8 Erste Schritte: Plug-In-Entwicklung

Das vorherige Kapitel hat das Konzept der Plug-Ins, Erweiterungspunkte und Erweiterungen eingeführt. Diese drei Elemente bilden zusammen die Bausteine, aus denen Eclipse besteht. Was bedeutet das praktisch für Sie als Programmierer? Oder anders ausgedrückt: Welche Dateien müssen Sie erzeugen und welchen Code müssen Sie schreiben, damit Eclipse Ihr Wunderwerk erkennt und integriert?

Alles beginnt mit einer *Plug-In-Manifest-Datei*. In dieser Datei deklarieren Sie, wo Sie Erweiterungen von anderen akzeptieren und wo Sie andere erweitern, ob es sich dabei um die Eclipse-Plattform selbst oder andere Eclipse-basierte Tools handelt, mit denen Sie eine Verknüpfung herstellen möchten. Bevor wir genau zeigen, wie Sie eine Manifest-Datei anlegen, sehen wir uns an, was die Plattform damit anfängt, um Eclipse zum Laufen zu bringen.

Beim Starten von Eclipse stellt die Laufzeitumgebung der Eclipse-Plattform als Erstes fest, welche Plug-Ins verfügbar sind. Dazu sucht sie in den Unterverzeichnissen des Verzeichnisses `<Installationsverzeichnis>\eclipse\plugins` nach Dateien mit dem Namen `plugin.xml` – das Plug-In-Manifest, um das es hier geht. Es parst jede Datei und sucht dabei nach Abhängigkeiten, aus welchem Code das Plug-In besteht, und natürlich nach den Erweiterungen, die das Plug-In vornimmt, und den Erweiterungspunkten, die es definiert. Alle diese Informationen speichert Eclipse in einer als *Plug-In-Registrierung* bezeichneten Baumstruktur, wobei als Schlüssel der Plug-In-Kennzeichner dient. Nachdem die Plug-In-Registrierung aufgebaut ist, überträgt das Laufzeitmodul der Plattform die Verantwortlichkeit für das Hochfahren der Umgebung an ein »Anwendungs-Plug-In«. Standardmäßig ist das das Workbench-Plug-In (`org.eclipse.ui`), das seinerseits das Hauptfenster öffnet, die am Anfang vorhandenen Symbolleisten und Menüs erstellt und die Plattform insgesamt zum Einsatz vorbereitet.

Normalerweise definieren Sie einfach Ihr Plug-In-Manifest, schreiben etwas Code und hoffen, dass Sie rechtzeitig zum Mittagessen zu Hause sind. In diesem Kapitel sollen Sie mehr über Plug-Ins im Allgemeinen erfahren und die grundlegenden Schritte ihrer Entwicklung verstehen, um sich dann in den folgenden Kapiteln mit spezielleren Themen zu beschäftigen. Da die Entwicklung von Eclipse-Plug-Ins mit Design-Überlegungen zur Integration beginnt, setzen wir an diesem Punkt ein und arbeiten uns dann bis zu den Implementierungsdetails durch. Abschließend unternehmen wir einen kurzen Trip durch die Plug-In-Entwicklungsumgebung (PDE[1]).

8.1 Erste Schritte mit Plug-Ins

Ein Tool mit Eclipse zu erstellen unterscheidet sich von anderen Entwicklungsumgebungen. Man beginnt nicht mit einem leeren Bildschirm und erstellt ein Tool. Stattdessen startet man mit einer stabilen Plattform und sucht, an welcher Stelle man sie erweitern und Elemente integrieren möchte. Angesichts dieser Unterscheidung sollten Sie die folgenden Punkte in Ihrem Entwicklungsprozess beachten:

1. PDE – Plug-in Development Environment (Plug-In-Entwicklungsumgebung)

1. *Beschreiben Sie Ihre Benutzerszenarios.*
   Im ersten Schritt gehen Sie Benutzerszenarios Ihrer integrierten Funktionalität durch, um den besten Ablauf aus Sicht des Benutzers zu bestimmen. Berücksichtigen Sie das volle Spektrum der grundlegenden Möglichkeiten von Eclipse, bei dem zu erwarten ist, dass die Benutzer mit Ihrem Tool interagieren, beispielsweise Ansichten, Editoren und Perspektiven. Schließen Sie Szenarios ein, die die Team-Programmierumgebung von Eclipse nutzen.
2. *Suchen Sie nach Integrationspunkten in Eclipse.*
   Alle Erweiterungen werden in Eclipse über Plug-Ins eingebunden. In den meisten Fällen verwenden Sie einen Erweiterungspunkt, den die Eclipse-Plattform bereitstellt. Nachdem Sie das Szenario kennen, das Sie realisieren, können Sie die Menge der zutreffenden Erweiterungspunkte ableiten. Dieses Kapitel dient als Richtlinie, wie Sie die Erweiterungspunkte finden.
3. *Trennen Sie Ihre Benutzeroberfläche vom Modell.*
   In den meisten Fällen besitzt Ihr Plug-In visuelle und nicht visuelle Aspekte, die Sie gegebenenfalls in unterschiedlichen Plug-Ins unterbringen können. Eclipse selbst folgt diesem Entwurfsmuster. Es fällt auf, dass die Plug-Ins der Benutzeroberfläche die Buchstabenfolge `ui` (für User Interface – Benutzeroberfläche) in der Plug-In-Kennung enthalten, beispielsweise `org.eclipse.ui` für das Workbench-Plug-In oder `org.eclipse.jdt.ui` für die Benutzeroberfläche des JDT-Plug-Ins.

   Es ist nicht unbedingt erforderlich, getrennte Plug-Ins für die Modell- und Benutzeroberflächenkomponenten vorzusehen, doch führt das zu einer sauberen Implementierung und verringert oftmals die Zeit für den Startvorgang der Plattform. Außerdem halten Sie sich dadurch die Möglichkeit offen, dass der nicht auf die Benutzeroberfläche bezogene Teil Ihres Tools auch außerhalb der Workbench eingesetzt werden kann (z.B. als Teil eines Apache Ant-Erstellungsskripts). Überlegen Sie also zuerst, wie Ihr Plug-In aussehen könnte und wie es sich in die vorhandene Benutzeroberfläche von Eclipse integrieren lässt. Das beinhaltet vor allem die Frage, ob Sie lediglich vorhandene Elemente der Benutzeroberfläche aufwerten oder eigene Elemente definieren möchten.

   Außerdem sollten Sie berücksichtigen, ob Sie ein von Eclipse bereitgestelltes Modell – beispielsweise das Modell, das dem JDT zugrunde liegt – verwenden oder ein eigenes Modell erzeugen wollen. Wenn Sie zusätzliche Funktionalitäten für die Java-Entwicklung bereitstellen, bietet es sich an, auf das leistungsfähige Java-Modell des JDT zurückzugreifen. In jedem Fall empfiehlt es sich, die Benutzeroberfläche vom Modell zu trennen.
4. *Spalten Sie klar abgegrenzte Funktionsgruppen Ihres Tools in separate Plug-Ins ab.*
   Da Eclipse eine Integrationsplattform ist, die viele unterschiedliche Tools in einer einzigen Umgebung vereint, ist es besonders wichtig, dass funktionell abgegrenzte Teile Ihrer Tools separat geladen (bzw. nicht geladen) werden können, um deren Anteil an der Leistung und Speicherbelastung von Eclipse zu reduzieren.

8.1.1 Integrationsszenarios

Die folgenden Szenarios sind nur einige Beispiele, wie man Eclipse erweitern kann. Es soll beileibe keine erschöpfende Auflistung sein.

Dokumentation

Ein einfaches Integrationsszenario integriert die eigene Dokumentation in die Eclipse-Dokumentation. Dieses Szenario ist nützlich, um eine privat produzierte Dokumentation, beispielsweise ein Dokument mit häufig gestellten Fragen (FAQs), in die eigene Tool-Dokumentation einzubinden. Der Dokumentationsinhalt kann als HTML- oder PDF-Datei vorliegen; Sie müssen lediglich das notwendige XML schreiben, um die Dokumentation zu integrieren. Es ist kein Java-Code erforderlich. Kapitel 23 geht näher auf das Einbinden von Hilfedokumentationen ein.

Kleine funktionelle Erweiterungen einbinden

Ein weiteres einfaches Integrationsszenario bezieht sich auf eine eigenständige inkrementelle Verbesserung. Beispielsweise können Sie eine Aktion zur vorhandenen Menüleiste oder Symbolleiste in der Workbench, in Ansichten oder Editoren hinzufügen, die Ihre Funktion aufruft, wenn sie der Benutzer wählt. Kapitel 9 und Kapitel 18 behandeln Aktionserweiterungen verschiedener Arten und Gültigkeitsbereiche zur Workbench.

Neue Ressourcentypen unterstützen

In einem komplexeren Szenario wollen Sie einen neuen Ressourcentyp unterstützen. Nehmen wir beispielsweise an, dass Sie diesen für Ihre eigene proprietäre Programmiersprache oder für ein Tool für die Anwendungsentwicklung, das einen speziellen Dateityp erzeugt, benötigen.

