von dirk bäumer, daniel megert
und andré weinand

mehr zum thema:
www.eclipse.org/
www.awprofessional.com/titles/0321205758

ECLIPSE PLUG-INS – ENTWICKELN UND PUBLIZIEREN

Liest man zur Zeit über Eclipse etwas in der Presse, so beschäftigen sich die Artikel entweder mit Eclipse als Open-Source-Entwicklungsumgebung für die Programmiersprache Java oder mit Eclipse als erweiterbare Tool-Plattform. Der Fokus dieses Artikels liegt auf dem Aspekt der Erweiterbarkeit.

Wir zeigen im Folgenden anhand eines Beispiels, welche Erweiterungsmöglichkeiten die Eclipse-Plattform bietet. Dazu entwickeln wir ein eigenes *Plug-In*, das verschiedene Metriken für Java-Dateien berechnet und deren Ergebnisse in einer entsprechenden *View* darstellt. Ferner demonstrieren wir, wie das entwickelte *Plug-In* mittels Eclipse auf einer Web-Seite publiziert werden kann, sodass es für andere Entwickler öffentlich zugänglich ist. Als Basis für das in diesem Artikel entwickelte *Plug-In* haben wir das Release 2.1.2 (November 2003) verwendet, das über die Download-Seite der Eclipse-Homepage verfügbar ist (**siehe Kasten 1**).

Eine einfache Metrik

Bevor wir mit der Entwicklung des neuen *Plug-Ins* beginnen können, müssen wir zuerst einen entsprechenden Entwicklungs-Workspace aufsetzen. Wir starten Eclipse dazu mit einem neuen Workspace und importieren alle benötigten *Plug-Ins*, um eine Erweiterung zur Java-Entwicklungsumgebung zu implementieren. Unter Zuhilfenahme des *Plug-In Development Environment (PDE)* sehen die Schritte wie folgt aus:

- Öffnen des Import-Assistenten (*wizard*) über *File > Import*
- Auswählen von *External Plug-Ins and Fragments* und zweimaliges Betätigen des *Next*-Knopfes
- Auswahl von „org.eclipse.jdt.ui" in der Liste der *Plug-Ins* und Betätigen des Knopfes *Add Required Plug-Ins*
- Auslösen des eigentlichen Imports durch die Aktion *Finish*

Weiterhin erzeugen wir ein neues *Plug-In* mit dem Namen „ospektrum.plugin":

- Wir öffnen den Assistenten *New Project* über *File > New > Project*.
- Dann selektieren wir *Plug-In Development* in der linken Spalte und *Plug-In Project* in der rechten; durch die Aktion *Next* wechseln wir zur nächsten Seite.
- Als Projektname geben wir „ospektrum.plugin" ein. Durch zweimaliges Betätigen des *Next*-Knopfes gelangen wir zur Seite *Plug-In Code Generators*, auf der wir den Eintrag *Create a blank plug-in project* auswählen.
- Die Aktion *Finish* erzeugt das neue Projekt.

Die erste Erweiterung, die das neue *Plug-In* bereitstellen soll, ist eine Aktion, welche die Anzahl der Zeilen einer Java-Datei (*compilation unit*) bestimmt. Schauen wir uns dazu erst einmal etwas genauer an, wie Erweiterungen an Eclipse vorgenommen werden. Eclipse besteht aus einer Menge von *Plug-Ins*, die einen kleinen Laufzeitkern (*runtime kernel*) erweitern. So sind die Java-Entwicklungsumgebung, die Anbindung an das CVS-Repository-System und die Workbench selbst bereits Erweiterungen dieses Laufzeitkerns.

Eclipse-Ressourcen im Netz

Erste Anlaufstelle für Informationen rund um Eclipse ist die Web-Seite www.eclipse.org. Hier finden sich zahlreiche Artikel sowie die neusten Eclipse-Versionen zum Herunterladen. Über den Verweis „community" gelangt man zu Informationen über andere Web-Seite, die sich ebenfalls dem Thema Eclipse widmen. Neben der Web-Seite existieren noch diverse News-Gruppen zum Thema. Erwähnt sei hier die Gruppe „eclipse.[platform,swt.tools.jdt]", die sich sowohl an Eclipse-Anwender als auch an *Plug-In*-Entwickler wendet.
Speziell den Entwicklern der Eclipse-Plattform vorbehalten sind die diversen Mailing-Listen. Sie sollten nicht für Anwenderfragen oder für technische Fragen rund um die *Plug-In*-Entwicklung zweckentfremdet werden.
Die Eclipse Web-Seite enthält ein Archiv sowohl für die News-Gruppen als auch für die Mailing-Listen. Es existieren ebenfalls Informationen, wie man in den Gruppen und Listen partizipieren kann.

Kasten 1

die autoren

Dirk Bäumer
(E-Mail: Dirk_Baeumer@ch.ibm.com),

Daniel Megert
(E-Mail: Daniel_Megert@ch.ibm.com)
und

André Weinand
(E-Mail: Andre_Weinand@ch.ibm.com)
sind Mitarbeiter der IBM OTI Labs in Zürich. Alle drei sind Committer für diverse Komponenten des Eclipse Open-Source-Projekts.

Eclipse besteht also im Gegensatz zu anderen Entwicklungsumgebungen nicht aus einem monolithischen Kern, der eine Menge von Erweiterungspunkten (*extension points*) anbietet, sondern aus einem kleinem Kern mit einer unbegrenzten Anzahl von Erweiterungsmöglichkeiten – unbegrenzt deshalb, weil jedes *Plug-In* selbst wieder eine Menge von Erweiterungspunkten definieren kann, die dann wiederum von anderen *Plug-Ins* implementiert werden können.

