Architektur und Konzepte von Eclipse 3

Ausarbeitung im Rahmen des Seminars

Komponentenbasierte Softwareentwicklung

Prof. Klaus-Peter Löhr, Karl Pauls
Freie Universität Berlin
WS 2004/05

von
Tilman Walther
walther@inf.fu-berlin.de

Basis-URL: http://www.tilman.de/uni/Eclipse3.pdf

Inhaltsverzeichnis

1 Entwicklung von Eclipse

Die Eclipse Plattform entstand als Nachfolger der Entwicklungsumgebung Visual Age von IBM. Ziel war es, eine leicht erweiterbare Entwicklungsumgebung für jede erdenkliche Anwendung zu schaffen. Der Gedanke dahinter: Eine offene Plattform sollte ermöglichen, sämtliche Werkzeuge "unter einem Dach" zu integrieren, um den Aufwand für Portierung und Kommunikation bei der Entwicklung zu minimieren (1; 2).

Eclipse wurde hauptsächlich durch die 1996 von IBM gekauften OTI Labs entwickelt, die vorher bereits an Visual Age for Java und anderen IBM Entwicklungswerkzeugen gearbeitet hatten. Parallel zum Erscheinen von Version 1.0 und der Freigabe als Open Source im November 2001 gründete IBM die Eclipse Community, die eine breite Unterstützung der Plattform sichern und die Anzahl der verfügbaren Plugins schnell vergrößern sollte. Neben IBM waren bei der Gründung Borland, Merant, QNX, Rational, Red Hat, SuSE, TogetherSoft und Webgain beteiligt (3, S. 5). Hier zeigt sich eine Besonderheit im Gegensatz zu anderen Open-Source-Projekten: Von Anfang an war es das erklärte Ziel, neben einer Open-Source-Entwicklergemeinde auch ein "funktionierendes Ökosystem" von kommerziellen Anbietern zu etablieren.

Im März 2003 erschien die Version 2.1 von Eclipse, die den endgültigen Durchbruch als Java IDE bringen sollte. Im Februar 2004 wurde die Eclipse Community schließlich in eine gemeinnützige Organisation umgewandelt, ein halbes Jahr bevor man im Juni die aktuelle Version 3.0 freigab. Diese brachte grundlegende Änderungen in der Architektur der Plattform: Die interne Plugin-Verwaltung wurde komplett auf eine eigene OSGi-Implementierung aufgebaut, so dass Plugins nun zur Laufzeit ge- und entladen werden können. Zusammen mit einer neuen Aufteilung der verschieden Komponenten, die den IDE-spezifischen Teil sauber von der Funktionalität als Plattform trennt, eröffnet die überarbeitete Archtektur völlig neue Möglichkeiten für Eclipse als Basis von Rich-Client-Anwendungen.