Als ersten Schritt identifizieren Sie Ihre Ressourcen. Dabei erstellen Sie eine Liste der Dateien, die Sie unterstützen möchten. Für jede Ressource untersuchen Sie den Lebenszyklus: die Ressource wird erzeugt, modifiziert und auf Festplatte oder in einem Repository gespeichert. Zum Beispiel sollte Ihr Entwurf einen einfachen Prozess unterstützen, um die Ressource zu erzeugen. Möglicherweise soll auch der Inhalt importiert und exportiert werden. Mit dem Assistenten-Framework ließe sich ein Erstellungsassistent, der über den Menübefehl FILE | NEW zugänglich ist, für die Ressource erzeugen. Kapitel 11 behandelt das Assistenten-Framework.

Im nächsten Schritt untersuchen Sie, was bei Wahl der Ressource geschehen soll. Welche Aktionen sind für diesen Typ von Ressource zutreffend? Diese Aktionen erscheinen im Kontextmenü bei der Auswahl der Ressource in der Navigator-Ansicht. Für die Ressource kann ein spezieller Editor erforderlich sein. Falls die Ressource textorientiert ist und sich für eine assistierte Texteingabe anbietet, stellt Eclipse einen konfigurierbaren Texteditor mit Funktionen wie zum Beispiel Inhaltsassistenz und farbliche Hervorhebungen bereit. In diesem Fall müssen Sie funktionelle Ergänzungen erstellen, um den Editor zu konfigurieren. In Kapitel 27 erhalten Sie Einblicke in das Erstellen von benutzerdefinierten JFace-Texteditoren. Andernfalls kann ein simpler

Texteditor genügen. Wenn sich die Ressource bevorzugt mit einem spezialisierten Nichttext-Editor (z.B. einem Grafikeditor) bearbeiten lässt, sollten Sie sich mit den Grundlagen von Editoren in Kapitel 13 beschäftigen.

Welche ergänzenden Ansichten sind erforderlich, um die Ressource adäquat zu präsentieren und entsprechende Navigationsmöglichkeiten bereitzustellen? Wenn die Ressource strukturierte Daten enthält, bietet sich die Outline-Ansicht an. Um die Eigenschaften der Ressource anzuzeigen, verwenden Sie die Properties-Ansicht. Entscheiden Sie, ob Ihr Editor Lesezeichen oder andere spezielle Markierungen akzeptieren soll (siehe Kapitel 17). Genügen die von Eclipse bereitgestellten Ansichten? Wenn nicht, erzeugen Sie gegebenenfalls benutzerdefinierte Ansichten. Wenn Sie die verwendeten Ansichten entwerfen, sollten Sie die erforderlichen Kommunikations- und Benachrichtigungsanforderungen spezifizieren. Ändert beispielsweise der Benutzer die Ressource mit einem Editor, der die in den Ansichten dargestellten Daten beeinflusst, müssen Sie den Code schreiben, um die Ansichten über die Änderungen zu informieren. Für jede benutzerdefinierte Ansicht sollten Sie alle ansichtsspezifischen Aktionen erstellen, einschließlich Sortier- und Filterverhalten, falls große Datenmengen in der Ansicht anzuzeigen sind. Kapitel 12 demonstriert, wie Sie eine Ansicht erstellen. Unbedingt sollten Sie einige Benutzerszenarios durchspielen, um sicherzustellen, dass Ihr Editor und die Ansichten harmonisch zusammenarbeiten. Indem Sie die Szenarios durchgehen, testen Sie die Benachrichtigungsanforderungen, die Ereignisbehandlung und den passenden Satz von Editor- und Ansichtsaktionen.

Während die Ressourcen modifiziert werden, möchten Sie vielleicht dem Benutzer ermöglichen, das Verhalten Ihres Editors und der Ansichten an deren Grundeinstellungen anzupassen. Kapitel 11 beschreibt, wie Sie eine eigene Grundeinstellungsseite in Eclipse integrieren.

In manchen Fällen werden Dateien nicht angelegt, indem man sie manuell bearbeitet, sondern durch laufende Programme, die die Datei erstellen, wenn sie eine abhängige Datei speichern oder wenn der Erstellungsvorgang einer abhängigen Datei explizit angefordert wird. Die Aufgabe eines Builders in Eclipse ist es, eine Ressource des einen Typs zu übernehmen und eine neue Ressource zu erstellen. Beispielsweise erstellt der Java-Builder Klassendateien aus Java-Quellcodedateien. Für Ihre Ressourcen kann ein spezieller Builder erforderlich sein. Berücksichtigen Sie in Ihrem Entwurf die Ressourcenbehandlung, wie zum Beispiel Benachrichtigungen bei Ressourcenänderungen, wodurch sich mehrere Ansichten einer Ressource synchronisieren lassen. Kapitel 15 behandelt Ressourcenbenachrichtigungen und Kapitel 16 beschäftigt sich mit dem Erstellen eines Builders.

Der letzte Aspekt im Lebenszyklus einer Ressource ist die Persistenz und Verwaltung der Ressource. Beachten Sie in Ihrem Entwurf, wie die Ressource gespeichert werden soll. Wenn die Ressource eine Änderungsverwaltung erfordert, können Ihre Benutzer von der integrierten Repository-Unterstützung in Eclipse profitieren.

Sie sollten sich nicht nur ansehen, welche verfügbaren Erweiterungspunkte infrage kommen, sondern auch betrachten, wie Sie Ihren Code für andere erweiterbar machen können. Kapitel 20 beschreibt, wie Sie neue Erweiterungspunkte erzeugen. Außerdem bedeutet das, dass Ihre Ansichten, Editoren und das Datenmodell offen und erweiterbar sein müssen.

8.1.2 Ihr Eclipse-basiertes Tool ausliefern

Nachdem Sie Ihr Tool in Eclipse integriert haben, müssen Sie es an Ihre Benutzer ausliefern. Zur Entwurfszeit entscheiden Sie, ob Eclipse bei Ihren Benutzern bereits installiert sein muss, bevor sie Ihr Tool installieren, oder ob Sie Eclipse als Paket zusammen mit Ihrem Installationsmedium ausliefern. Weiterhin sollten Sie sich über das Erscheinungsbild Gedanken machen, das Sie Ihren Benutzern in Bezug auf die Produktmarke präsentieren möchten. Das heißt, welche Art von Startbildschirm, Lizenz und Anfangsperspektive soll beim Starten Ihres Tools erscheinen? Das Verpacken von Plug-Ins zur Auslieferung beschreibt Kapitel 22.

8.2 Erste Schritte mit Erweiterungen und Erweiterungspunkten

Erweiterungspunkte (Extension Points) definieren, wo andere Plug-Ins Funktionalität in ein Plug-In einbinden können. Wie eben erwähnt, besteht eine wichtige Aufgabe bei der Erweiterung von Eclipse darin, die geeigneten Erweiterungspunkte zu finden. In gewissem Sinne geben die Erweiterungen und Erweiterungspunkte einer Plug-In-Manifest-Datei über die Verbindungen zwischen Plug-Ins Auskunft.

Die Ähnlichkeit der Begriffe Erweiterungen und Erweiterungspunkte verdient eine nähere Erläuterung. Sie entsprechen zwei Plug-In-Manifest-Tags mit ähnlichen Namen, `<extension-point>` und `<extension>`. Das erste definiert einen neuen Erweiterungspunkt; das zweite *liefert einen Beitrag* zu einem bereits definierten Erweiterungspunkt. Im Sprachgebrauch des Plug-In-Entwicklers »erweitert« man einen Erweiterungspunkt, wenn man ihn verwendet. Dieses und die folgenden Kapitel konzentrieren sich auf Erweiterungen zu bereits definierten Erweiterungspunkten, d.h. auf das `<extension>`-Tag. Kapitel 20 beschäftigt sich dann damit, wie man das `<extension-point>`-Tag verwendet.

Jeder Erweiterungspunkt hat einen Kennzeichner, der entsprechend der Namenskonvention für Java-Pakete angegeben wird. Im Verlauf dieses Buches lernen Sie die Mehrheit der von Eclipse definierten Erweiterungspunkte kennen. Tabelle 8.1 fasst zusammen, welche Erweiterungen Sie vornehmen und wo in der Eclipse-Benutzeroberfläche wiederfinden können, gibt den dazu gehörenden Erweiterungspunkt an und nennt das Kapitel, das sich näher damit befasst.