Wie wird nun sichergestellt, dass eine Architektur, die mit einer Vielzahl von Erweiterungen umgehen können muss, noch skaliert? So besteht Eclipse derzeit aus 66 *Plug-Ins*, und kommerzielle Produkte, wie etwa „WebSphere Studio Application Developer" von IBM, werden mit mehre-

ren hundert *Plug-Ins* geliefert. Die Lösung heißt hier: verzögertes Laden. Dazu unterteilt Eclipse ein *Plug-In* in einen deklarativen und einen implementierenden Teil. Der deklarative Teil beschreibt die Funktionalität eines *Plug-Ins* und erlaubt so zum Beispiel der Workbench, Elemente wie Assistenten oder Aktionen in Menüs darzustellen, ohne den eigentlichen Programmcode bereits laden und ausführen zu müssen. Startet man beispielsweise Eclipse und öffnet den *New Project*-Assistenten (*File > New > Project*), so enthält dieser den Eintrag *Java Project*. Dieser Eintrag wird erzeugt, indem der deklarative Teil eines *Plug-Ins* beim Starten von Eclipse gelesen wird. Der deklarative Teil eines *Plug-Ins* wird in einer XML-basierten Manifest-Datei niedergelegt („plugin.xml"). Zusammenfassend ergibt sich somit folgendes Bild:

- Eclipse wird um neue Funktionalität erweitert, indem Implementierungen für einen Erweiterungspunkt bereitgestellt werden.
- Eine Menge solcher Implementierungen wird in einem *Plug-In* zusammengefasst. Ein *Plug-In* ist die kleinste installierbare Einheit in Eclipse.
- Ein *Plug-In* besteht aus einer Manifest-Datei, die beschreibt, welche Erweiterungspunkte das *Plug-In* implementiert, welche eigenen Erweiterungspunkte es zur Verfügung stellt, welches Java-Archiv die Implementierung enthält und welche anderen *Plug-Ins* es zur Ausführung benötigt. Die Implementierung erfolgt in der Programmiersprache Java. Daneben enthält ein *Plug-In* noch Ressourcen, Ikonen oder internationalisierte Zeichenketten.

Wenden wir uns nun wieder unserem Beispiel zu. Ziel ist es, die neue Funktionalität, die die Anzahl der Zeilen einer Java-Datei berechnet, dem Kontextmenü einer solchen hinzuzufügen (**siehe Kasten 2**). Um Kontextmenüs zu erweitern, definiert die Workbench den Erweiterungspunkt org.eclipse.ui.popupMenus. Wir müssen somit in unserer Manifest-Datei deklarieren, dass unser *Plug-In* diesen Erweiterungspunkt implementiert, und den entsprechenden Java-Programmcode bereitstellen. Die Manifest-Datei des Metrik-*Plug-Ins*, das den Eintrag für die Erweiterung des Kontextmenüs für Java-Dateien enthält, ist in **Listing 1** dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="ospektrum.plugin"
   name="Objekt Spektrum Plug-In"
   version="1.0.0">
   <runtime>
      <library name="plugin.jar"/>
   </runtime>
   <extension point="org.eclipse.ui.popupMenus">
      <objectContribution
         id="ospektrum.plugin.contribution"
         objectClass=
                  "org.eclipse.jdt.core.ICompilationUnit">
         <action
            id="ospektrum.plugin.actions.NumberOfLines"
            label="Number of Lines"
            class="ospektrum.plugin.NumberOfLinesAction"/>
      </objectContribution>
   </extension>
</plugin>
```

Listing 1: Die Manifest-Datei des Metrik-Plug-Ins

Für eine detaillierte Beschreibung des Erweiterungspunktes org.eclipse.ui.popupMenus verweisen wir auf das Online-Hilfesystem. Wir wollen lediglich auf das id-Attribut der Elemente objectContribution und action sowie auf das Attribut objectClass eingehen. Fast alle XML-Elemente einer Manifest-Datei haben einen systemweit eindeutigen Kennzeichner. Um die Eindeutigkeit sicherzustellen, verwendet Eclipse die Konvention, den Kennzeichnern den Wert des *Plug-In* id-Attributes voranzustellen. Das Attribut objectClass gibt an, für welche selektierten Objekte die Aktionen, die anschließend in der Manifest-Datei aufgeführt sind, erscheinen sollen. In unserem Beispiel erscheint die Aktion im Kontextmenü von Objekten, die typkonform zu org.eclipse.jdt.core.ICompilationUnit sind. Objekte dieser Klasse repräsentieren in Eclipse Dateien mit der Endung .java.

Laut unseren Aussagen müsste der deklarative Teil bereits ausreichen, um die Aktion in der Benutzungsschnittstelle sichtbar zu machen. Um dies zu testen, starten wir eine zweite Eclipse-Instanz, indem wir die Aktion *Run > Run As > Run-time Workbench* auswählen. Um die beiden Eclipse-Instanzen im Weiteren voneinander unterscheiden zu können, bezeichnen wir die Instanz, mit der wir unser *Plug-In* entwickeln, als Entwicklungs-Workspace und die andere als Test-Workspace.

Im Test-Workspace erzeugen wir nun ein Java-Projekt und fügen eine neue Java-Datei hinzu. Deren Kontextmenü sollte nun den Eintrag *Number Of Lines* enthalten. Wählen wir diese Aktion aus, präsentiert Eclipse einen Informationsdialog, der anzeigt, dass die ausgewählte Aktion momentan nicht verfügbar ist. Zudem erscheint in der Konsole unseres Entwicklungs-Workspace eine Fehlermeldung, dass die Klasse NumberOfLinesAction nicht geladen werden konnte. Dies ist zu erwarten, da ja noch keine Implementierung bereitgestellt wurde.

Wie finde ich mögliche Erweiterungspunkte?

Das integrierte Hilfesystem enthält eine Dokumentation aller in Eclipse existierenden Erweiterungspunkte. Man gelangt ins Hilfesystem über *Help > Help Contents*. Folgende Online-Bücher sind dabei für den *Plug-In* Entwickler interessant: „Platform Plug-In Developer Guide" und „JDT Plug-In Developer Guide".
Wird der Workspace mittels der Funktion *File > Import > External Plug-Ins and Fragments* erzeugt, so steht der Quellcode von Eclipse zum Browsen zur Verfügung. Es lohnt sich auf jeden Fall, sich die eine oder andere Manifest-Datei eines anderen *Plug-Ins* anzuschauen.

Kasten 2

Jede Implementierung eines Erweiterungspunktes in Eclipse muss einem wohl definierten Protokoll genügen. Protokolle werden in der Programmiersprache Java durch Interfaces oder Klassen ausgedrückt. Demnach sind in Eclipse Erweiterungspunkte mit einem oder mehreren Interfaces (oder Klassen) assoziiert. Für Aktionen innerhalb des Erweiterungspunktes org.eclipse.ui.popup Menus ist dies das Interface org.eclipse.ui.IObjectActionDelegate. Die Implementierung von NumberOfLinesAction zeigt **Listing 2**[1]).

[1]) *Wir versehen Instanzvariablen mit dem Präfix „f", um sie besser von lokalen Variablen und Parametern unterscheiden zu können.*