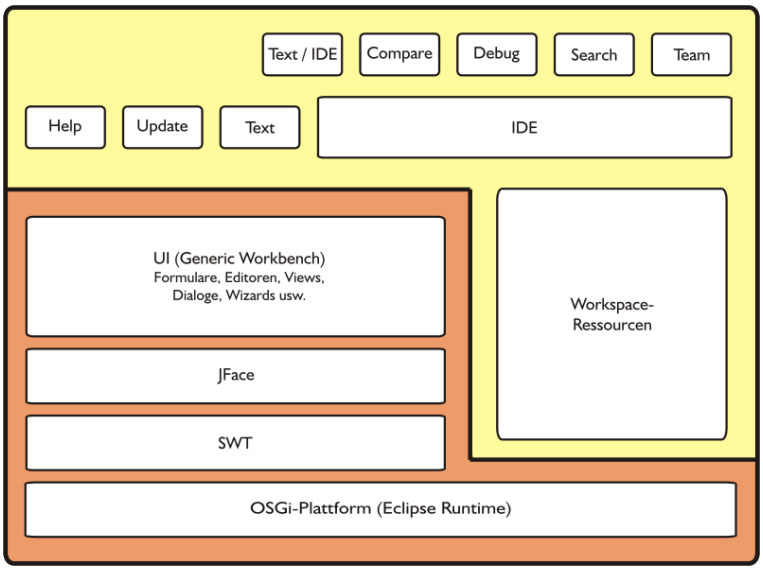

Abb. 1: Die Eclipse Platform

2 Ein Überblick

Die Eclipse Plattform bietet auf verschiedenen Ebenen Möglichkeiten zur Erweiterung und Nutzung der gelieferten Funktionalität. Eclipse selbst kann über *Plugins*[1] erweitert werden, um speziellen Bedürfnissen als Entwicklungswerkzeug zu genügen. Ebenfalls möglich ist die Nutzung von Eclipse als Plugin-Verwaltung, um beliebige Programme komponentenbasiert erweiterbar zu gestalten. Des weiteren können, aufbauend auf die neue *Rich Client Platform*, Eclipse-basierte Anwendungen erstellt werden, die die verschiedenen Elemente der Plattform nutzen. Und nicht zuletzt bieten die Bibliothek des *SWT* und das Framework *JFace* die Option, losgelöst von der Eclipse Plattform bei der Erstellung von Benutzeroberflächen eingesetzt zu werden.

Auch wenn all diese Möglichkeiten im Rahmen dieser Arbeit natürlich bei weitem nicht eingehend behandelt werden können, soll doch ein möglichst umfassender Überblick geliefert werden. Auf diese Weise können die Konzepte hinter Eclipse am besten verstanden und die Möglichkeiten, die sich daraus ergeben erkannt werden. Das Augenmerk liegt dabei hauptsächlich auf der Plugin-Architektur, die einen fast schon radikal zu nennenden kom-

[1] Um die Lesbarkeit zu verbessern, wird im weiteren Verlauf die immer gebräuchlichere Schreibweise "Plugin", anstelle der formaleren Variante mit Bindestrich gebraucht.

pontenbasierten Ansatz verfolgt, dem - zu Recht - nach wie vor steigende Aufmerksamkeit zuteil wird.

Zunächst werden jedoch die Klassenbibliothek SWT und das zugehörige Framework JFace betrachtet, da sie losgelöst von der Plattform eingesetzt werden können und auch konzeptionell in keinem direkten Zusammenhang mit der Plugin-Architektur oder der Rich Client Platform stehen.

3 Frameworks und Bibliotheken

3.1 SWT

Eclipse ist in Java geschrieben, nutzt aber mit dem *Standard Widget Toolkit (SWT)* ein eigenes Framework für das User Interface, das - anders als Java Swing - auf die GUI-Elemente des Betriebssystemwidgets zurückgreift, wodurch auch die komplex verschachtelte Eclipse GUI noch relativ performant läuft. Allerdings bedingt diese Tatsache auch, dass Eclipse nicht der Java-Doktrin *"Code once, run everywhere"* folgt, da das SWT nicht für sämtliche Plattformen portiert wird, auf denen Java läuft. In der Praxis stellt dies allerdings nur eine geringe Einschränkung dar, da die "drei Großen" im Desktopbereich (Windows, Linux und OS X) unterstützt werden.

Das SWT bildet eine dünne Schicht, die GUI-Aufrufe über das *Java Native Interface* direkt an das Betriebssystem weiterleitet (4, S. 146). Dies bedingt, dass man - für Java-Programmierer ungewohnt - Speicherplatz manuell freigeben muss, wenn Widgets nicht mehr benötigt werden. Allerdings verwaltet das SWT die Elemente in einer Baumstruktur: Wird ein Elternknoten gelöscht, wird auch der Speicher der untergeordneten Elemente freigegeben.

Im Gegensatz zum Java-eigenen *Abstract Windowing Toolkit* muss man sich beim SWT nicht mit dem kleinsten gemeinsamen Nenner von Widgets für sämtliche unterstützten Plattformen begnügen. Elemente, die auf einer Plattform nicht existieren, werden entweder emuliert oder sind schlicht nicht benutzbar (5). Auch hier ist SWT deutlich von *"Code once, run everywhere"* entfernt, die Zielplattform sollte zum Zeitpunkt der Programmierung bekannt sein.