Zweck	Erweiterungspunkt	Behandelt in Kapitel
Aktionen dem Fenstermenü oder der Symbolleiste der Workbench hinzufügen	`org.eclipse.ui.actionSets`	9 und 18
Aktionen der Fenstermenüleiste oder der Symbolleiste der Workbench hinzufügen, wenn die angegebene Ansicht/der Editor in der Perspektive geöffnet ist	`org.eclipse.ui.actionSetPartAssociations`	
Aktionen der Symbolleiste oder als Menübefehle eines Editors hinzufügen	`org.eclipse.ui.editorActions`	
Aktionen einem Kontextmenü eines Editors, einer Ansicht oder eines Objekts hinzufügen	`org.eclipse.ui.popupMenus`	
Die Symbolleiste oder das Pulldown-Menü einer Ansicht erweitern	`org.eclipse.ui.viewActions`	
Einen neuen Assistenten für die Standardmenübefehle EXPORT und IMPORT bereitstellen	`org.eclipse.ui.exportWizards` `org.eclipse.ui.importWizards`	11
Einen neuen Assistenten für den Standardmenübefehl FILE	NEW bereitstellen	`org.eclipse.ui.newWizards`	11
Eine Grundeinstellungsseite für das Dialogfeld WINDOW	PREFERENCES bereitstellen	`org.eclipse.ui.preferencePages`	
Seiten für das Eigenschaftsdialogfeld eines Objekts bereitstellen	`org.eclipse.ui.propertyPages`	
Neue Ansichten für den Menübefehl WINDOW	SHOW VIEW definieren	`org.eclipse.ui.views`	12
Zusätzliche Filter für den Menübefehl FILTERS der Navigator-Ansicht definieren	`org.eclipse.ui.resourceFilters`	
Zusätzliche Text- oder Symboldekorationen für Objektbeschriftungen bereitstellen	`org.eclipse.ui.decorators`	
Neue Editoren für Ressourcen definieren; diese erscheinen als Menübefehle im Untermenü von OPEN WITH.	`org.eclipse.ui.editors`	13 und 27

Tabelle 8.1 Beiträge zu den Erweiterungspunkten der Eclipse-Plattform

Zweck	Erweiterungspunkt	Behandelt in Kapitel
Neue Perspektiven für den Menübefehl WINDOW	OPEN PERSPECTIVE definieren. Neue Tastenkürzel für Perspektiven und Ansichten sowie Action Sets für eine vorhandene Perspektive hinzufügen.	`org.eclipse.ui.perspectives` `org.eclipse.ui.perspectiveExtensions`	14
Eigene inkrementelle Erstellungsvorgänge (PROJECT | REBUILD PROJECT) für vorhandene Ressourcentypen oder für eigene Ressourcen definieren. Projektfähigkeiten erweitern.	`org.eclipse.core.resources.builders` `org.eclipse.core.resources.natures`	16
Eine Ressource mit bestimmten Benutzerinformationen markieren. In Abhängigkeit von dem Markierungstyp können die Benutzerinformationen in Ansichten oder Editoren angezeigt werden, wie zum Beispiel in der Tasks-Ansicht, auf dem vertikalen Lineal von Texteditoren und als Bezeichnungsdekorationen in der Outline-Ansicht.	`org.eclipse.core.resources.markers`	17
Über den Menübefehl HELP | HELP CONTENTS verfügbare Onlinehilfe definieren	`org.eclipse.ui.help`	23

Tabelle 8.1 Beiträge zu den Erweiterungspunkten der Eclipse-Plattform (Forts.)

Die letzte Spalte nennt das Kapitel, das den in der mittleren Spalte angegebenen Erweiterungspunkt zum ersten Mal behandelt. In der Tabelle taucht nicht jedes Kapitel auf, weil sich Eclipse nicht ausschließlich um Erweiterungspunkte dreht. Es gibt andere nützliche Frameworks, die Sie ebenfalls kennen sollten. Beispielsweise sind das Standard Widget Toolkit (SWT) und JFace keine Plug-Ins, sondern Frameworks der Benutzeroberfläche, die Sie aber in Ihrer Plug-In-Implementierung einsetzen. Die fortgeschritteneren Themen in späteren Kapiteln gehen über die allgemeine Verwendung von Erweiterungen hinaus und geben unter anderem eine Einführung, wie sich andere Eclipse-Plug-Ins, zum Beispiel das JDT, erweitern lassen.

Wenn Sie zu einer gegebenen Erweiterung beitragen möchten, müssen Sie auch Kenntnisse über ihr unterstützendes Framework besitzen. Um beispielsweise einen Assistenten für den Menübefehl FILE | NEW hinzuzufügen, müssen Sie mehr wissen, als die Parameter dieses Erweiterungspunktes. Dazu benötigen Sie Kenntnisse zum Framework, das sich dahinter verbirgt und das Assistentendialogfeld mit dessen Seiten erzeugt, wie es die Navigation von Seite zu Seite behandelt usw. Auf dieses Thema geht Kapitel 11 ein.

Analog behandeln andere Kapitel die Erweiterungspunkte, die sich auf den Bereich von Eclipse beziehen, den Sie erweitern möchten. Bei vielen Erweiterungen stellt das

Kapitel allgemeine Schritt-für-Schritt-Anweisungen zur Implementierung vor, die Sie an Ihren speziellen Fall anpassen können.

Nachdem Sie nun in groben Zügen wissen, was Sie in einem Plug-In-Manifest finden, kommen wir zu den Grundlagen zurück, wie man eine Manifest-Datei erzeugt. Alle Erweiterungen und Erweiterungspunkte werden in der Manifest-Datei eines Plug-Ins als XML spezifiziert. Der Inhalt des Elements `<extension>` wird mit der `ANY`-Regel deklariert. Das heißt, dass sich beliebiges formatgerechtes XML innerhalb des Konfigurationsabschnittes einer Erweiterung angeben lässt (zwischen den Tags `<extension>` und `</extension>`). Anschließend müssen Sie diese Tags und Attribute sorgfältig eingeben, da zur Entwicklungszeit kaum Prüfungen für die manuell eingegebenen Elemente stattfinden. Dennoch brauchen Sie nicht ängstlich zu sein; Eclipse definiert eine Perspektive, in der Ihnen spezialisierte Editoren und Assistenten helfen – die so genannte *Plug-in Development Environment* (PDE). Zum Beispiel kann Sie der NEW EXTENSION-Assistent der PDE durch den Erweiterungsprozess auf mehreren Wegen führen. Erstens kann der NEW EXTENSION-Assistent eine Vorlage verwenden, die eine oder mehrere Erweiterungs-spezifische Assistentenseiten präsentiert, um die erforderlichen Parameter abzufragen (siehe Abbildung 8.1).

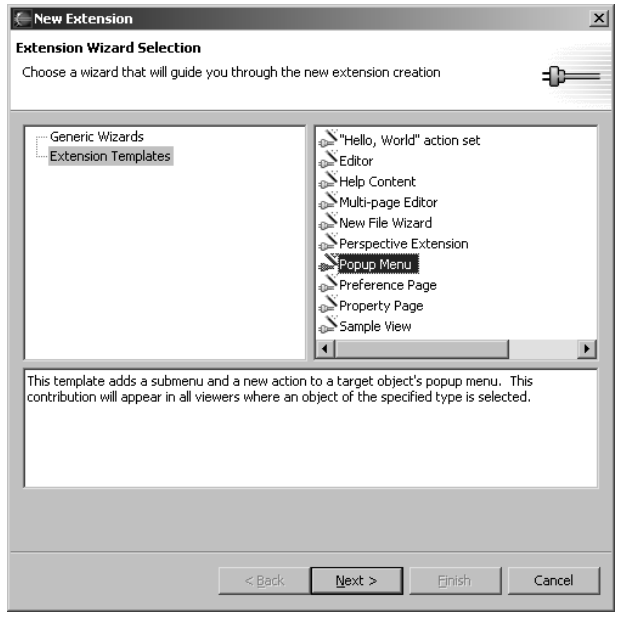

Abbildung 8.1 Der NEW EXTENSION-Assistent und die Erweiterungsvorlagen

Alternativ können Sie eine Erweiterung mit einem generischen NEW EXTENSION-Assistenten erzeugen. Er führt Sie durch das Erstellen einer Erweiterung basierend auf einer Schemadefinition der erwarteten untergeordneten Tags und Attribute.

Mehr zur PDE erfahren Sie gegen Ende dieses Kapitels. Es zeigt sich, dass Eclipse umfangreiche Hilfe für diejenigen bereitstellt, die die Eclipse-Umgebung erweitern möchten. Dazu gehören wie bereits erwähnt spezialisierte Editoren, Ansichten und Assistenten sowie Unterstützung für das Testen unter Laufzeitbedingungen.