```
package objektspektrum.plugin
public class NumberOfLinesAction
                  implements IObjectActionDelegate {
   private IWorkbenchPart fWorkbenchPart;
   private ISelection fSelection;
   public void setActivePart(
         IAction action,
         IWorkbenchPart targetPart) {
      fWorkbenchPart= targetPart;
   }
   public void run(IAction action) {
      ICompilationUnit unit= getCompilationUnit();
      try {
         Document doc= new Document(unit.getSource());
         MessageDialog.openInformation(
            fWorkbenchPart.getSite().getShell(),
            "Number Of Lines",
            "The result is: " + doc.getNumberOfLines());
      } catch (JavaModelException e) {
         // error handling
      }
   }
   public void selectionChanged(
         IAction action,
         ISelection selection) {
      fSelection= selection;
   }
   private ICompilationUnit getCompilationUnit() {
      return (ICompilationUnit)
         ((IStructuredSelection)fSelection).
         getFirstElement();
   }
```

Listing 2: Die Implementierung der NumberOfLinesAction

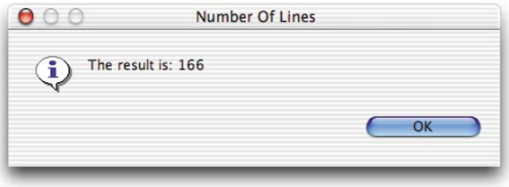

Abb. 1: Der Ergebnisdialog der Aktion zum Zeilenzählen

Die Methode setActivePart wird immer dann aufgerufen, wenn die Laufzeitumgebung von Eclipse eine neue Instanz der Klasse NumberOfLinesAction erzeugt. Wir verwenden die Methode, um uns eine Referenz auf den *Part* zu merken, für den die Aktion erzeugt wurde. Dies ist in unserem Fall die *View*, die die selektierte Java-Datei darstellt. Die Methode selectionChanged wird unmittelbar, bevor sich das Kontextmenü öffnet, aufgerufen. Unsere Implementierung speichert die übergebene Selektion in einer Instanzvariablen, damit die Methode run auf sie zugreifen kann. Alle drei Methoden haben einen zusätzlichen Parameter vom Typ IAction. Diese Instanz repräsentiert den Platzhalter, den die Laufzeitumgebung aufgrund der Beschreibung in der Manifest-Datei erzeugt hat. Die Instanz kann verwendet werden, um den visuellen Zustand (*name, enabled/disabled*, ...) der Aktion zu verändern.

Die Implementierung der Methode run konvertiert zunächst das erste Element der gespeicherten Selektion in ein Objekt vom Typ ICompilationUnit. Die beiden harten Typkonvertierungen in der Methode getCompilationUnit sind gefahrlos, da

- in der Manifest-Datei festgelegt ist, dass die Aktion nur für Elemente vom Typ ICompilationUnit erscheint,
- Kontextmenüerweiterungen vom Typ objectContribution voraussetzen, dass die *View* ihre Selektion in strukturierter Form anbietet.

Für das eigentliche Zählen der Zeilen wird ein Dokument mit dem Inhalt der selektierten Java-Datei erzeugt. Ein Dokument bietet nämlich bereits eine entsprechende Methode zum Zählen der Zeilen an. Anschließend wir das Ergebnis mit Hilfe eines Informationsdialogs präsentiert (siehe Abb. 1).

Die Leser, die den präsentierten Code bereits in ihr Programm übernommen haben, werden feststellen, dass dieser nicht fehlerfrei übersetzt werden kann. Woran liegt das? Wie bereits erwähnt, muss ein *Plug-In* angeben, von welchen anderen *Plug-Ins* es abhängt. Momentan referenziert unser *Plug-In* kein weiteres *Plug-In* (die Manifest-Datei hat kein requires-Element). Klassen, die den Zugriff auf Java-Ressourcen innerhalb des Workspace ermöglichen, werden durch das *Plug-In* „org.eclipse.jdt.core" und der Informationsdialog durch das *Plug-In* „org.eclipse.ui" zur Verfügung gestellt. Wir müssen also diese *Plug-Ins* in die Liste der benötigten *Plug-Ins* aufnehmen:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin id="ospektrum.plugin" ...>
...
  <requires>
    <import plugin="org.eclipse.ui"/>
    <import plugin="org.eclipse.jdt.core"/>
    <import plugin="org.eclipse.jdt.ui"/>
  </requires>
...
</plugin>
```

Noch einfacher geht dies mit dem *Plug-In Manifest Editor*: Wir öffnen unsere Manifest-Datei damit, wechseln zur *Dependencies*-Seite und fügen mit dem *Add*-Knopf die *Plug-Ins* „org.eclipse.ui", „org.eclipse.jdt.core" und „org.eclipse.jdt.ui" hinzu. Letzteres wird für zusätzliche Erweiterungen benötigt. Wenn wir nun *Save* auswählen, wird ein automatischer Bau des Projekts ausgelöst, wodurch alle Fehler behoben sein sollten.

Darstellung der Ergebnisse

Im nächsten Schritt wollen wir den modalen Dialog durch eine zweckmäßigere, nicht-modale Darstellungsart ersetzen. Dazu wählen wir die von der Eclipse-Plattform bereitgestellte Abstraktion *View*. *Views* unterstützen typischerweise Editoren, indem sie zusätzliche Informationen über selektierte Ressourcen oder Objekte zeigen und die Navigation innerhalb dieser ermöglichen.

Zur besseren Unterstützung einer geplanten späteren Erweiterung um weitere Metriken wählen wir für die *View* eine tabellarische Darstellungsform: Jede Zeile entspricht einer Metrik, die Spalten enthalten den Namen der Metrik, den Resultatwert sowie gegebenenfalls den Pfad der zu Grunde liegenden Java-Datei (siehe Abb. 2).