Zur Zeit gibt es die Bilbliothek für Windows (98 bis XP und CE), Linux (mit GTK+ 2.2 oder Motif), MacOS X, QNX und verschiedene Unix-Varianten. Sie kann als Jarfile

zusammen mit betriebssystemspezifischen Libraries in Anwendungen eingebunden werden und vergrößert diese um ein bis zwei Megabyte.

3.2 JFace

Da die Programmierung von SWT (ähnlich wie bei Swing) die Trennung von Modell und Darstellung stark erschwert, gibt es die Erweiterung JFace.

JFace ist ein Framework, das *Viewer* als Abstraktion von komplexen SWT-Komponenten wie Dialogen und Wizards bereitstellt. Außerdem verfügt JFace über ein eigenes Ereignismodell für Viewer, mit dem verschiedene Viewer nach dem *MVC*-Prinzip verwendet werden können: Viewer sind keine reinen Repräsentationsobjekte der GUI, sondern verwalten auch ihre Dateninhalte wie z.B. Tabelleneinträge, aber eben streng getrennt in *Model* und *View*. Das hat zum Beispiel den Vorteil, dass verschiedene Viewer auf ein Modell angewendet werden können.

Mit Hilfe von JFace lässt sich das SWT auch gut für komplexe Benutzeroberflächen einsetzen.

3.3 Andere Teile der Plattform

Neben dem SWT und JFace können natürlich auch die anderen Teile der Plattform (wie in Abb. 1 dargestellt) in eigene Anwendungen eingebaut werden. Allerdings sind diese in der Regel nicht eigenständig, sondern bauen auf andere Teile der RCP auf, so dass sie sich sinnvoll nur in Programmen verwenden lassen, die komplett auf Basis von Eclipse laufen.

4 Die Architektur von Eclipse

4.1 Exkurs: OSGi

Wie bereits erwähnt, wurde mit der Version 3 die Plugin-Verwaltung von Eclipse auf Basis von OSGi umgestellt. Doch was ist OSGi?

Die *Open Services Gateway Initiative* ist ein Industriekonsortium, das eine Softwareplattform für die Verwaltung von Diensten auf Komponentenbasis standardisiert. Das ursprüngliche Einsatzgebiet von OSGi ist die Automatisierung von elektronischen Geräten in Haushalten und Fahrzeugen, das Modell passt aber so gut für Eclipse, dass man sich entschloss, das proprietäre Komponentenmanagement von Eclipse 2 durch eine eigene OSGi-Implementierung zu ersetzen.

OSGi-Komponenten (sog. *Bundles*) sind, wie Eclipse, in Java geschrieben und bieten Dienste an, die über einen OSGi-Server geliefert werden. Dieser hält Informationen über sämtliche verfügbaren Dienste vor und übernimmt die Verwaltung des Lebenszyklus.

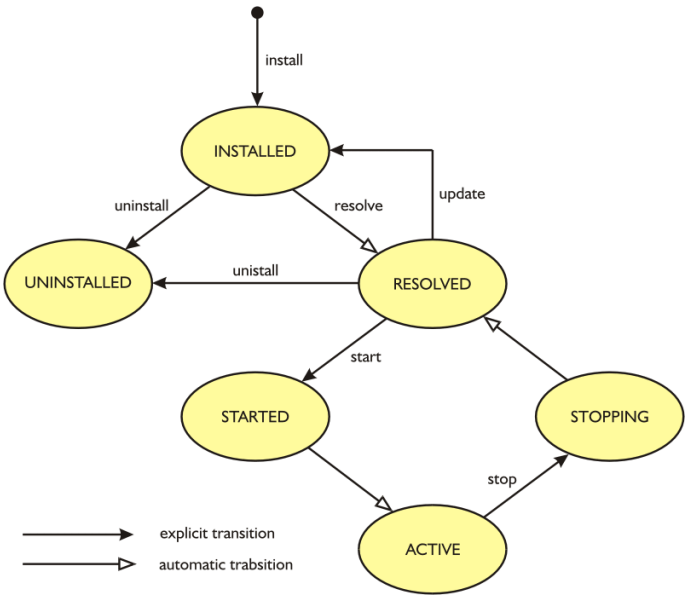

Abb. 2: Zustandsdiagramm OSGi-Bundles

In Eclipse erfüllt die Laufzeitumgebung die Aufgabe des OSGi-Servers, genauer das Plugin *org.eclipse.osgi*, das Teil der Laufzeitumgebung ist. Die Eclipse-Plugins bilden das Äquivalent zu den Bundles von OSGi und bekommen vom Laufzeitsystem mit der Plugin Registry eine Anlaufstelle für das Auffinden und Registrieren von Diensten zur Verfügung gestellt.