8.3 Grundlegende Schritte beim Implementieren eines Plug-Ins

Wenn Sie ein Plug-In entwickeln, durchlaufen Sie im Allgemeinen folgende Schritte:

1. *Entscheiden Sie, wie Ihr Plug-In in die Plattform integriert wird, und suchen Sie die betreffenden Erweiterungspunkte;* das sind die Erweiterungspunkte, die Sie erweitern. Die Eclipse-API lässt sich grob in Benutzeroberfläche (z. B. Workbench) und Nicht-Benutzeroberfläche (z. B. Arbeitsbereich) gliedern.
2. *Bestimmen Sie die Anforderungen der Erweiterungspunkte.*
   Jeder Erweiterungspunkt verlangt, dass Sie ihm spezifische Informationen bereitstellen. Die Onlinedokumentation und die generierte Javadoc (selbstverständlich auch die Kapitel dieses Buches und die Beispiele auf der Begleit-CD) sind ausgezeichnete Quellen, die die erforderlichen Informationen erläutern.
3. *Deklarieren Sie Ihr Plug-In-Manifest.*
   Der Plug-in-Manifest-Editor der PDE hilft Ihnen, die Manifest-Datei des Plug-Ins korrekt zu verfassen.
4. *Implementieren Sie die Funktionalität für Ihre Erweiterung.*
   Viele Erweiterungspunkte verlangen, dass Sie die Funktionalität mit entsprechendem Code vervollständigen. Der Autor des Erweiterungspunktes stellt normalerweise entweder eine Schnittstellenklasse bereit, die die erwarteten Methoden definiert, die Ihre Klasse implementieren muss, oder eine abstrakte Superklasse, die Sie erweitern können.
5. *Definieren Sie die Plug-In-Klasse (optional).*
   Ein Plug-In repräsentiert eine Zusammenstellung von genau definierter Funktionalität, die Eclipse erweitert. Eine *Plug-In-Klasse* ist jedoch die Singleton-Klasse, die mit jedem Plug-In verbunden ist. Sie können Ihre eigene Plug-In-Klasse definieren, um von wesentlichen Ereignissen des Lebenszyklus benachrichtigt zu werden, oder als Referenz auf Ressourcen, die von anderen Erweiterungen innerhalb des Plug-Ins benötigt werden.
6. *Installieren Sie das Plug-In.*
   Der Abschnitt »Das Plug-In installieren« später in diesem Kapitel gibt einen kurzen Überblick über die Installationsoptionen und Kapitel 22 behandelt sie im Detail.

Nach all diesen Ausführungen zu Plug-Ins, Erweiterungen und Erweiterungspunkten sind Sie bestimmt schon darauf gespannt, wie eine Plug-In-Manifest-Datei aussieht. Das nachstehende Beispiel ist ein Plug-In-Manifest für einen fiktiven »Getting Started Editor« und bringt viele der hier behandelten Tags zusammen:

```
<?xml version="1.0"?>
<plugin
  name="Getting Started Editor"
  id="com.ibm.lab.examples.gseditor"
  version="1.0.1"
  vendor-name="IBM"
  class="com.ibm.examples.gseditor.EditorPlugin">
```

```
  <requires>
    <import plugin="org.eclipse.ui"/>
    <import plugin="org.eclipse.core.resources"/>
  </requires>

  <runtime>
    <library name="gsedit.jar"/>
  </runtime>

  <extension point="org.eclipse.ui.editors">
    <editor
      id="com.ibm.lab.examples.myreadmetexteditor"
      name="Getting Started Editor"
      icon="icons/text_edit.gif"
      extensions="readme"
      class="com.ibm.lab.examples.MyReadmeTextEditor">
    </editor>
  </extension>
</plugin>
```

Überrascht? Mehr ist es nicht. Dieses Plug-In definiert den neuen Editor »Getting Started Editor«, der Dateien mit der Dateierweiterung `readme` bearbeitet. Der Editor selbst ist in der Klasse `com.ibm.lab.examples.MyReadmeTextEditor` definiert und in der JAR-Datei `gsedit.jar` verpackt.

Die konkreten Parameter dieses Beispiels sind momentan nicht wichtig, vielmehr interessiert die allgemeine Struktur. Die Beispiele in späteren Kapiteln behandeln die verschiedenen Erweiterungspunkte, ihre untergeordneten Tags und die Attribute im Detail. Sehen wir uns nun die Tags an, die allen Plug-Ins gemeinsam sind.

8.3.1 Das Plug-In-Manifest deklarieren

Ein Plug-In-Manifest wird in XML spezifiziert und muss den Namen `plugin.xml` tragen. Normalerweise besteht ein Plug-In aus mehreren Dateien, wobei mindestens die Manifest-Datei `plugin.xml` erforderlich ist. Zusätzlich binden die meisten Plug-Ins ausführbaren Java-Code und andere Ressourcen ein, die das Plug-In für die Realisierung seiner Funktionen benötigt, beispielsweise Bild-, Eigenschafts- und HTML-Dateien.

Eine Plug-In-Manifest-Datei gliedert sich in mehrere Abschnitte, beginnend mit dem Element `<plugin>`:

```
<plugin
    id='com.ibm.lab.helloworld'
    name='Hello, World'
    version='1.0.0'
    vendor='IBM'
    class='com.ibm.lab.helloworld.HelloWorldPlugin'>
```

Die **fett** gedruckten Attribute sind erforderlich. Das Attribut `id` wird wie die meisten `id`-Parameter in der Manifest-Datei gemäß der Konvention zur Benennung von Java-Paketen angegeben, um Namensbereichskonflikte zu vermeiden. Es definiert eine Programmreferenz auf das Plug-In selbst. Das Attribut `name` gibt die Bezeichnung an, die in der Plug-in Registry-Ansicht und ähnlichen Dialogfeldern erscheint. Das Attribut `class` bestimmt die Klasse, die von den Ereignissen im Lebenszyklus des Plug-Ins benachrichtigt wird, wie beispielsweise Startvorgang und Herunterfahren, nachdem das Plug-In aktiviert ist.

Der Abschnitt `<requires>` definiert die abhängigen Plug-Ins:

```
<requires>
  <import
    plugin='org.eclipse.runtime.ui'
    version='2.0.1'
    match='compatible'
    optional='false'
    export='false'/>
</requires>
```

Die Attribute `version` und `match` spezifizieren die gewünschte Plug-In-Version und den dafür anzuwendenden Übereinstimmungsalgorithmus. Obwohl diese Attribute nicht obligatorisch sind, verdienen sie dennoch besondere Aufmerksamkeit. Das Attribut `version` wird allgemein mit drei Zahlen (»`2.0.1`« im obigen Beispiel) im Format `major.minor.service` angegeben, wobei die Bestandteile folgende Bedeutung haben:

- `major` definiert die Kompatibilitätsebene zwischen den Freigaben. Ein größerer Wert von `major` deutet an, dass Elemente dieser Freigabeversion mit der vorherigen Freigabe inkompatibel sind.
- `minor` kennzeichnet die Variante der Hauptversion, die akzeptabel ist, wenn `match='compatible'` gesetzt wird.
- `service` gibt die Aktualisierung einer bestimmten `major.minor`-Freigabe an, die bei `match='equivalent'` akzeptiert wird.

Außerdem lässt sich eine vierte Zahl im Format `major.minor.service.qualifier` mit folgender Bedeutung angeben:

- `qualifier` zeigt die Quellcodekontrollversion derselben Komponente an. In Bezug auf die Versionsübereinstimmung wird der Qualifizierer in allen Fällen mit Ausnahme von `match='perfect'` ignoriert. Dieser mit der Version 2.0 von Eclipse eingeführte Bestandteil des Attributes wird allerdings selten verwendet.

Das Attribut `match='greaterOrEqual'` wird angegeben, wenn eine neuere Plug-In-Version akzeptabel ist. In manchen Fällen lässt man das `version`-Attribut weg, da Plug-In-Versionen im Allgemeinen aufwärtskompatibel sind. Mit den Attributen `version` und `match` lassen sich Abhängigkeiten zwischen Plug-Ins festlegen und erzwingen.

Um diesen Punkt zu verdeutlichen, betrachten wir den Fall, bei dem `version='1.0.0'` spezifiziert ist und das Attribut `match` des Tags `<import>` die in Tabelle 8.2 angegebenen Werte hat.

Installierte Versionen	'greaterOrEqual'	'compatible'	'equivalent'	'perfect'
1.0.0	akzeptiert	akzeptiert	akzeptiert	akzeptiert
1.0.1	akzeptiert	akzeptiert	akzeptiert	abgelehnt
1.1.0	akzeptiert	akzeptiert	abgelehnt	abgelehnt
1.2.2	akzeptiert	akzeptiert	abgelehnt	abgelehnt
2.0.0	akzeptiert	abgelehnt	abgelehnt	abgelehnt

Tabelle 8.2 Ergebnisse bei unterschiedlichen Werten für das Attribut `match` *des Tags* `<import>`

Je weiter Sie in der Tabelle nach rechts kommen, desto restriktiver sind die Regeln und das Risiko wächst, dass Ihr Plug-In nicht geladen wird. Indem Sie weiterhin mit den APIs arbeiten, die in früheren Versionen des abhängigen Plug-Ins definiert sind, erhöhen Sie die Wahrscheinlichkeit, dass die Version (oder eine kompatible Version) verfügbar ist.

Es lassen sich mehrere Versionen eines Plug-Ins gleichzeitig installieren. Das wird unterstützt und erwartet. Allerdings wirft das folgende Frage auf: Was passiert, wenn mehrere Plug-Ins von unterschiedlichen Versionen desselben Plug-Ins abhängig sind? Nur eine Version eines Plug-Ins kann geladen werden, sodass das Laufzeitmodul von Eclipse die neueste verfügbare Version lädt und den Versionsabgleich mit dieser durchführt.

Hinweis

Die obigen Regeln zum Versionsabgleich beziehen sich auf die Kompatibilität und nicht die erweiterte Funktionalität. Auch wenn die Marketing-Abteilung darauf besteht, die nächste Freigabe der Version 1.0 als 2.0 auszuliefern, könnte es aus programmtechnischer Sicht die Version 1.1.0 sein, wenn sie 100%ig API-kompatibel mit Version 1.0.x ist. Versionsnummern in der Markenbezeichnung für »neue und verbesserte« Funktionalität werden in Features definiert, auf die wir in Kapitel 22 zurückkommen.