Wie bereits bei der NumberOfLinesAction beginnen wir die Implementierung auch hier mit einem deklarativen Teil in der Manifest-Datei des *Plug-Ins*:

```
<extension point="org.eclipse.ui.views">
  <view
    id="ospektrum.plugin.MetricsResultView"
    name="Metrics Results"
    class= "ospektrum.plugin.MetricsResultView"
    icon="icons/collectmetrics.gif" />
</extension>
```

Den von der Plattform definierten Erweiterungspunkt org.eclipse.ui.views füllen wir mit einem MetricsResultView. Auch diesen können wir bereits ohne Vorliegen der konkreten Implementierung testen. Allerdings müssen wir hierbei beachten, dass Eclipse *Views* im Normalfall nicht automatisch instanziiert und sichtbar macht, sondern dass sie explizit aktiviert werden müssen. Wir öffnen hierzu *Window* > *Show View* > *Other* und wählen im Dialog in der Rubrik *Other* die *View Metrics Results* aus (an dieser Stelle wird auch die Bedeutung des im obigen Erweiterungspunkt definierten Attributs icon deutlich). Die *View* wird in der Rubrik *Other* aufgelistet, da sie in der Manifest-Datei keiner Kategorie zugeordnet wurde.

Auch hier erscheint statt der *View* ein Fehlerdialog, der aber deutlich macht, dass die Benutzungsschnittstelle bereits ohne das Vorhandensein einer konkreten Implementierung funktioniert.

Wir wenden uns nun der Implementierung der MetricsResultView zu. Der Erweiterungspunkt erwartet, dass die *View* das Interface IViewPart implementiert. Damit wir aber nicht dessen 14 Methoden implementieren müssen, leiten wir unsere MetricsResultView von der Default-Implementierung ViewPart ab (siehe Listing 3).

Über die neu eingeführte Methode setInput wird unsere *View* mit den Resultaten der einzelnen Metriken initialisiert. Die Aggregation der einzelnen Ergebnisse erfolgt mittels einer ArrayList; ein einzelnes Resultat wird durch die Klasse MetricsResult repräsentiert:

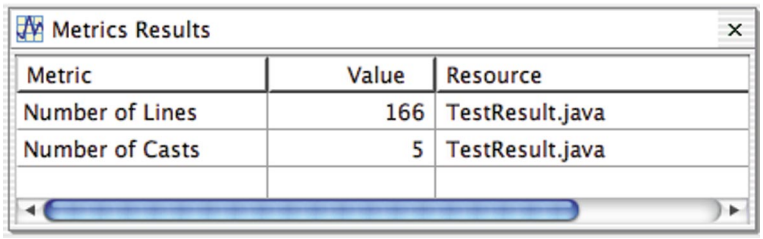

Abb. 2: Der MetricsResultView

```
package objektspektrum.plugin
public class MetricsResultView extends ViewPart {
  private TableViewer fViewer;
  // new method
  public void setInput(ArrayList results) {
    fViewer.setInput(results);
  }
  // overridden ViewPart.createPartControl
  public void createPartControl(Composite parent) {
    Table table= new Table(parent, SWT.SINGLE |
      SWT.H_SCROLL | SWT.V_SCROLL | SWT.BORDER |
      SWT.FULL_SELECTION);
    table.setHeaderVisible(true);
    table.setLinesVisible(true);
    TableColumn c= new TableColumn(table, SWT.NONE,0);
    c.setText("Metric"); c.setWidth(150);
    c= new TableColumn(table, SWT.NONE, 1);
    c.setText("Value"); c.setWidth(100);
    c.setAlignment(SWT.RIGHT);
    column = new TableColumn(table, SWT.NONE, 2);
    column.setText("Resource"); column.setWidth(300);
    fViewer= new TableViewer(table);
    fViewer.setContentProvider(
      new MetricsResultContentProvider());
    fViewer.setLabelProvider(
      new MetricsResultLabelProvider());
  }
}
```

Listing 3: Die Implementierung der MetricResultView

```
public class MetricsResult {
  private String fName;
  private int fValue;
  private ICompilationUnit fCU;
  public MetricsResult(String name, int value,
      ICompilationUnit cu) {
    fName= name; fValue= value; fCU= cu;
  }
  public String getName() { return fName; }
  public int getValue() { return fValue; }
  public ICompilationUnit getCU() { return fCU; }
}
```

Die Methode createPartControl wird von der Plattform aufgerufen, wenn die *View* sichtbar gemacht werden soll, d. h. ihre *Widgets* in die *Widget*-Hierarchie des Workbench-Fensters eingebettet werden müssen. Wir erzeugen hier zunächst ein Tabellen-*Widget* und initialisieren seine drei Spalten mit entsprechendem Titel und Breite.

Anschließend könnten wir die Tabelle mit den Daten unserer Metriken füllen. Doch wollen wir hier einer sauberen Trennung von Darstellung und Modell gemäß dem *Model/View/Controller*-Modell den Vorzug geben. Dazu verwenden wir einen TableViewer, wie er uns von der Eclipse-Teilkomponente *JFace* zur Verfügung gestellt wird. Ein TableViewer „vermittelt" gewissermaßen zwischen einem einfachen Tabellen-*Widget* und dem Zugriff auf ein abstraktes zweistufiges Modell, repräsentiert durch einen ContentProvider und einen LabelProvider. Der ContentProvider definiert das API für den Zugriff auf die Zeilen eines beliebigen Modells, ein LabelProvider das API für den Zugriff auf die Spalten einer Zeile sowie für die Darstellung jeder Zelle:

```
public class MetricsResultContentProvider
            implements IStructuredContentProvider {
  public Object[] getElements(Object inputElement) {
    if (inputElement instanceof List)
      return ((List)inputElement).toArray();
    return null;
  }
  // leere Implementierung weiterer Methoden
}
```

Für unser Beispiel müssen wir hierzu nur die Methode getElements so implementieren, dass sie die Einzelergebnisse unseres MetricsResults als Array zurückliefert.

In einem zweiten Schritt muss nun ein Objekt vom Typ MetricsResult auf die einzelnen Spalten der Tabelle abgebildet werden. Dazu wird in der Methode getColumnText des MetricsResultLabelProvider für jeden Spaltenindex der Tabelle ein entsprechendes Attribut des MetricsResults als String zurückgegeben (**Listing 4**).

