plattform zu seinen Spezialgebieten.

E-Mail: lippert@acm.org
http://www.martinlippert.com/

Links und Literatur

- http://www.eclipsepowered.org/: Blog von Ed Burnette mit vielen Tipps und Tricks für RCP-Anwendungen.
- http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/rcp/index.html: Die offizielle Webseite zur Eclipse Rich Client Platform. Enthält neben einen Überblick auch ein mehrteiliges Tutorial sowie Verweise auf Beispiele.
- http://www.martinlippert.com/: Refactorings rund um Eclipse-Plugin- und -RCP-basierte Anwendungen.

- Kai Brüssau/Oliver Widder (Hrsg.), Herbert Brückner, Roger Butenuth, Martin Lippert, Matthias Lübken, Birgit Schwartz-Reinken, Markus Völter, Lars Wunderlich: Eclipse – Die Plattform: J2EE-Development, Framework Eclipse, Rich Clients mit Eclipse. Software und Support Verlag, 2004.