Das Attribut `optional` gibt an, ob die Importabhängigkeit streng durchzusetzen ist. Der Standardwert `false` bedeutet, dass die in den `<import>`-Klauseln aufgeführten Plug-Ins verfügbar sein müssen, damit Ihr Plug-In geladen werden kann. Setzt man dieses Attribut auf `true`, darf ein erforderliches Plug fehlen und Ihr Plug-In lässt sich trotzdem laden. Dieses Attribut können Sie mit Ihren Menüaktionserweiterungen kombinieren, um die Auswahl nur zu erlauben, wenn ein Plug-In verfügbar ist. Beispielsweise verwendet das JDT diesen Ansatz, um nur dann Debugger-bezogene Menübefehle in die Benutzeroberfläche einzubinden, wenn die Debug-Umgebung aktiv ist. Kapitel 9 erläutert diese Beispiele ausführlicher.

Die Klausel `<runtime>` spezifiziert eine oder mehrere Bibliotheken (JAR-Dateien), die den Laufzeitcode des Plug-Ins definieren:

```
<runtime>
  <library name="runtime.jar">
```

```
    <export name="*"/>
  </library>
</runtime>
```

Ihnen ist bekannt, wie die Java-Sichtbarkeitsmodifizierer (`public`, `private`, `protected` usw.) den Zugriff auf Ihre Klassen beeinflussen. Die `export`-Klausel legt auf einer feineren Stufe fest, welche Pakete oder Klassen für andere außerhalb des Plug-Ins selbst sichtbar sind. Damit können Sie Klassen definieren, die für Ihr Plug-In öffentlich, für andere Plug-Ins jedoch nicht sichtbar sind. Das Attribut `name="*"` kennzeichnet, dass alle in der Klassenbibliothek des Plug-Ins definierten öffentlichen Klassen sichtbar sind. Um dies einzuschränken, spezifizieren Sie Paketnamen im gleichen Format, das die Java-Anweisung `import` akzeptiert, oder mit vollständig qualifizierten Klassennamen. Es empfiehlt sich, die Exporte zu minimieren. Andernfalls riskieren Sie eine engere Kopplung zwischen Plug-Ins und geringere Flexibilität, insbesondere von erforderlichen Plug-Ins. Im Allgemeinen sollte es genügen, nur die erforderlichen Plug-Ins oder Bibliotheken, die zur öffentlichen Schnittstelle eines Plug-Ins beitragen, zu exportieren.

Das Tag `<library>` akzeptiert auch Unterverzeichnispfade relativ zum Installationsverzeichnis des Plug-Ins. Diese Möglichkeit bietet sich an, wenn Sie die Standorte der Ressourcen spezifizieren möchten, nach denen der Klassenlader sucht (z. B. `*.properties`-Dateien).

```
<runtime>
  <library name="resources/" />
</runtime>
```

Dieses Plug-In weist den Klassenlader an, im Installationsverzeichnis des Plug-Ins und dessen Unterverzeichnis `resources` zu suchen.

8.3.2 Die Plug-In-Klasse definieren

Optional kann ein Plug-In eine so genannte *Plug-In-Klasse* definieren, indem das `class`-Attribut des Tags `<plugin>` angegeben wird. Ihre Plug-In-Klasse erweitert entweder `Plugin` oder `AbstractUIPlugin` und wird von den Ereignissen im Lebenszyklus der Workbench über die Methoden `startup` und `shutdown` benachrichtigt. Die Klasse `Plugin` definiert unter anderem folgende Methoden:

- `getStateLocation` gibt ein Dateiverzeichnis zurück, in das ein Plug-In persistente Daten schreiben kann (das `plugins`-Verzeichnis und dessen Unterverzeichnis gelten als schreibgeschützt).
- `openStream` gib einen Eingabestrom für eine Datei zurück, die relativ zum Installationsverzeichnis des Plug-Ins ist.
- `getPluginPreferences` gibt einen Plug-In-spezifischen Grundeinstellungsspeicher zurück (eine durch Schlüssel indizierte Persistenztabelle grundlegender Werte wie `string`, `integer`, `boolean` und `float`). Kapitel 11 gibt weitere Hinweise zur Unterstützung von Grundeinstellungen.

Die Klasse `AbstractUIPlugin` enthält weitere Methoden, die für Plug-Ins der Benutzeroberfläche hilfreich sind:

- `getImageRegistry` gibt eine Registrierung von gemeinsam genutzten Bildern zurück.
- `getDialogSettings` liefert eine Tabelle von persistenten Dialogeinstellungen für die verschiedenen Assistenten und Dialogfelder. Dialogfeldeinstellungen verwendet man im Allgemeinen, um dem Benutzer das erneute Eingeben von Informationen zu ersparen, indem man beispielsweise die letzten zehn Sucheinträge in einem Kombinationsfeld anzeigt.
- `getWorkbench` ist eine Komfortmethode, die die aktuelle `IWorkbench`-Instanz zurückgibt. Die Workbench ist das Wurzelobjekt der Eclipse-Benutzeroberfläche. Von hier aus können Sie alle aktiven Fenster abfragen, die den Workbench-Inhalt anzeigen, neue Fensterereignisse registrieren usw.

In Ihrem Code sollten Sie das Unterverzeichnis des Plug-Ins zur Laufzeit als schreibgeschützt behandeln, um sicherzustellen, dass ein Plug-In in einer gegebenen Plattformkonfiguration arbeitet (z.B. auf der Linux-Plattform, wo Eclipse und seine Plug-Ins auf einem gemeinsamen schreibgeschützten Serververzeichnis installiert werden können und der Arbeitsbereich mit Lese-Schreib-Zugriff im Home-Verzeichnis des Benutzers angelegt wird).

Wenn ein Plug-In Zustandsdaten schreiben muss, sollte es seinen Arbeitspfad mithilfe der Plattform-API abrufen. Plug-Ins verwenden ein Arbeitsverzeichnis unter dem Verzeichnis `eclipse\.metadata\.plugins`, um ihre Plug-In-spezifischen Dateien zu lesen und zu schreiben. Die Methode `Plugin.getStateLocation()` gibt das Unterverzeichnis `.metadata` des Plug-Ins zurück. Beispielsweise kann ein Plug-In mit der Kennung `com.ibm.lab.example` und der Plug-In-Klasse `ExamplePlugin`, die die Singleton-Methode `getDefault()` implementiert, eine Methode kodieren, um eine schreibgeschützte Instanz von `java.io.File` zurückzugeben:

```
private File getStateFile() {
  IPath path = ExamplePlugin.getDefault().getStateLocation();
  path = path.append("state.dat");
  return path.toFile();
}
```

Im Beispiel liefert der Aufruf dieser Methode eine `File`-Instanz, die sich in <*Installationsverzeichnis*>\eclipse\workspace\.metadata\.plugins\com.ibm.example\state.dat befindet.

Analog gibt es Methoden in der Klasse `Plugin`, um den Installationsort des Plug-Ins zurückzugeben, beispielsweise `ExamplePlugin.getDefault().getDescriptor().getInstallURL()`. Ein Eingabestrom für eine Datei, die sich außerhalb des Installationsverzeichnisses des Plug-Ins befindet, lässt sich einfach durch Aufruf von `ExamplePlugin.getDefault().openStream(new Path("config.ini"))` öffnen. Mit dieser Methode können Sie auch Bild- oder Konfigurationsdateien abrufen, die im Verzeichnis Ihres Plug-Ins installiert wurden.

Alle Klassen in den JAR-Dateien, die das Tag <`library`> der Plug-In-Manifest-Datei spezifiziert, sind für die Plug-In-Klasse sichtbar. Folglich stellt die Plug-In-Klasse

einen idealen Standort für Methoden dar, die im Allgemeinen für solche Klassen nützlich sind, aus denen sich das Plug-In als Ganzes zusammensetzt. Andere Plug-Ins, die Ihr Plug-In importieren, können ebenfalls auf derartige Methoden zugreifen. Die Komfortmethode `ResourcesPlugin.getWorkspace()` ist ein gutes Beispiel einer nützlichen Methode, die für jedes Plug-In verfügbar ist, das das Ressourcen-Plug-In importiert. In der Tat ist diese Methode wichtig genug, um in die PDE-Auswahl der optional generierten Komfortmethoden einer Plug-In-Klasse aufgenommen zu werden.

Beim erstmaligen Aktivieren eines Plug-Ins verifiziert der Klassenlader, ob die zugeordnete Plug-In-Klasse (falls sie definiert ist) geladen und initialisiert worden ist. Fehlt das `class`-Attribut im `<plugin>`-Tag, wird eine Instanz von `DefaultPlugin` erzeugt. Somit verfügen alle Plug-Ins über eine Plug-In-Klasse, die über den Aufruf von `Platform.getPlugin(String id)` oder über eine direkte Referenz auf die Plug-In-Klasse (wie zum Beispiel `ResourcesPlugin.getWorkspace()`) zugänglich ist. Als Erleichterung kann der PDE-Plug-In-Codegenerator optional eine statische `getDefault()`-Methode definieren, die die Singleton-Instanz der Plug-In-Klasse zurückgibt.