Als letzten Schritt verbinden wir unsere MetricsView mit der zuvor implementierten NumberOfLinesAction. In der Methode run erzeugen wir ein MetricsResult mit dem Ergebnis der Metrik NumberOfLines, lokalisieren unseren MetricsResultView über dessen Identifikation innerhalb der aktiven Seite der Workbench und initialisieren ihn mit einer Liste von MetricsResults (**Listing 5**).

Zum Schluss wollen wir noch dafür sorgen, dass durch einen Doppelklick auf eine Zeile des MetricsResultsViews die korrespondierende Java-Datei (letzte Spalte der Tabelle) in einem Editor geöffnet wird (**Listing 6**). Dazu installieren wir auf dem Tabellen-*Widget* einen SelectionListener und implementieren die im Falle eines Doppelklicks aufgerufene Methode widgetDefaultSelected.

```
public class MetricsResultLabelProvider
            implements ITableLabelProvider {
  public String getColumnText(Object element,
              int columnIndex) {
    MetricsResult result= (MetricsResult) element;
    switch (columnIndex) {
    case 0:
      return result.getName();
    case 1:
      return Integer.toString(result.getValue());
    case 2:
      return result.getCU().getElementName();
    }
    return null;
  }
  // leere Implementierung weiterer Methoden
}
```

Listing 4: Die Implementierung des MetricResultLabelProviders

Wir extrahieren zunächst das ausgewählte MetricsResult aus dem SelectionEvent, daraus dann die Java-Datei und versuchen diese mit einem Editor zu öffnen. Hierzu benutzen wir die Methode openInEditor, die uns von der Klasse JavaUI zur Verfügung gestellt wird.

Eigene Erweiterungspunkte

Nachdem wir bisher nur die Erweiterungspunkte anderer *Plug-Ins* benutzt haben, wollen wir nun für unser *Plug-In* einen eigenen Erweiterungspunkt definieren. Über diesen sollen weitere Metriken hinzugefügt werden können, d. h. aus unserem sehr spezifischen *Plug-In* zum Zählen von Codezeilen wird nun ein allgemeineres Metrik-*Plug-In*.

Auch die Definition eines neuen Erweiterungspunkts beginnt wieder in der Manifest-Datei unseres Metrik-*Plug-Ins*: Mittels eines *Tags* extension-point wird der neue Erweiterungspunkt mit einer eindeutigen Identifikation sowie einem lesbaren Namen versehen:

```
<plugin id="ospektrum.plugin" . . . >
  <extension-point
    id= "metrics"
    name= "A metric">
</plugin>
```

Als nächstes müssen wir nun die Abstraktion finden, die der Erweiterungspunkt repräsentieren soll. Da es das Ziel ist, neue Metriken hinzufügen zu können, betrachten wir unsere existierende Metrik des „Zeilen-Zählens" etwas genauer:

```
ICompilationUnit cu= . . .
ArrayList results= . . .
try {
  Document doc= new Document(cu.getSource());
  results.add(new MetricsResult("Number of Lines",
      doc.getNumberOfLines(), cu));
} catch (CoreException ex) {
  // error handling
```

```
public void run(IAction action) {
  IStructuredSelection selection=
            (IStructuredSelection) fSelection;
  ICompilationUnit cu= (ICompilationUnit)
            selection.getFirstElement();
  ArrayList results= new ArrayList();
  try {
    Document doc= new Document(cu.getSource());
    results.add(new MetricsResult("Number of Lines",
        doc.getNumberOfLines(), cu));
  } catch (JavaModelException ex) {
    // error handling
  }
  // show results
  IWorkbenchPage page= fWorkbenchPart.getSite().
      getWorkbenchWindow().getActivePage();
  IViewPart vp=
      page.showView("ospektrum.plugin.
      MetricsResultView");
  if (vp instanceof MetricsResultView)
    ((MetricsResultView)vp).setInput(results);
}
```

Listing 5: Die Implementierung der Methode NumberOfLinesAction.run()

```
public void createPartControl(Composite parent) {
  // ...
  table.addSelectionListener(new SelectionAdapter() {
    public void widgetDefaultSelected
          (SelectionEvent e) {
      Object data= e.item.getData();
      if (! (data instanceof MetricsResult))
        return;
      ICompilationUnit cu=
              ((MetricsResult)object).getCU();
      if (cu == null)
        return;
      try {
        JavaUI.openInEditor(cu);
      } catch (CoreException ex) {
        // error handling
      }
    }
  }
  // ...
}
```

Listing 6: Die Implementierung der Methode MetricsResultView.create Part Control()

}

Die Metrik verwendet als Eingabe ein Objekt vom Typ ICompilationUnit und liefert als Ergebnis ein MetricsResult. Außerdem können bei der Anwendung der Metrik CoreExceptions auftreten. Ein – zugegebenermaßen sehr vereinfachender – Abstraktionsprozess führt dann zu folgender Schnittstelle einer Metrik:

```
public interface IMetric {
  MetricsResult process(ICompilationUnit element)
                    throws CoreException;
}
```

Wir erwarten damit von jeder Implementierung unseres neuen Erweiterungspunkts, dass sie das Interface IMetric implementiert.

Als erstes Anwendungsbeispiel für unseren Erweiterungspunkt nehmen wir nun unsere bisher fest eingebaute Metrik und wandeln sie in die Implementierung eines Erweiterungspunktes um. D. h. wir unterscheiden nicht zwischen fest eingebauten Metriken und von außen hinzugefügten. Dieses scheinbare Detail ist tatsächlich Ausdruck des wichtigen Eclipse-Prinzips der Selbstgenügsamkeit: Wir benutzen intern die gleichen Mechanismen, die wir auch anderen – externen – Entwicklern zur Verfügung stellen.

Unsere neue Klasse NumberOfLinesMetric sieht nun als Implementierung des Interfaces IMetric folgendermaßen aus:

```
public class NumberOfLinesMetric implements IMetric {
  public MetricsResult process(
      ICompilationUnit element) throws CoreException {
    Document doc= new Document(element.getSource());
    return new MetricsResult("Number of Lines",
                  doc.getNumberOfLines(), element);
  }
}
```

Damit diese nun in unserem *Plug-In* verwendet werden kann, müssen wir sie selbstverständlich in unserer Manifest-Datei deklarieren:

```
<plugin id="ospektrum.plugin" . . . >
  <extension point="ospektrum.plugin.metrics">
    <metric
      class= "ospektrum.plugin.NumberOfLinesMetric"/>
  </extension>
</plugin>
```

Man beachte, dass sich die Deklaration von der weiter oben erfolgten Definition eines Erweiterungspunktes nur im Bindestrich in extension-point unterscheidet!