4.2 Das Plugin-System von Eclipse

Seit der ersten Version verfolgt Eclipse das Konzept einer reinen Plugin-Architektur. Das bedeutet, dass nicht nur eine bestimmte Funktionalität über Plugins erweitert werden kann,

sondern praktisch jeder Teil von Eclipse ein Plugin darstellt. Dieser streng komponentenbasierte Ansatz[2] bietet größtmögliche Flexibilität für den Ausbau der Plattform und die Erweiterung um einzelne Features, sei es durch das Entwicklungsteam von Eclipse oder durch die Anwender selbst.

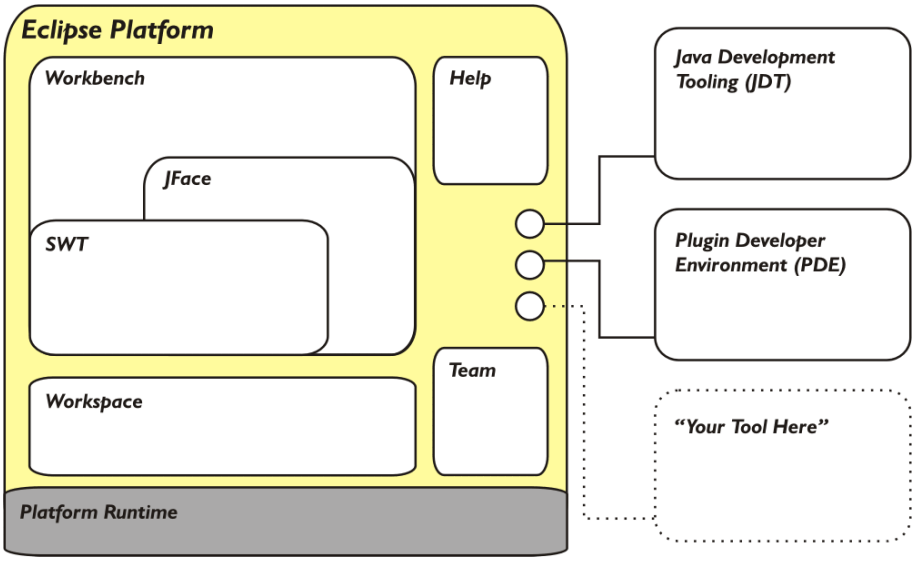

Abb. 3: Erweiterung der Plattform

Eclipse wird als Software Development Kit mit zwei Plugins ausgeliefert: Dem *Java Development Tooling (JDT)*, das Eclipse zur Java IDE erweitert und dem *Plugin Developer Environment (PDE)*, mit dem man neue Plugins für Eclipse entwickeln kann. Damit ist Eclipse "nach dem Auspacken" eine vollständige Entwicklungsumgebung und liefert bereits alles, was man braucht um eigene Erweiterungen vorzunehmen. Die beiden mitgelieferten Komponenten sollen neben der Funktionalität, die sie bereitstellen auch als Beispiel und Vorlage für eigene Plugins dienen.

Plugins werden über *Extension Points* verschachtelt. Ein Plugin baut eine Menge von Erweiterungen *(extensions)* auf einen oder mehrere Extension Points auf. Die erweiterten Plugins befinden sich dabei auf derselben oder einer untergeordneten Ebene der Plugin-Hierarchie; normalerweise können mehrere Plugins an einen Extension Point angeschlossen werden.

[2]Im Folgenden werden die Begriffe Komponente und Plugin synonym verwendet, da dies dem Komponentenbegriff für Eclipse weitestgehend entspricht.