8.3.3 Das Plug-In installieren

In Kapitel 6 haben Sie eine Einführung in den Update-Manager erhalten. Den Update-Manager können Sie auch die Installation Ihres Plug-Ins abwickeln lassen. Dazu müssen Sie ein Feature definieren, das das Plug-In repräsentiert. Das ist das bevorzugte Verfahren, um das Plug-In letztlich auszuliefern. Für Test- und Entwicklungszwecke verwendet man allerdings im Allgemeinen die in den nächsten beiden Unterabschnitten vorgestellten Alternativen.

Das Plug-In ohne den Update-Manager installieren

Um das Plug-In ohne den Update-Manager zu testen und zu installieren, müssen Sie die Grundlagen der Installationsverzeichnisstruktur von Eclipse kennen.

Eclipse hat ein reserviertes Unterverzeichnis `plugins` außerhalb des Wurzelinstallationsverzeichnisses, das für jedes Plug-In ein Unterverzeichnis enthält. Per Konvention haben die Unterverzeichnisse den gleichen Namen wie die Plug-In-ID. Als zusätzliche Konvention für ausgelieferte Plug-Ins hängt man die Versionsnummer als Suffix an, beispielsweise `org.eclipse.core.runtime_2.1.1`. Jedes Unterverzeichnis enthält eine Plug-In-Manifest-Datei `plugin.xml`. Die Namenskonventionen sind nicht zwingend vorgeschrieben – Sie können Ihr Plug-In-Installationsverzeichnis beliebig benennen, es empfiehlt sich aber, den obigen Konventionen zu folgen, um mögliche Namensbereichskonflikte zu vermeiden. Alle Plug-In-Ressourcen einschließlich JAR-, HTML-, Bild- und Eigenschaftsdateien müssen im Installationsverzeichnis oder in einem Verzeichnis relativ dazu gespeichert werden. Damit stellen Sie sicher, dass jedes Plug-In Standort-unabhängig ist und sich Installation und Updates von Plug-Ins vereinfachen.

Für die ersten Testläufe verwenden Sie die PDE, um Ihre Plug-Ins zu testen und zu verwalten. Auf der Grundeinstellungsseite *Target Platform* der PDE (WINDOW | PREFERENCES | *Plug-in Development* | *Target Platform*) können Sie die Liste der Plug-Ins auswählen, die Sie testen möchten. Wenn Sie bereit zum Testen sind, wählen Sie entweder RUN | RUN AS | RUN-TIME WORKBENCH oder DEBUG | DEBUG AS | RUN-TIME

WORKBENCH, um eine zweite Instanz von Eclipse mit den angegebenen Plug-Ins zu starten. Dieses Verfahren verwendet die Übung in Kapitel 33 und der restliche Teil dieses Buches.

Um das Plug-In außerhalb der PDE zu testen, können Sie eine zweite Kopie von Eclipse getrennt von Ihrer Entwicklungsinstallation installieren. Damit bilden Sie die endgültige Auslieferungskonfiguration besser nach als beim Testen unter PDE-Regie. Sobald Sie mit dem Test beginnen können, kopieren Sie die Manifest-Datei des Plug-Ins, dessen gepackte JAR-Datei und seine erforderlichen Ressourcen in ein neues Unterverzeichnis außerhalb des Verzeichnisses `plugins` und starten dann Ihre Testinstallation von Eclipse.

Das Plug-In als Feature des Update-Managers ausliefern

Features erlauben die Organisation der Plug-Ins, sodass sie sich durch Eclipse verwalten lassen. In Teil I dieses Buches haben Sie bereits den Update-Manager kennen gelernt. Wie der vorherige Abschnitt erläutert hat, werden die direkt in das `plugins`-Verzeichnis hinzugefügten Plug-Ins vom Laufzeitmodul der Plattform erkannt, während sie der Update-Manager als »nicht verwaltete Plug-Ins« behandelt und ignoriert.

Ein Feature definieren Sie nach dem gleichen Verfahren wie ein Plug-In, d.h. indem Sie eine Feature-Manifest-Datei (`feature.xml`) definieren und in einem Verzeichnis desselben Namens wie die Feature-Kennung speichern. Feature-Verzeichnisse liegen in der Eclipse-Plattform in einem Unterverzeichnis namens `features`.

Kapitel 22 zeigt, wie Sie Features erzeugen, und beschreibt die erforderlichen Schritte, damit sie der Update-Manager finden, installieren und Service-Updates auf das Feature anwenden kann.

8.4 Die Plug-In-Entwicklungsumgebung

Eclipse basiert auf einer Plug-In-Architektur und wurde mit der Erwartung konzipiert, dass Entwickler mithilfe von Eclipse selbst Erweiterungen zu Eclipse erstellen. Um den Plug-In-Entwicklern zu helfen, die notwendigen Programmkomponenten zu erzeugen und zu verwalten, und das gewissermaßen rekursive Vorgehen zu unterstützen, mit einem Entwicklungstool das Tool selbst zu entwickeln, bindet Eclipse eine Umgebung ein, die speziell für diesen Zweck geschaffen wurde: die PDE (Plug-in Development Environment).

Dieser Abschnitt gibt einen kurzen Überblick über die PDE. Die Übung in Kapitel 33 führt Sie durch die PDE und demonstriert, wie Sie damit Plug-Ins erzeugen, testen und debuggen.

8.4.1 PDE-Ansichten und -Editoren

In Teil I haben Sie bereits das Java Development Toolkit (JDT) von Eclipse kennen gelernt. Die PDE bindet alle Funktionen des JDT ein und bringt zudem neue PDE-spezifische Ansichten und Editoren mit. Abbildung 8.2 zeigt die PDE nach dem Start eines Plug-In-Projekts.

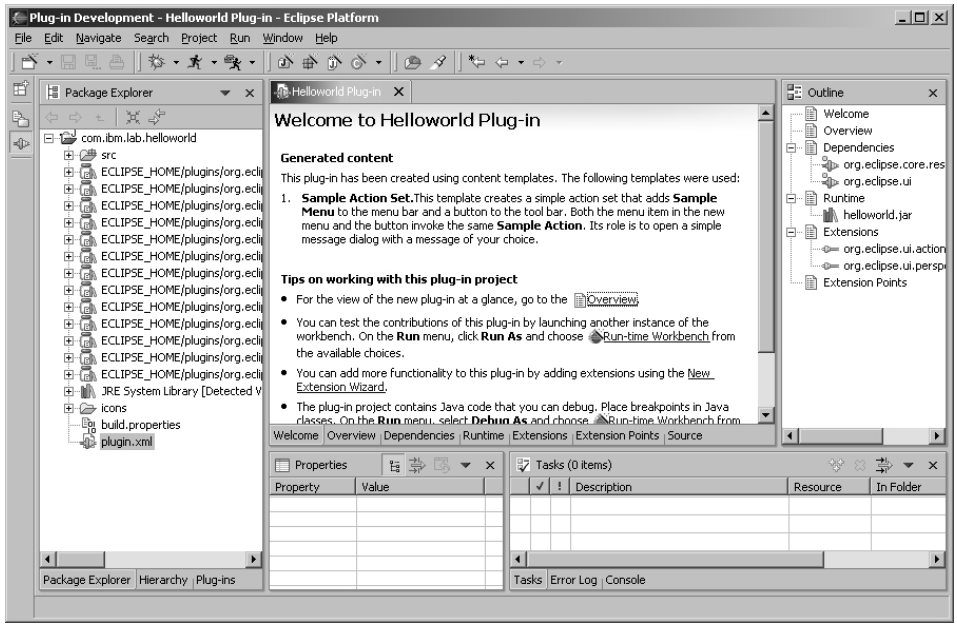

Abbildung 8.2 Die Plug-In-Entwicklungsumgebung

Sicherlich haben Sie bemerkt, dass Sie den Projekttyp wählen müssen, wenn Sie ein Projekt anlegen: *Simple*, *Java* oder *Plug-in Development*. Der NEW PROJECT-Assistent erzeugt das Projektverzeichnis und die erforderlichen Ordner für den jeweiligen Projekttyp (z.B. `bin` und `src` im Fall eines Plug-In-Projekts). Ein Projekt kann spezielle Verhaltensweisen definieren, die für die enthaltenen Ressourcentypen geeignet sind, wie zum Beispiel das inkrementelle Kompilieren von Java-Quelldateien. Wenn Sie ein Plug-In-Projekt erzeugen, erscheint die PDE-Perspektive automatisch und öffnet die Manifest-Datei `plugin.xml` des Plug-Ins im dazu gehörenden Editor, dem Plug-In-Manifest-Editor. Abbildung 8.3 zeigt dessen OVERVIEW-Seite.

Jede Seite im Plug-In-Manifest-Editor zeigt den Inhalt eines Abschnittes der Plug-In-Manifest-Datei in einem übersichtlichen und leicht verständlichen Format an, wie es Tabelle 8.3 beschreibt.