Außerdem muss die Identifikation des Erweiterungspunkts immer mit der Identifikation des umgebenden *Plug-Ins* qualifiziert werden, d. h. für unser Beispiel lautet der Erweiterungspunkt ospektrum.plugin.metrics, obwohl er zuvor nur als metrics definiert wurde.

Im nächsten Schritt müssen wir nun unsere existierende NumberOfLinesAction so abändern, dass sie nicht mehr „fest verdrahtet" die Anzahl von Zeilen zählt, sondern alle Implementierungen unseres Erweiterungspunktes findet und ausführt. Außerdem geben wir der Aktion den passenderen Namen ComputeMetricsAction.

Hier kommt nun ein weiteres wichtiges Eclipse-Prinzip zur Anwendung: das „Add, not Replace"-Prinzip. Erweiterungspunkte sollten immer so definiert werden, dass sie mehrere Erweiterungen zulassen, d. h. eine externe Erweiterung sollte niemals eine interne Implementierung ersetzen können.

Das Sammeln aller Implementierungen unseres Erweiterungspunktes erfolgt in einer neuen Hilfsmethode collectAllMetrics der ComputeMetricsAction:

```
IMetric[] collectAllMetrics() throws CoreException {
  IPluginRegistry registry=
                Platform.getPluginRegistry();
  IConfigurationElement[] elements=
            registry.getConfigurationElementsFor(
              "ospektrum.plugin", "metrics");
  IMetric[] result= new IMetrics[elements.length];
  for (int i= 0; i < elements.length; i++)
    result[i]= (IMetrics)
        elements[i].createExecutableExtension("class");
  return result;
}
```

Aus der *Plug-In*-Registratur der Plattform lassen wir uns über den qualifizierten Namen alle Implementierungen unseres neuen Erweiterungspunktes als Feld von IConfigurationElementen geben. Die Registratur wird beim Starten von Eclipse mit Informationen aus den Manifest-Dateien aller *Plug-Ins* gefüllt, ohne dass bereits der korrespondierende *Plug-In*-Code geladen werden muss.

Mit der Methode createExecutableExtension wird der unter dem Attribut class gespeicherte Klassenname benutzt, um eine Instanz der entsprechenden Metrik zu erzeugen und damit die *Plug-In*-Aktivierung auszulösen.

Das eigentliche Ausführen der Metriken in der run-Methode der ComputeMetricsAction sieht nun folgendermaßen aus:

```
public void run(IAction action) {
  ICompilationUnit cu= getCompilationUnit();
  List results= new ArrayList();
  try {
    IMetric[] allMetrics= collectAllMetrics();
    for (int i= 0; = allMetrics.length; i++)
      results.add(allMetrics[i].process(cu));
    // Darstellung der Ergebnisse
  } catch (CoreException ex) {
    // error handling
  }
}
```

Wir sammeln und erzeugen zunächst alle Metriken und wenden sie dann auf die ausgewählte Java-Datei an. Das von jeder Metrik gelieferte Ergebnis sammeln wir in einer Liste, die wie bisher im MetricsResultView dargestellt wird.

Da die Methode collectAllMetrics erst beim Ausführen von ComputeMetricsAction aufgerufen wird, ist sichergestellt, dass alle *Plug-Ins*, die eine Metrik bereitstellen, erst bei Bedarf, d. h. verzögert geladen werden.

Eine weitere Metrik

Der neue Erweiterungspunkt ospektrum.plugin.metrics gibt uns nun die Möglichkeit neue Metriken zum System hinzuzufügen. Wir wollen diese Möglichkeit nutzen, um die Anzahl der harten Typkonvertierungen in einer Java-Datei zu ermitteln. Eine einfache Lösung wäre es nach Textausdrücken der Form *„(.*)"* in einem Programm zu suchen. Dies würde allerdings ein ungenaues Resultat liefern, da auch Ausdrücke der Form *„(1 + 1)"* als Typkonvertierung interpretiert würden. Wir müssen also einen Weg finden, die grammatikalische Struktur eines Java-Programms zu verstehen.

Compiler verwenden üblicherweise eine abstrakte Syntax, um ihre interne Repräsentation eines Programms zu spezifizieren. Typische Elemente einer abstrakten Syntax sind beispielsweise Anweisungen (*Statements*), Ausdrücke (*Expressions*) oder Kennzeichner (*Identifier*). Ist die interne Repräsentation als Baum organisiert, spricht man von einem abstrakten Syntax-Baum. Der abstrakte Syntax-Baum eines Java- ▶

Programms hat einen Wurzelknoten, der die Java-Datei repräsentiert. Dieser besitzt *n* Unterknoten für die in der Datei deklarierten Typen. Die Unterknoten eines Typknotens sind wiederum Knoten für Felder, Methoden, geschachtelte Typen usw.

Das Eclipse *Plug-In* org.eclipse.jdt.core bietet die notwendigen Klassen an, um von einer Java-Datei einen abstrakten Syntax-Baum[2]) zu bauen. Die Klassen befinden sich in dem Packet org.eclipse.jdt.core.dom. Ihre Verwendung diskutieren wir am besten an dem Beispiel, das die Anzahl der Typkonvertierungen einer Java-Datei bestimmt (**Listing 7**).

Die Methode parseCompilationUnit der Klasse AST (*AST* steht für *Abstract Syntax Tree*) erzeugt den abstrakten Syntax-Baum einer Java-Datei. Der Wurzelknoten ist vom Typ CompilationUnit. Ein konkreter Baum kann mit Hilfe eines Besuchers (*Visitor*) traversiert werden. Der von Eclipse bereitgestellte Besucher für abstrakte Syntax-Bäume heißt ASTVisitor und enthält für jeden Knotentyp eine entsprechende visit-Methode. Der Rückgabewert der Methode visit kontrolliert dabei, ob die Kinderknoten ebenfalls besucht werden (Rückgabewert true) oder nicht (Rückgabewert false).

Wir bilden also eine Unterklasse von ASTVisitor und re-implementieren die Methode visit(CastExpression), da wir ja die Anzahl der Typkonvertierungen zählen wollen. Mittels der Methode accept wenden wir den Besucher auf den Wurzelknoten an und liefern als Ergebnis der Berechnung eine Instanz der Klasse MetricsResult zurück.