Die Extension Points, die ein Plugin zur Verfügung stellt, werden in einem Manifest beschrieben. Auf diese Weise ist das Laufzeitsystem in der Lage, sämtliche Verbindungen zwischen den verschiedenen Komponenten nachzuvollziehen, ohne Exemplare von ihnen erzeugen zu müssen - Plugins werden erst in den Speicher geladen, wenn ihre Funktionalität benötigt wird. Dieses verzögerte Laden ermöglicht es, Eclipse-basierte Anwendungen mit einer nahezu unbegrenzten Anzahl von Plugins zu erweitern, da eine installiertes Erweiterung zunächst so gut wie keine Ressourcen benötigt. Da Plugins über die OSGi-Registry auch wieder entladen werden können, lassen sich Anwendugen auf eine hochgradig flexible Weise mit Plugin-Funktionalität ausstatten.

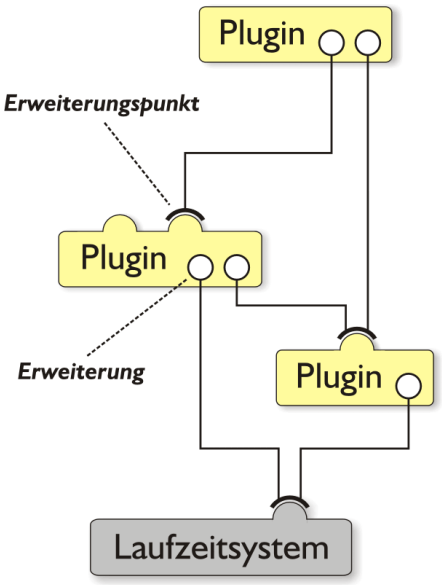

Abb. 4: Plugin-Verschachtelung

Einen Wermutstropfen bringt diese Technik allerdings mit sich: Einmal deklarierte Extension Points können bei der Eclipse-eigenen OSGi-Implementierung nicht wieder entfernt werden. Das wird für die meisten Anwendungsgebiete zunächst keinen Nachteil darstellen, ist aber eine deutliche Einschränkung des Dienstkonzepts von OSGi, da sich Extension Points "ansammeln" können.

Die Wurzel der Plugin-Architektur bildet das von der Laufzeitumgebung gelieferte Plugin *org.eclipse.core.runtime*, die als "halbes Plugin" auf keine Extension Points aufbaut, sondern sie nur bereitstellt (6). (Hier ist zwischen der Plugin *Runtime*, der Laufzeitumgebung, und dem von ihr zur Verfügung gestellte Plugin *org.eclipse.core.runtime* zu unterscheiden. Tatsächlich stellt die Laufzeitumgebung nämlich zwei Plugins bereit, auf die alle anderen Plugins aufbauen: *org.eclipse.osgi* und *org.eclipse.core.runtime*.)

Die Möglichkeiten sollen nun an einem Beispiel verdeutlicht werden. Im Interesse der Verständlichkeit soll es sich dabei nicht um eine "echte" Erweiterung der Eclipse Platform handeln, sondern um eine einfache Anwendung, die mit Hilfe von Eclipse nachträglich mit Plugin-Fähigkeit ausgestattet wird. Die grundlegenden Konzepte des Plugin-Systems werden allerdings genauso gut deutlich.

4.3 Ein Beispiel

Wir gehen davon aus, das ein Programm vorliegt das Bilddateien anzeigen kann. Dies soll hier durch die einfache Ausgabe eines Strings repäsentiert werden:

```
1 public class Picster {
2   public static void main(String[] args) {
3     String bild = "Ein Bild";
4     System.out.println(bild);
5   }
6 }
```

Um dieses Programm nun mit Plugins erweitern zu können, muss es das Interface *org.eclipse.core.runtime.IPlatformRunnable* implementieren, dessen `run`-Methode die Funktion der `main`-Methode übernimmt. Damit wird es zum Plugin, das die Runtime am Extension Point *org.eclipse.core.runtime.applications* erweitert:

```
1 import org.eclipse.core.runtime.*;
2
3 public class Picster implements IPlatformRunnable {
4   public Object run(Object args) throws Exception {
5     String bild = "Ein Bild";
6     System.out.println(bild);
7     return EXIT_OK;
8   }
9 }
```