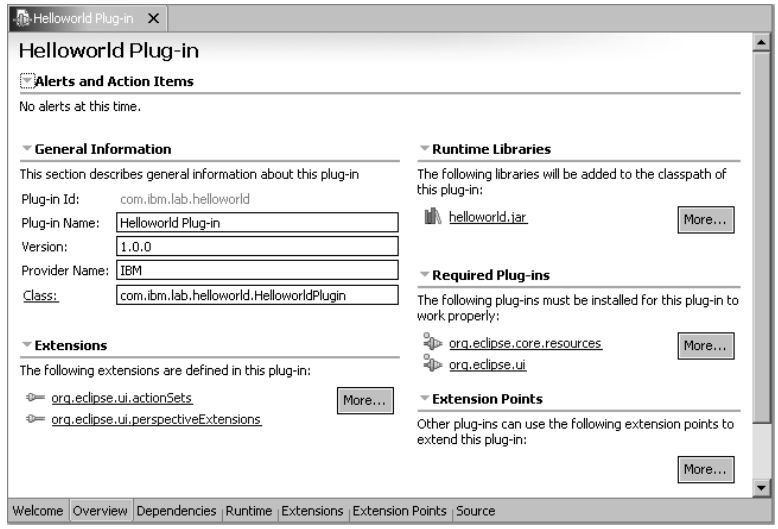

Abbildung 8.3 Überblicksseite des Plug-in-Manifest-Editors

Seite	Inhalt
OVERVIEW	Eine Zusammenfassung des Plug-In-Manifests und Erinnerungen an relevante Fehler- oder Informationsmeldungen unter »Alerts and Action Items«
DEPENDENCIES	Die <`requires`>-Klausel
RUNTIME	Die <`runtime`>-Klausel
EXTENSIONS	Die Liste der <`extension`>-Klauseln
EXTENSIONS POINTS	Die Liste der <`extension-points`>-Klauseln
SOURCE	Formatierter `plugin.xml`-Inhalt, der sich direkt bearbeiten lässt.

Tabelle 8.3 Seiteninhalte im Plug-In-Manifest-Editor

Über diese Seiten können Sie Abschnitte der dazu zugehörenden `plugin.xml`-Datei bearbeiten oder auf die Seite SOURCE gehen und das XML direkt modifizieren. Der Quellinhalt Ihrer XML-Datei wird automatisch aktualisiert, sodass immer der aktuelle Inhalt zu sehen ist, wenn Sie zwischen den Seiten der PDE wechseln.

Die PDE stellt die beiden Laufzeitansichten Plug-in Registry (Plug-In-Registrierung) und Error Log (Fehlerprotokoll) bereit, die Ihnen beim Debuggen Ihres Plug-Ins helfen (siehe Abbildung 8.4).

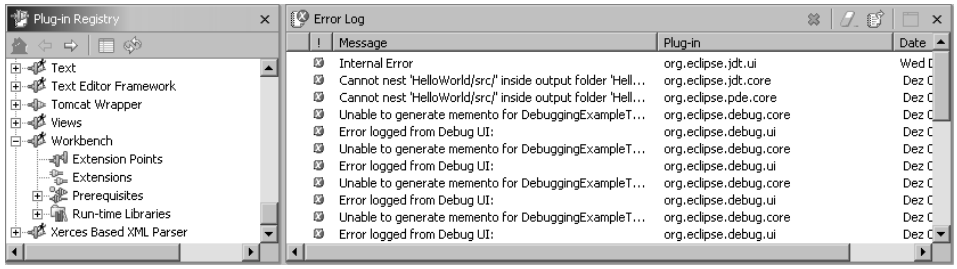

Abbildung 8.4 Die PDE-Ansichten Plug-in Registry und Error Log

Die Plug-in Registry-Ansicht zeigt die Inhalte der beim Startvorgang gelesenen Plug-In-Manifest-Dateien sortiert nach Plug-In-ID. Beim Starten eines Plug-Ins versieht die Plug-in Registry-Ansicht den Plug-In-Eintrag mit einem Dekorationssymbol (einer rennenden Person). Die Error Log-Ansicht ist ausschließlich den Fehlern vorbehalten, die von geladenen Plug-Ins oder vom Plattformkern, der die Plug-Ins verarbeitet, stammen. Kapitel 21 zeigt, wie Ihr Plug-In in dieses Protokoll schreiben kann, und erläutert andere Verfahren wie zum Beispiel das Tracing, um die Möglichkeiten für die Fehlersuche und -beseitigung Ihres Plug-Ins zu erweitern.

Zur Entwicklungsumgebung gehört als dritte PDE-Ansicht die Plug-ins-Ansicht (siehe Abbildung 8.5) die Ressourcen anzeigt, die mit externen Plug-Ins verbunden sind (diejenigen Plug-Ins außerhalb Ihres Arbeitsbereichs), wie zum Beispiel die Manifest-Datei und die JAR-Datei des Plug-Ins.

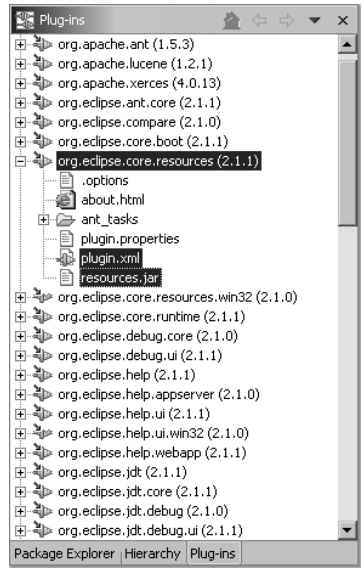

Abbildung 8.5 Die PDE-Ansicht Plug-ins

Die Plug-ins-Ansicht bietet Menüoptionen, über die Sie den Code von externen Plug-Ins in Ihren aktuellen Arbeitsbereich importieren und ihn hier modifizieren können. In Kapitel 33 lernen Sie, wie man diese Menüoptionen einsetzt.

8.4.2 Die Laufzeit- und Entwicklungsumgebung

Da Eclipse dazu verwendet wird, Eclipse-Plug-Ins zu erstellen, ist es einfacher, diese beiden Umgebungen separat zu verwalten. Die erste bezeichnen wir als *Entwicklungsumgebung*, d.h. die Instanz, die Sie anfangs gestartet haben. Die zweite firmiert unter der Bezeichnung *Laufzeitumgebung*; sie ist die Instanz, die die Entwicklungsumgebung startet, wenn Sie Ihre Plug-Ins testen.

Eclipse muss wissen, welche Plug-Ins zur Entwicklungsumgebung und welche zur Laufzeitumgebung gehören. Die zur Entwicklungsumgebung gehörenden Plug-Ins lassen sich leicht definieren: es handelt sich um alle die Plug-Ins, die sich unterhalb des `plugins`-Verzeichnisses befinden. Um die zur Laufzeitumgebung gehörenden Plug-Ins einzuordnen, hat man mehrere Möglichkeiten. Standardmäßig umfasst die Laufzeitumgebung alle Plug-Ins in geöffneten Projekten im Arbeitsbereich. Außerdem können Sie auf der Seite *Target Platform* der PDE-Grundeinstellungen »externe« Plug-Ins (die sich nicht im Arbeitsbereich befinden) in der Laufzeitinstanz sichtbar machen, wie es Abbildung 8.6 zeigt.

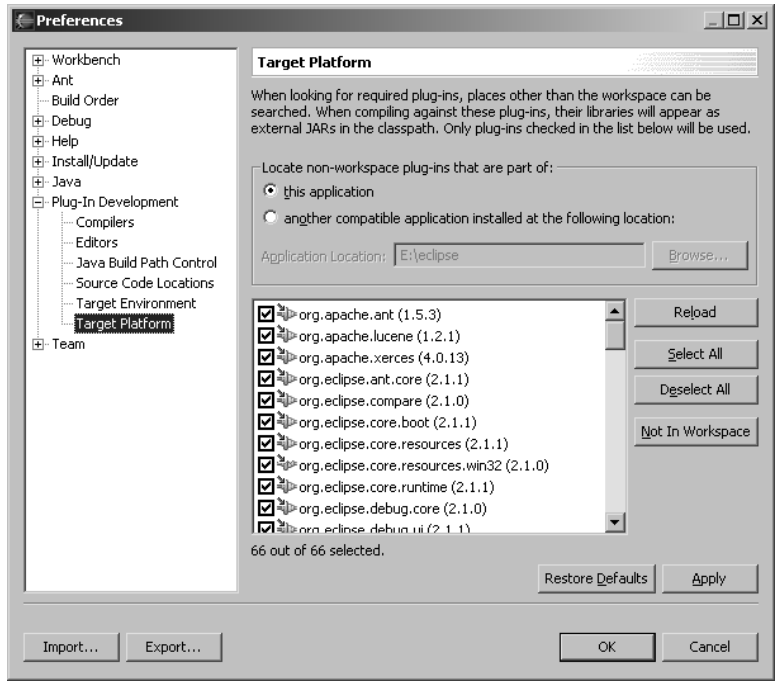

Abbildung 8.6 Die Seite Target Platform der PDE-Grundeinstellungen

Wählen Sie die Plug-Ins aus, die Ihr Plug-In-Manifest in der <requires>-Klausel festlegt. Damit es schnell geht, können Sie NOT IN WORKSPACE wählen, um alle Plug-Ins zu spezifizieren, die sich im plugins-Verzeichnis, aber noch nicht in Ihrem Arbeitsbereich befinden. Diese Auswahl kann mehr Plug-Ins in Ihre Testumgebung einbinden, als Ihr Plug-In tatsächlich benötigt, für Testzwecke hat das aber keine praktischen Konsequenzen und wir konfigurieren die PDE auf diese Weise für die Beispiele im Buch. Die Datei readme.html auf der Begleit-CD zum Buch beschreibt die übrigen Details, wie die PDE-Umgebung zu konfigurieren ist, bevor Sie die Lösungen importieren.