Zum Schluss fügen wir unsere NumberOfCastsMetric in einem neuen *Plug-In* moremetrics zum System hinzu:

```
<plugin id="ospektrum.moremetrics" . . . >
...
  <requires>
    ...
    <import plugin="ospektrum.plugin"/>
  </requires>
  <extension point="ospektrum.plugin.metrics">
    <metric
      class= "ospektrum.moremetrics.NumberOfCastsMetric"/>
  </extension>
...
</plugin>
```

Plug-Ins publizieren

Der letzte Schritt auf dem Weg zu einem erfolgreichen *Plug-In* besteht darin, es für andere verfügbar zu machen. Da ein Softwareprodukt typischerweise nicht nur aus einem *Plug-In* besteht, wurden zur Softwareverteilung so genannte Features eingeführt. Ein Feature gruppiert mehrere *Plug-Ins* in eine logische Einheit und enthält zusätzliche Informationen, wie z.B. den obligatorischen Lizenzvertrag. Um ein Feature erfolgreich bauen zu können, muss jedes darin enthaltene *Plug-In* noch deklarieren, aus welchen Teilen es besteht. Diese Informationen sind in der Datei „build.properties" des jeweiligen *Plug-Ins* gespeichert. Wird ein Projekt mit dem entsprechenden PDE-Assistenten angelegt, so existiert die Datei bereits. Für unser Metrik-*Plug-In* muss die Datei wie folgt aussehen:

```
source.plugin.jar = src/
bin.includes = plugin.xml, *.jar, icons/
```

Die Einträge haben folgende Bedeutung:

- source.plugin.jar beschreibt, aus welchen Quelldateien das Archiv plugin.jar erzeugt wird. In der Regel ist dies eine Aufzählung der Quellcode-Ordner eines Projekts.
- bin.includes listet alle Dateien und Ordner auf, die zum Plug-In gehören.

Für unser Metrik-*Plug-In* erstellen wir nun ein Feature, indem wir den *New Feature*-Assistenten (*File* > *New* > *Project*, *Plug-In Development-Feature Project*) verwenden. Als Namen geben wir „ospektrum-feature" ein und auf der Seite *Referenced Plug-Ins and Fragments* wählen wir das „ospektrum.plugin" aus. Der Assistent generiert das Feature-Projekt sowie das Feature-Manifest „feature.xml" und öffnet automatisch den *Feature Manifest Editor*. Damit wir ein für Eclipse legales Feature haben, müssen wir auf der *Information*-Seite in der *Section* den Eintrag *License Agreement* auswählen und einen Lizenztext eingeben. Wie das Feature-Manifest nach Abschluss dieser Schritte aussieht, zeigt **Listing 8**.

Als Update-Site kann jeder Ort verwendet werden, der über eine URL adressierbar ist, wie beispielsweise ein Ordner auf einem Web-Server. Der interne Aufbau einer Update-Site ist simpel. Sie besteht aus Features und *Plug-Ins*, deren Inhalte in Archiven zusammengefasst sind. Die Versionsnummer ist im Namen des Archivs kodiert. Das Site-Manifest beschreibt den Inhalt der Update-Site und sieht für unser Metrik-Feature wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
  <feature
      url ="features/ospektrum-feature_1.0.0.jar"
      id ="ospektrum.feature" version="1.0.0"/>
</site>
```

```
public class NumberOfCastsMetric implements IMetric {
  static class CastVisitor extends ASTVisitor {
    public int result= 0;
    public boolean visit (CastExpression node) {
      result++;
      return super.visit(node);
    }
  }
  public MetricsResult process (IcompilationUnit
                        element) throws CoreException {
    CompilationUnit root=
          AST.parseCompilationUnit(element, false);
    CastVisitor visitor= new CastVisitor();
    root.accept(visitor);
    return new MetricsResult("Number of Casts",
              visitor.result, element);
  }
}
```

Listing 7: Die Implementierung der NumberOfCastsMetric

Eine Update Site erzeugen wir am einfachsten mit Hilfe von PDE. Zunächst öffnen wir den *New Site*-Assistenten (*File* > *New* > *Project* > *Plug-In Development Update-Site Project*); als Projektnamen verwenden wir „ospektrum-updateSite". Nach Betätigen des *Finish*-Knopfes öffnet der Assistent automatisch das Site-Manifest im *Site Manifest Editor*. Dem Site-Manifest fügen wir unser Metrik-Feature hinzu, indem wir auf der *Build*-Seite den *Add*-Knopf betätigen und „ospektrum.feature" selektieren. Da wir das Feature für Klienten sichtbar machen wollen, müssen wir es in der Liste noch mit einem Häkchen versehen. Die Update-Site sowie alle angegebenen Features und *Plug-Ins* werden erstellt, sobald man auf der *Build*-Seite des Editors auf den Knopf *Build* drückt.

Jeder, der nun über das Netzwerk oder Internet (HTTP) auf diesen Ort zugreifen kann, hat die Möglichkeit, die verfügbaren Features und deren *Plug-Ins* anzuschauen und zu installieren.

Zum Beweis öffnen wir in Eclipse den Update-Manager (*Help* > *Software Updates* > *Update Manager*) und wählen im *My Computer*-Navigator (*View Feature Updates*) den Ordner des momentanen Workspace aus. Darin erscheint unsere Update-Site mit einer speziellen Ikone ( ). In der Kategorie *Other* finden wir unser Metrik-Feature, wählen es aus und installieren es (*Install Now*-Knopf auf der rechten Seite, **siehe Abb. 3**). Nach erfolgreicher Installierung wird Eclipse

[2]) *Die von Eclipse erzeugte interne Repräsentation eines Programms ist in Wahrheit eine Mischung eines Parse-Baums und eines abstrakten Syntax-Baums. So enthält der Baum Knoten für Klammern (zum Beispiel für den Ausdruck „(1 + 1)") oder Informationen zu Javadoc-Kommentaren, die in einem reinen abstrakten Syntax-Baum eigentlich nicht mehr vorkommen. Wir sprechen aber dennoch von einem abstrakten Syntax-Baum.*

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
    id="ospektrum-feature"
    label="ospektrum-feature"
    version="1.0.0"
    provider-name="">
  <license>
    Meine Lizenz
  </license>
  <requires>
    <import plugin="org.eclipse.ui"/>
    <import plugin="org.eclipse.jdt.core"/>
    <import plugin="org.eclipse.jdt.ui"/>
  </requires>
  <plugin
    id="ospektrum.plugin"
    download-size="0"
    install-size="0"
    version="1.0.0"/>
</feature>
```