Das Programm muss nun noch einen Extension Point zur Verfügung stellen, an dem die Plugins andocken können. Dieser muss in der Datei *plugin.xml* beschrieben werden:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?eclipse version="3.0"?>
3 <plugin id="picster" name="Picster" version="3.0.0">
4   <!-- benötigte Plugins: Runtime mit OSGi-Server -->
5   <requires>
6     <import plugin="org.eclipse.core.runtime"/>
7   </requires>
8
```

```
9    <!-- Eclipse wird durch das Programm am Extension Point
        applications erweitert -->
10   <extension id="application" point="org.eclipse.core.runtime.
        applications">
11     <application>
12       <run class="picster.Picster"/>
13     </application>
14   </extension>
15
16   <!-- Defninition des neuen Extension Point -->
17   <extension-point id="filter" name="Filter"/>
18 </plugin>
```

Führt man das Programm nun aus, so hat sich an der Funktionalität zunächst nichts geändert. Um es nun zu erweitern, definieren wir zunächst ein Interface über das Bilder an Plugins weitergereicht werden, die es bearbeiten sollen und ein zugehöriges Plugin:

```
1  // Über die filter-Methode des Interface wird
2  // das imaginäre Bild an das Plugin übergeben
3  public interface PluginFilter {
4    String filter(String s);
5  }
6
7  // Der Rotationsfilter dreht das imaginäre Bild
8  public class RotationFilter implements PluginFilter {
9    public String filter(String bild) {
10     System.out.println("RotationFilter arbeitet");
11     return bild + " gedreht";
12   }
13 }
```

Das Plugin wird ebenfalls über *plugin.xml* der Registry bekannt gemacht, indem folgende Zeilen in das `plugin`-Tag eingefügt werden:

```
1    <extension id="movingextensions" point="picster.filter">
2      <filter class="picster.RotationFilter"/>
3    </extension>
```

Auf die gleiche Weise können nun beliebig viele Erweiterungen an den Extension Point angeschlossen werden, die übergebene Bilder modifizieren und zurückliefern.

Nun sind die Grundlagen vorhanden, um im ursprünglichen Programm auf Plugins zurückzugreifen. Dafür müssen verfügbare Plugins von der Registry geholt und das "Bild" an sie übergeben werden. Die endgültige Version des Programms sieht dann so aus:

```
1  public class Picster implements IPlatformRunnable {
2    public Object run(Object args) throws Exception {
3      String bild = "Ein Bild";
4
5      // Referenz auf die Registry holen
6      IExtensionRegistry registry = Platform.getExtensionRegistry();
7
8      // verfügbare Plugins für den Extension Point holen
9      IConfigurationElement[] plugins = registry.
          getConfigurationElementsFor("picster.filter");
10
11     // das Bild an die Plugins zur Bearbeitung übergeben
12     for (int j = 0; j < plugins.length; j++) {
13       PluginFilter filter = (PluginFilter) plugins[j].
            createExecutableExtension("class");
14       bild = filter.filter(bild);
15     }
16
17     System.out.println(bild);
18     return EXIT_OK;
19   }
20 }
```

Das Programm übergibt nun Bilder vor dem Anzeigen an die vorhandenen Plugins, so dass sie gedreht (gespiegelt, skaliert...) werden. Durch die Erweiterung der Runtime konnte diese Funktionalität mit minimalen Änderungen vorgenommen werden. Der ursprüngliche Quellcode von derart überarbeiteten Programmen bleibt praktisch komplett erhalten, nur die zusätzliche Funktionalität muss implementiert werden. Dabei führt der OSGi-Server das gesamte Management im Hintergrund durch, so dass sich die Arbeit auf die Implementierung der gewünschten Funktionen beschränkt.

4.4 Rich Clients auf Basis von Eclipse

Die *Rich Client Platform (RCP)*, die mit Eclipse 3 eingeführt wurde, bildet ein generisches Framework für die Entwicklung von klientenzentrierten Anwendungen. Zwar konnten auch die vorhergehenden Versionen von Eclipse schon als Basis für eigene Anwendungen genutzt werden (7), jedoch war die Funktionalität der Eclipse IDE nicht klar von der Funktionalität des User Interface getrennt. Eclipse-basierte Rich Clients waren stets um die Workbench herum aufgebaut und ließen sich nicht von sämtlichen IDE-spezifischen Bedienelementen befreien, zum Beispiel war das *Projects*-Menü fest in der Benutzeroberfläche eingebettet. Dies ließ Eclipse als Basis für viele Rich Client Anwendungen uninteressant werden (8). Allerdings konnte man schon seit der ersten Version die Plugin Runtime nutzen, um beliebige Anwendungen um Plugin-Funktionalität zu erweitern, wenn auch ohne das mit Version 3 eingeführte "hot plugging".

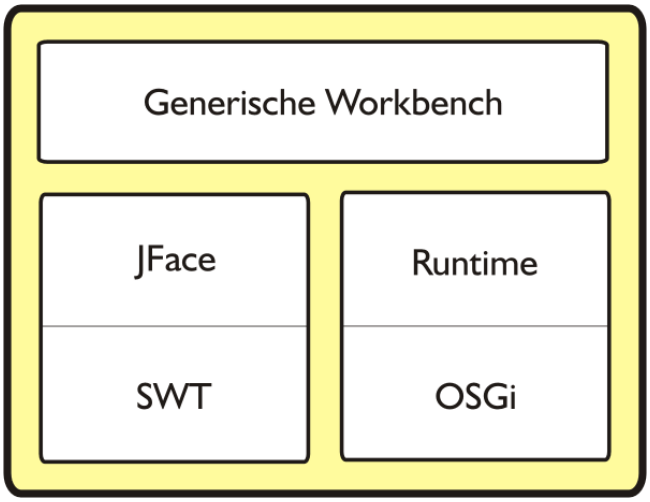

Abb. 5: Die Rich Client Platform

Mit der neuen Rich Client Platform verfügt Eclipse über eine generische Workbench, die klar von der Funktionalität der IDE getrennt wurde und als Basis für eigene Entwicklungen genutzt werden kann. Rich-Client-Anwendungen werden im Prinzip genauso erstellt wie das in 4.3 aufgeführte Plugin-Beispiel: Die Anwendung erweitert den Extension Point *org.eclipse.core.runtime.applications*, wodurch sie zum Plugin der RCP wird. Um die Workbench zu starten, wird die statische Methode `createAndRunWorkbench` des *PlatformUI*-Objekts aufgerufen. Dabei werden ein SWT-Display-Objekt und ein *WorkbenchAdvisor*-Objekt übergeben, über die die Benutzeroberfläche initialisiert und konfiguriert werden kann. Der *WorkbenchAdvisor* übernimmt das Lifecycle Management der Workbench.

Auch hierzu ein kurzes Beispiel:

```
1  import org.eclipse.core.runtime.IPlatformRunnable;
2  import org.eclipse.swt.widgets.Display;
3  import org.eclipse.ui.PlatformUI;
4  import org.eclipse.ui.application.WorkbenchAdvisor;
5
6  public class RichClientAnwendung implements IPlatformRunnable {
7    public Object run(Object args) {
8      WorkbenchAdvisor workbenchAdvisor = new RcpWorkbenchAdvisor();
9      try {
10       int returnCode = PlatformUI.createAndRunWorkbench(
           createDisplay(), new WorkbenchAdvisor() {
11         public String getInitialWindowPerspectiveId() {
12           return "perspective1";
13         }
14       });
15       if (returnCode == PlatformUI.RETURN_RESTART) {
16         return EXIT_RESTART;
17       } else {
18         return EXIT_OK;
19       }
20     } finally {
21       display.dispose();
22     }
23   }
24 }
```

Die Klasse *WorkbenchAdvisor* ist als `abstract` deklariert, es muss ein Exemplar einer eigenen Implementierung an `createAndRunWorkbench` übergeben werden. (Hier mittels einer anonymen Klasse realisiert.) Deren Methode `getInitialWindowPerspectiveId` liefert die zu verwendende Perspektive an die Workbench zurück. Der Benutzeroberfläche können nun GUI-Elemente hinzugefügt werden, indem man Initialisierungs-Methoden der *WorkbenchAdvisor*-Klasse überschreibt.

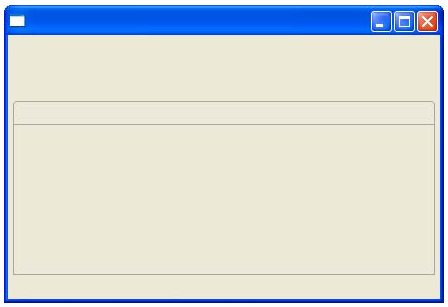

Abb. 6: Eine Basis-Workbench

5 Ausblick

Auch wenn die Eclipse Platform in der aktuellen Version schon eine Vielzahl an Möglichkeiten bietet, ist das Ende der Entwicklung noch lange nicht erreicht. Zur Zeit wird verstärkt an der Integration von Werkzeugen zur Entwicklung von Webanwendungen gearbeitet, insbesondere an der Unterstützung von J2EE-Anwendungen (9). Ebenfalls bereits auf der Zielgeraden ist die Entwicklung einer Test- und Performance-Umgebung unter Leitung von Intel, die beispielsweise Profiler zu Eclipse beisteuern soll. Hier zeigt sich, dass sich Eclipse immer mehr vom "großen Bruder", dem parallel von IBM weiterentwickelten Websphere, emanzipiert und die kommerzielle Entwicklungsumgebung in immer mehr Punkten überflügelt, insbesondere was die Menge der verfügbaren Erweiterungen angeht.

Ein anderes Hauptprojekt ist die Umsetzung der Rich Client Platform für mobile Endgeräte. Die Mitarbeit von Firmen wie Nokia und Motorola an der Portierung zeigt, wie ernst Eclipse inzwischen auch fernab vom Desktop-PC genommen wird. Das "kommerzielle Ökosystem", dessen Aufbau von Anfang an im Blickpunkt der Eclipse Foundation stand, scheint langsam aber sicher zum Selbstläufer zu werden.

Neben diesen großen gibt es eine Vielzahl von kleineren kommerziellen und freien Projekten, die die Erweiterung der Plattform zum Ziel haben. Der komponentenbasierte Ansatz einer offenen Plugin-Architektur, der Eclipse zugrunde liegt, kommt durch die Verschaltung von mehr und mehr Plugins immer stärker zum Tragen. Das *"extensible IDE for everything and yet nothing in particular"* ist wohl noch lange nicht am Ende seiner Möglichkeiten angelangt.

Literatur

[1] Erickson, Mark: *The Eclipse code donation (Interview).* http://www-106.ibm.com/developerworks/linux/library/l-erick.html. – Online–Ressource, Abruf: 18.2.2005

[2] Milinkovic, Mike: *Eclipse now and in the future (Interview).* http://www.itwriting.com/eclipse1.php. – Online–Ressource, Abruf: 18.2.2005

[3] Gallardo, David ; Brunette, Ed ; McGovern, Robert: *Eclipse in Action.* Manning Publications, 2003. – ISBN 1–93011–096–0

[4] Daum, Berthold: *Java-Entwicklung mit Eclipse 3.* dpunkt.verlag, 2004. – ISBN 3–89864–281–X

[5] Aust, Stefan M.: Natives Look and Feel - Eine Einführung in das Standard Widget Toolkit. In: *Eclipse special* (2004)

[6] Weinand, André ; Bäumer, Dirk: Mehr als eine Plattform - Eclipse-Technologie als Basis für die Anwendungsentwicklung. In: *Eclipse special* (2004)

[7] Daum, Berthold: *Java-Entwicklung mit Eclipse 2.* dpunkt.verlag, 2003. – ISBN 3–89864–227–5

[8] Lippert, Martin ; Völter, Markus: Versprechen eingehalten - Eclipse als Plattform für die Entwicklung von flexiblen Rich-Client-Anwendungen. In: *Eclipse special* (2004)

[9] Milinkovic, Mike: Inside Eclipse (Interview). In: *Java Spektrum - Systems 2004* (2004)