8.4.3 Ein Plug-In erzeugen und ausführen

Ein Plug-In zu erstellen ist relativ leicht, insbesondere da die PDE Assistenten zur Codegenerierung von Plug-Ins als Teil der Erzeugung eines Plug-In-Projekts bereitstellt, wie es Abbildung 8.7 zeigt.

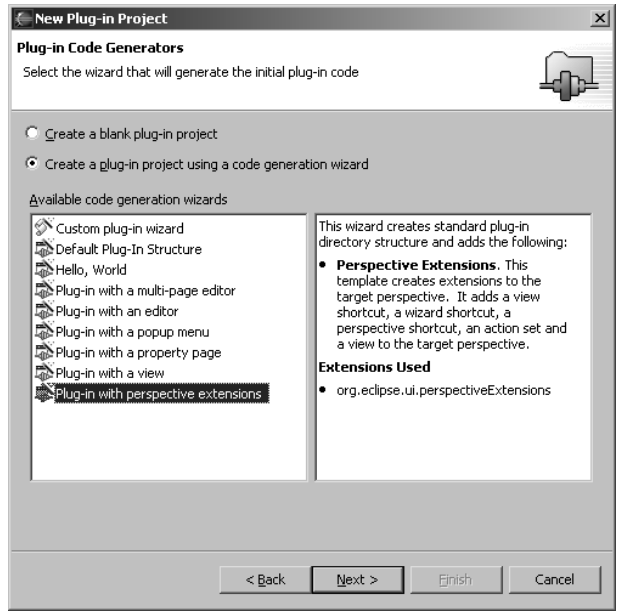

Abbildung 8.7 Assistenten für die Codegenerierung von Plug-Ins

In Kapitel 33 haben Sie die Wahl, das Plug-In »Hello, World« mit dem Assistenten zur Codegenerierung zu erstellen oder es manuell Schritt für Schritt aufzubauen, um die Abläufe von Plug-Ins und der PDE besser zu verstehen.

Nachdem Sie ein Plug-In geschrieben haben, führen Sie es im nächsten Schritt aus und müssen es gegebenenfalls debuggen. Wenn Sie RUN in der PDE wählen, startet eine Laufzeitinstanz von Eclipse. Diese Instanz von Eclipse bindet die Plug-Ins in Ihrem Arbeitsbereich und diejenigen, die Sie auf der Seite *Target Environment* des Dia-

logfelds für die *Plug-in Development*-Grundeinstellungen angegeben haben, ein. Um Ihre Plug-Ins einfach auszuführen, wählen Sie RUN | RUN AS | RUN-TIME WORKBENCH. Den Debugger rufen Sie mit RUN | DEBUG AS | RUN-TIME WORKBENCH auf. Bei komplexeren Startvorgängen können Sie alternativ benutzerdefinierte Startkonfigurationen erzeugen. Wählen Sie dazu RUN | RUN oder RUN | DEBUG, um das Dialogfeld LAUNCH CONFIGURATIONS zu öffnen, das Abbildung 8.8 zeigt.

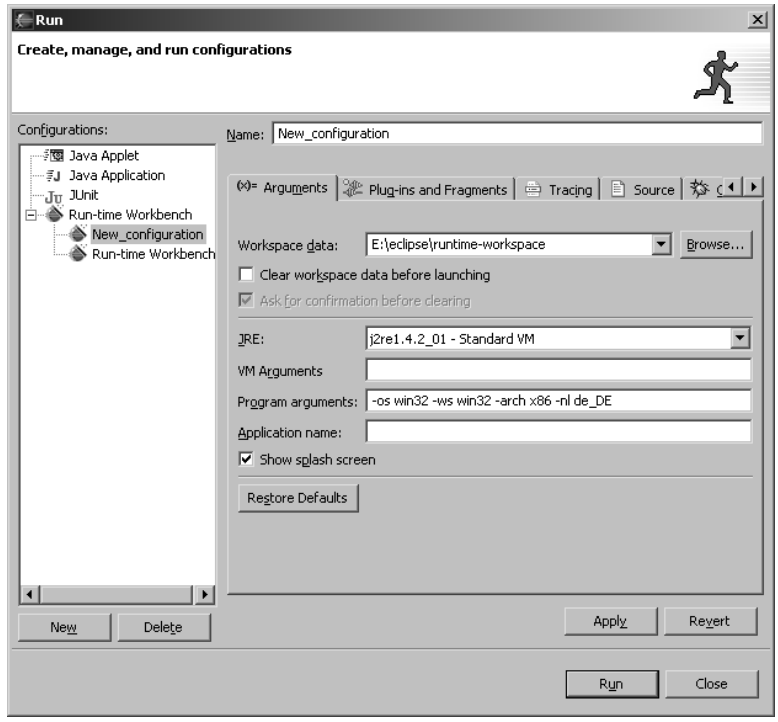

Abbildung 8.8 Startkonfigurationen verwalten

Mit einer Startkonfiguration können Sie die Startumgebung Ihrer Laufzeitinstanz in Abhängigkeit vom konkreten Fall statt plattformweit verwalten.

8.4.4 Eclipse mit Eclipse entwickeln: Eigenständiges und verteiltes Self-Hosting

Die Lösungen in diesem Buch gehen davon aus, dass Sie eigenständiges Self-Hosting verwenden, d.h. die in die Laufzeitumgebung eingebundenen Plug-Ins werden durch die Projekte im Arbeitsbereich und die auf der Seite *Target Platform* der PDE-Grundeinstellungen ausgewählten Plug-Ins bestimmt. Das Setup ist schnell und komfortabel, es gibt aber folglich Verweise in der `.classpath`-Datei Ihres Projekts, die diese Auswahl kennzeichnen. Das hat keine ernsthaften Auswirkungen, wenn Sie allein arbeiten, kann aber etwas unbequem werden, wie Sie Projekte mit anderen Ent-

wicklern gemeinsam nutzen möchten, die ihren Arbeitsbereich nach anderen Prämissen einrichten. Zusätzliche Informationen hierzu finden Sie im Abschnitt »UI Developer Resources« des PDE-Teilprojekts in *eclipse.org*. Insbesondere wenn Sie in einer Team-Umgebung arbeiten, sollten Sie das Dokument mit dem Titel »Self-hosting in Eclipse using PDE« lesen. Wollen Sie dagegen nicht mit einem Team arbeiten, können Sie wie bereits erwähnt die Schaltfläche NOT IN WORKSPACE auf der Seite TARGET PLATFORM wählen. Damit weisen Sie die PDE an, alle in Ihrer Entwicklungsumgebung verfügbaren Plug-Ins und alle Plug-Ins in Ihrem Arbeitsbereich in Ihrer Laufzeittestumgebung zu vereinen.

8.5 Zusammenfassung der Übung

Wenn Sie die Übung in Kapitel 33 noch nicht absolviert haben, sollten Sie das schleunigst nachholen. Die Übung führt Sie durch das Erstellen eines einfachen »Hello, World« Plug-Ins und demonstriert dabei die Merkmale der PDE. Zur Übung gehört keine Codevorlage, sodass Sie sofort beginnen können.

8.6 Kapitelzusammenfassung

Die Eclipse-Workbench definiert eine IDE-Plattform, die neue Funktionalitäten von unterschiedlichen Entwicklern integrieren kann und dabei eine nahtlose Benutzeroberfläche und eine einheitliche Benutzerführung gewährleistet. Diese Integration läuft über Plug-Ins und Plug-In-Erweiterungen. Nachdem Sie den Ausgangspunkt aller Plug-Ins – das Plug-In-Manifest – und seine Hauptelemente – Erweiterungen und Erweiterungspunkte – kennen gelernt haben, können Sie dazu übergehen, diejenigen zu verwenden, aus denen die Eclipse-Plattform besteht. Die restlichen Kapitel dieses Buches konzentrieren sich auf diese Plug-Ins und Erweiterungspunkte sowie deren zugrunde liegenden Frameworks, mit denen Sie Eclipse erweitern können.

8.7 Quellen

Developing and deploying plug-ins for WebSphere Studio. *https://www6.software.ibm.com/reg/devworks/dw-wssplugin-i*. (Eine kurze Registrierung ist erforderlich.)

Self-hosting in Eclipse using PDE. Siehe die »Development Resources« der Benutzeroberflächenkomponente des PDE-Teilprojekts unter *http://www.eclipse.org/pde/index.html*.