Listing 8: Die Manifest-Datei des Metrik-Features

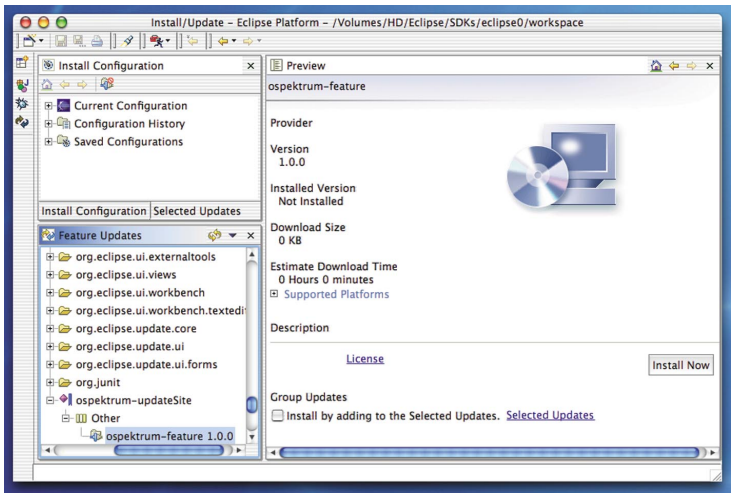

Abb. 3: Der Updatemanager in Aktion

neu gestartet und die neuen Metrikfunktionen können benutzt werden.

Was haben wir vereinfacht?

Wie üblich bei einem Artikel dieser Länge haben wir, um die Verständlichkeit zu verbessern, das eine oder andere vereinfacht. Im Folgenden wollen wir die wichtigsten Punkte kurz aufzählen.

- Momentan können Metriken nur für Java-Dateien berechnet werden. Es sind aber auch Metriken für andere Java-Elemente, wie *Packages* oder Typen denkbar. Um dies zu ermöglichen, müsste der Erweiterungspunkt generalisiert und mit einem weiteren Attribut, ähnlich dem Attribut objectClass des Erweiterungspunktes für Kontextmenüs, versehen werden.
- Eigentlich möchte man Metriken für einen ganzen Workspace oder für mehrere Projekte berechnen und nur die Java-Elemente in einer *View* anzeigen, für die der errechnete Wert einen gewissen Schwellwert übersteigt. Um über die Java-Elemente eines Workspace zu iterieren, bietet das *Plug-In* „org.eclipse.jdt.core" entsprechende Klassen an. Lohnenswert ist dabei ein Blick auf die Klasse JavaCore sowie alle Unterklassen von IJavaElement.
- Präferenzseiten wären bestens geeignet, um die Schwellwerte einer Metrik zu konfigurieren. Als Einstieg zum Thema Präferenzseiten sei hier der Artikel „Preferences in the Eclipse Workbench UI" (www.eclipse.org/articles/Article-Preferences/preferences.htm) empfohlen.
- Im vorgestellten Programmcode werden alle Ausnahmesituationen ignoriert. Dieses Vorgehen ist für ein produktives *Plug-In* nicht akzeptabel. Um dies zu verbessern, verweisen wir den Leser auf die Klassen ILog und ErrorDialog, die den Umgang mit Ausnahmesituationen vereinfachen.
- Die Deluxe-Variante würde die Metriken automatisch bei jedem Bau eines Java-Projekts inkrementell aktualisieren. Eclipse ermöglicht es anderen *Plug-Ins*, am Übersetzungsvorgang zu partizipieren. Wertvolle Informationen liefern hier die Online-Dokumentation zum Erweiterungspunkt org.eclipse.core.resources.builders sowie Unterklassen von IncrementalProjectBuilder.
- Die momentane Implementierung unterscheidet nicht zwischen internen Klassen und Klassen, die andere *Plug-Ins* für ihre eigene Implementierung verwenden können. In Eclipse wird diese Trennung durch Verwendung geeigneter *Package*-Namen erreicht. Klassen, die nicht Teil des APIs sind, haben in ihrem Paket-Namen die Zeichenkette „internal". In unserem Beispiel sollten also die Klassen IMetric und MetricsResult in einem API-Paket und alle anderen Klassen in einem internen Paket liegen.
- An manchen Stellen haben wir bereits das PDE verwendet, um spezielle Aktionen einfacher auszuführen. PDE bietet neben den beschriebenen Aktionen insbesondere komfortable Editoren zum Bearbeiten von *Plug-In*-spezifischen XML-Dateien an. Wird zum Beispiel der *Plug-In Manifest Editor* verwendet, um einen neuen Erweiterungspunkt zu definieren, so unterstützt der Editor den Anwender bei der Definition eines entsprechenden Schemas.

Vielleicht konnten diese Punkte Anregungen geben, wie Leser das hier vorgestellte *Plug-In* weiterentwickeln können.

Fazit

Der Beitrag hat entlang eines Beispiels alle wichtigen Schritte, die bei der Entwicklung eines *Plug-Ins* notwendig sind, demonstriert. Die gewonnen Erfahrungen lassen sich folgendermaßen zusammenfassen:

- Die Eclipse-Plattform verfügt über mächtige, aber dennoch einfach benutzbare Erweiterungsmechanismen.
- Die Architektur der Plattform ist so ausgelegt, dass sie auch mit einer großen Anzahl von *Plug-Ins* umgehen kann.
- Eclipse macht selbst intensiv Gebrauch von seinen eigenen Erweiterungsmöglichkeiten. Dies garantiert die gute Qualität der Plattform und seiner Architektur.
- Das *Java Development Environment* zusammen mit PDE machen Eclipse zu einer idealen Entwicklungsumgebung, um *Plug-Ins* für Eclipse zu entwickeln.

So bleibt nur noch, den Lesern viel Spaß bei der Entwicklung ihres ersten *Plug-Ins* zu wünschen.

Wir danken Erich Gamma für sein konstruktives Feedback.